DAY 1

Introduction to Immutable State Patterns in JavaScript

What Is Immutability?

Immutability means you **do not modify (mutate) data directly**.

Instead, when data needs to change, you create a **new version** of the object or array — leaving the original untouched.

Why Use Immutable State?

Even outside of frameworks like Angular, immutability brings several benefits:

Benefit	Description
Predictable State	Changes are clearer and easier to track
Undo/Redo Friendly	History of changes can be preserved
Testable Logic	Pure functions with no side effects
Prevents Bugs	Avoids unintended side effects from shared references
Better for UI Frameworks	Helps UI libraries detect when to update the screen

Core Rule

- Don’t mutate existing data
- Always create a **new copy**

Common Immutable Operations

```
const user = { name: 'Ali' };
user.name = 'Ahmad';  //  Mutation
```

Creating a new object:

```
const user = { name: 'Ali' };
const updatedUser = { ...user, name: 'Ahmad' };  // Immutable
```

Arrays

Pushing directly:

```
const items = [1, 2];
items.push(3);  //  Mutation
```

Adding immutably:

```
const items = [1, 2];
const newItems = [...items, 3];  //  Immutable
```

Updating or Removing in Arrays

Map to update an item:

```
const items = [{ id: 1, name: 'A' }, { id: 2, name: 'B' }];
const updatedItems = items.map(item =>
  item.id === 2 ? { ...item, name: 'Updated' } : item
);
```

Filter to remove an item:

```
const filtered = items.filter(item => item.id !== 1);
```

Sample: Immutable vs Mutable Behavior

```
let person = { name: 'Ali', age: 30 };

//  Mutable
function mutatePerson(p) {
  p.age = 31;
  return p;
}

//  Immutable
function updatePerson(p) {
  return { ...p, age: 31 };
}
```

Why This Matters

- Mutating changes the original object → can cause side effects
- Immutable updates keep the original intact → safer and more predictable

Common Anti-Patterns & Replacements

Anti-pattern	Immutable Alternative
arr.push(item)	[...arr, item]
arr.splice(...)	arr.filter(...)
obj.key = val	{ ...obj, key: val }
delete obj.key	Destructure to remove key manually

Bonus: Libraries That Help

- `immer`: Write code that looks like mutation, but produces immutable updates
- `lodash/fp`: Functional tools for safer data handling
- `immutable.js`: Persistent data structures

Summary

- Treat data as **read-only** by default
- Use **object spreading (`{ ...obj }`)** and **array spreading (`[...arr]`)** to update data
- Immutability helps write cleaner, safer, and more UI-friendly code

Source code reference:

https://github.com/wanmuz86/angular-int-adv-lab1-intermediatejs/tree/main

Component Communication in Angular

Angular apps are made up of components arranged in a tree. Communication between these components is essential and happens in several ways:

1. `@Input()` – Pass Data from Parent to Child

Used when a parent **component** needs to send data to a **child component**.

Syntax:

```
// child.component.ts
@Input() title: string;
```

```
<!-- parent.component.html -->
<app-child [title]="'Hello from parent'"></app-child>
```

Use Case:

- Passing values like strings, numbers, arrays, or objects to a reusable component.

2. @Output() – Emit Events from Child to Parent

https://angular.dev/api/core/Output

Used when the child **component** wants to notify the **parent component** of something (e.g., a button was clicked).

Syntax:

```


// child.component.ts
@Output() clicked = new EventEmitter<string>();

buttonClicked() {
  this.clicked.emit('Child clicked me!');
}
```

```
<!-- parent.component.html -->
<app-child (clicked)="handleChildEvent($event)"></app-child>
```

Use Case:

- Forms, buttons, selections in the child that need to update something in the parent.

3. `@ViewChild()` – Access Child Component or DOM Element from Parent

https://angular.dev/api/core/ViewChild

Allows the **parent component** to directly access a **child component's methods/properties** or native DOM element.

Syntax:

```
// parent.component.ts
@ViewChild(ChildComponent) childRef: ChildComponent;

ngAfterViewInit() {
  this.childRef.someChildMethod();
}
```

Use Case:

- Trigger a method or get data from a child component directly.
- Accessing native HTML elements like inputs for focus or value retrieval.

4. Shared Services with Observables – Communicate Across Unrelated Components

A service can be used to **share data or events across components**, even if they aren't parent-child.

Example:

```
```

```
// shared.service.ts

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

  private dataSubject = new BehaviorSubject<string>('initial value');
  data$ = this.dataSubject.asObservable();

  setData(data: string) {
    this.dataSubject.next(data);
  }
}
```

```
// component-a.ts (sender)
this.sharedService.setData('New Value');
```

```
// component-b.ts (receiver)
this.sharedService.data$.subscribe(data => {
  console.log(data); // "New Value"
});
```

Use Case:

- Communication between components on different routes.
- State sharing across unrelated parts of the app.

Summary

Technique	Direction	Use Case
`@Input()`	Parent to Child	Pass data down
`@Output()`	Child to Parent	Send events or data up
`@ViewChild()`	Parent to Child	Access child's method or DOM element

Shared Service	Any to Any	Communicate between non-related components

Lab: Component Communication in Angular

Objective:

Learn how Angular components communicate using:

- `@Input()` (Parent to Child)
- `@Output()` (Child to Parent)
- `@ViewChild()` (Access Child in Parent)
- Shared Services (Across unrelated components)

PART 1: `@Input()` – Parent to Child Communication

Create a new project

```
ng new lab2-input-output
```

Step 1.1: Create Components

```
ng generate component parent
ng generate component child
```

Step 1.2: Modify Child Component

`child.component.ts`

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

@Component({
  selector: 'app-child',
  template: `<p>Message from Parent: {{ message }}</p>`,
})
export class ChildComponent {
  @Input() message: string = '';
}
```

Step 1.3: Modify Parent Component

parent.component.html

```
<h2>Parent Component</h2>
<app-child [message]="'Hello from Parent!'"></app-child>
```

parent.component.ts

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

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

}
```

Call the app-parent inside app.component.html

```
<app-parent></app-parent>
```

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ParentComponent } from "./parent/parent.component";

@Component({
 selector: 'app-root',
 imports: [ ParentComponent],
 templateUrl: './app.component.html',
 styleUrl: './app.component.css'
})
export class AppComponent {
 title = 'lab2-input-output';
```

```
}
```

PART 2: `@Output()` – Child to Parent Communication

Step 2.1: Update Child Component

`child.component.ts`

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

@Component({
  selector: 'app-child',
  template: `<button (click)="sendMessage()">Send to Parent</button>`,
})
export class ChildComponent {
  @Output() notify = new EventEmitter<string>();

  sendMessage() {
    this.notify.emit('Message from Child!');
  }
}
```

`child.component.html`

```
<p>Message from parent {{message}}</p>
<button (click)="sendMessage()">Send Message to parent</button>
```

Step 2.2: Handle Event in Parent

`parent.component.html`

```
<h2>Parent Component</h2>
<app-child (notify)="receiveMessage($event)"></app-child>
<p>{{ childMessage }}</p>
```

`parent.component.ts`

```
childMessage = '';
```

```
receiveMessage(message: string) {
  this.childMessage = message;
}
```

PART 3: `@ViewChild()` – Access Child from Parent

Step 3.1: Add Method to Child

`child.component.ts`

```
sayHello() {
  console.log('Hello from Child Component!');
}
```

Step 3.2: Call from Parent using `@ViewChild`

`parent.component.ts`

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

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html',
})
export class ParentComponent implements AfterViewInit {
  @ViewChild(ChildComponent) childRef!: ChildComponent;

  ngAfterViewInit() {
    this.childRef.sayHello();
  }
}
```

PART 4: Shared Service for Communication

Generate the components (sender/receiver)

```
ng generate component sender
ng generate component receiver
```

Step 4.1: Create Service

```
ng generate service shared
```

shared.service.ts

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class SharedService {
  private messageSource = new BehaviorSubject<string>('Default message');
  message$ = this.messageSource.asObservable();

  changeMessage(newMessage: string) {
    this.messageSource.next(newMessage);
  }
}
```

Step 4.2: Use in Sender Component

sender.component.ts

```
constructor(private sharedService: SharedService) {}

sendData() {
  this.sharedService.changeMessage('Data from Sender Component');
}
```

sender.component.html

```
<h3>Shared Service - Sender</h3>
<button (click)="sendMessage()">Send Message</button>
```

Step 4.3: Use in Receiver Component

receiver.component.ts

```
import { Component, OnInit } from '@angular/core';
import { SharedService } from '../shared.service';
```

```
@Component({
 selector: 'app-receiver',
 imports: [],
 templateUrl: './receiver.component.html',
 styleUrl: './receiver.component.css'
})
export class ReceiverComponent implements OnInit {
  currentMessage:string = '';

 constructor(private sharedService: SharedService) {

  }

 ngOnInit(): void {
   this.sharedService.message$.subscribe(message => {
     this.currentMessage = message;
   } )
 }

}
```

receiver.component.html

```
<h3>Receiver component</h3>
<p>Passed message: {{currentMessage}}</p>
```

app.component.html

```
<h2>Parent - Child Communication</h2>
<app-parent></app-parent>

<h2>Communication through service</h2>
<app-sender></app-sender>
<app-receiver></app-receiver>
```

Conclusion

You should be able to:

- Display data passed via `@Input`
- Capture events with `@Output`
- Trigger a method using `@ViewChild`
- Share data using a service between `Sender` and `Receiver`

Additional Challenge:

- Add a form in the parent and pass input values to the child dynamically using `[(ngModel)]`.
- Combine `@Input` and `@Output` for a two-way binding effect using a pattern.

Full source code for the lab

https://github.com/wanmuz86/angular-int-adv-lab2-inputoutputrecap

Angular Change Detection: Default vs OnPush

Change Detection is the process by which Angular updates the DOM when data in your application changes.

https://angular.dev/best-practices/runtime-performance
https://angular.dev/api/core/ChangeDetectionStrategy
https://blog.angular-university.io/onpush-change-detection-how-it-works/
https://blogs.halodoc.io/understanding-angular-change-detection-strategy/

1. Default Change Detection Strategy

Behavior:

Angular traverses the entire component tree (top-down) **on every change detection cycle**, including when:

- An event occurs (click, input, etc.)
- A `setTimeout`, `setInterval`, or `Promise` resolves
- An HTTP request completes
- A value is updated in a service

Angular uses **Zone.js** to intercept all async events and run CD automatically.

Pros:

- Simple and automatic
- Works with mutable objects

Cons:

- **Can be inefficient** in large applications
- Even components with no data changes may be checked

2. OnPush Change Detection Strategy

Behavior:

- Angular **only checks the component when**:
  - An `@Input()` reference changes (not just mutated)
  - An event is emitted from inside the component (e.g., click)
  - `ChangeDetectorRef.markForCheck()` is called manually

How to Enable:

```
@Component({
  selector: 'app-my-component',
  changeDetection: ChangeDetectionStrategy.OnPush,
  templateUrl: './my-component.component.html',
})
export class MyComponent { ... }
```

Pros:

- **Better performance: skips unnecessary checks**
- **Ideal for presentational, stateless, and pure components**
- **Works best with immutable state patterns**

Cons:

- Changes to **mutated objects** may not trigger updates
- Requires better understanding of immutability and reference changes

Example: OnPush vs Default

Example with `OnPush`:

```
@Input() user: { name: string };
```

```
ngOnChanges() {
  // Updates only if a new object reference is passed
}
```

```
// This will NOT trigger OnPush update:
this.user.name = 'New Name';

// This WILL trigger OnPush update:
this.user = { name: 'New Name' }; // new reference
```

Summary Table

Strategy	When CD Runs	Works with Mutations?	Performance
Default	Any change	Yes	Medium
OnPush	On Input ref change, local event, or manual trigger	No	High

Best Practices

- Use **OnPush** for **presentational/stateless components**
- Use **immutability** (`Object.assign`, spread operator) to update inputs
- Avoid mutating objects/arrays directly when using OnPush

Lab: Angular Change Detection (Default vs OnPush)

Objective:

Understand how Angular change detection works by comparing:

- The **Default** strategy (automatic full-tree check)
- The **OnPush** strategy (optimized with reference checking)

```
ng new cd-lab --routing=false --style=css
cd cd-lab
```

Setup

Step 1: Generate Components

```
ng generate component default-cd
ng generate component onpush-cd
ng generate component parent
```

PART 1: Default Change Detection (Automatic)

Step 1.1: Create a `user` object in the parent

parent.component.ts

```
export class ParentComponent {
  user = { name: 'Ali' };

  mutateUser() {
    this.user.name = 'Ahmad';
  }

  replaceUser() {
    this.user = { name: 'Ahmad' };
  }
}
```

Step 1.2: Pass user to `DefaultCDComponent`

`parent.component.html`

```
<h3>Default Change Detection</h3>

<app-default-cd [user]="user"></app-default-cd>
<button (click)="mutateUser()">Mutate Name</button>
<button (click)="replaceUser()">Replace Object</button>
```

Step 1.3: Display user in `DefaultCDComponent`

`default-cd.component.ts`

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

@Component({
 selector: 'app-default-cd',
 imports: [],
 templateUrl: './default-cd.component.html',
 styleUrl: './default-cd.component.css'
})
export class DefaultCdComponent {

 @Input() user: any;

 ngDoCheck(){
   console.log("Default Change Detection Check");
   // This method is called on every change detection cycle
   // It can be used to manually check for changes if needed
 }
}
```

`default-cd.component.html`

```
<p>User: {{ user.name }}</p>
```

PART 2: OnPush Change Detection (Optimized)

Step 2.1: Use `ChangeDetectionStrategy.OnPush`

`onpush-cd.component.ts`

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

@Component({
  selector: 'app-onpush-cd',
  templateUrl: './onpush-cd.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class OnpushCdComponent {
  @Input() user: any;

  ngDoCheck() {
    console.log('OnPushCDComponent checked');
  }
}
```

Step 2.2: Add OnPush component to parent

`parent.component.html` (append this below previous section)

```
<h3>OnPush Change Detection</h3>
<app-onpush-cd [user]="user"></app-onpush-cd>
```

LAB TESTING & OBSERVATION

Test 1: Mutate the user object (change `.name`)

- Click **Mutate Name**
- `DefaultCDComponent` updates
- `OnPushCDComponent` **does NOT update**

Test 2: Replace the user object (new reference)

- Click **Replace Object**
- `DefaultCDComponent` updates
- `OnPushCDComponent` updates

Console Output:

Watch `ngDoCheck()` logs to see which component gets re-checked.

Bonus: Force OnPush Check Manually

Step 3.1: Inject and use `ChangeDetectorRef`

`onpush-cd.component.ts`

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

constructor(private cdr: ChangeDetectorRef) {}

forceCheck() {
  this.cdr.markForCheck();
}
```

`onpush-cd.component.html`

```
<p>User: {{ user.name }}</p>
<button (click)="forceCheck()">Force Detect</button>
```

Now try mutating the object **and then click "Force Detect"** — it will refresh.

Summary

Action	Default Strategy	OnPush Strategy
Mutate object property	Updates	No update
Replace whole object	Updates	Updates
Manual trigger	Not needed	Possible

Source code:

https://github.com/wanmuz86/angular-int-adv-lab3-cd

Angular Structural Directives & Control Flow

What Are Structural Directives?

Structural directives are Angular directives that **modify the structure of the DOM** — they **add, remove, or manipulate elements** based on conditions.

They are prefixed with `*` and applied directly to HTML elements.

Common Structural Directives

1. `*ngIf` – Conditional Rendering

Renders an element **only if the expression is true**.

```
<p *ngIf="isLoggedIn">Welcome back!</p>
<p *ngIf="!isLoggedIn">Please log in.</p>
```

With `else`:

```
<ng-template #noAccess><p>Access Denied</p></ng-template>
<p *ngIf="hasAccess; else noAccess">You have access</p>
```

2. `*ngFor` – Iterating Over a List

Repeats an element **for each item in a collection**.

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

With index:

```
<li *ngFor="let item of items; let i = index">
  {{ i + 1 }}. {{ item }}
</li>
```

3. `*ngSwitch` – Multiple Conditions

Renders one of many elements depending on a **matching case**.

```
<div [ngSwitch]="role">
  <p *ngSwitchCase="'admin'">Admin Panel</p>
  <p *ngSwitchCase="'user'">User Dashboard</p>
  <p *ngSwitchDefault>Guest View</p>
</div>
```

How Structural Directives Work

- Angular transforms structural directives into `<ng-template>` behind the scenes.
- Example:

```
<p *ngIf="show">Hello</p>
```

is transformed into:

```
<ng-template [ngIf]="show">
  <p>Hello</p>
</ng-template>
```

Best Practices

- Use `trackBy` with `*ngFor` for better performance:

```
<li *ngFor="let item of items; trackBy: trackById">{{ item.name }}</li>
```

```
trackById(index: number, item: any): number {
  return item.id;
}
```

- Avoid deeply nested structural directives (e.g., `*ngIf` inside `*ngFor`); consider using computed arrays in the component instead.

Summary Table

Directive	Purpose	Example Syntax
*ngIf	Conditionally show/hide element	<div *ngIf="show">Hello</div>
*ngFor	Repeat element for each item	<li *ngFor="let i of items">{{ i }}
*ngSwitch	Show one of many views	<div *ngSwitchCase="'value'">Content</div>

Angular Custom Directives

https://angular.dev/api/core/HostListener
https://angular.dev/guide/directives/attribute-directives
https://www.w3schools.com/jsref/dom_obj_event.asp [list of event]

What is a Directive?

A directive is a **class with Angular-specific behavior** that you can apply to elements in the DOM to **add custom logic or modify appearance/behavior**.

Angular has three types:

1. **Component** – a directive with a template
2. **Structural Directive** – changes DOM layout (e.g., `*ngIf`)
3. **Attribute Directive** – changes **appearance or behavior** of an element

Custom Attribute Directive

Use Case:

Add behavior like hover effects, color changes, tooltips, validation, etc.

Example: Highlight Directive

Step 1: Generate a directive

```
ng generate directive highlight
```

highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]' // Usage: <div appHighlight></div>
})
export class HighlightDirective {
  @Input() appHighlight = 'yellow';
```

```
  constructor(private el: ElementRef) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.appHighlight);
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight('');
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Usage in Template:

```
<p appHighlight="lightblue">Hover me to see highlight!</p>
```

Key Concepts:

Concept	**Purpose**
@Directive	Declares a class as an Angular directive
selector	Defines the name used in HTML ([appName])
ElementRef	Gives access to the DOM element
@Input()	Accepts data from the host element ([appHighlight]="'blue'")
@HostListener()	Subscribes to events on the host element (e.g., mouseenter, click)

Use Cases for Custom Directives

- **UI Behavior**: Hover effects, expand/collapse, autofocus
- **Validation**: Custom form validators
- **Permission Control**: Show/hide content based on roles

- **Reusable UI Logic**: Animate, style, or manage state

Summary

Term	Description
@Directive	Declares a custom directive class
ElementRef	Access the native DOM element
@Input()	Receive input from the template
@HostListener()	React to DOM events like hover or click
appHighlight	Custom directive name (used in the template)

Lab: Creating a Custom Directive in Angular

Objective:

Learn how to create and use a **custom attribute directive** that changes the background color of an element when hovered.

```
ng new custom-directive-lab
cd custom-directive-lab
ng serve
```

Setup: Generate a Directive

Step 1: Generate a new directive

```
ng generate directive highlight
```

This creates:

- `src/app/highlight.directive.ts`

PART 1: Build a Simple Highlight Directive

Step 2: Open `highlight.directive.ts` and replace the content with:

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

@Directive({
 selector: '[appHighlight]'
})
export class HighlightDirective {

 private defaultColor = 'yellow';
 private highlightColor: string = this.defaultColor;

 constructor(private el: ElementRef) {}

 @Input()
 set appHighlight(color: string) {
   this.highlightColor = color || this.defaultColor;
 }
```

```
 @HostListener('mouseenter') onMouseEnter() {
   this.highlight(this.highlightColor);
 }

 @HostListener('mouseleave') onMouseLeave() {
   this.highlight('');
 }

 private highlight(color: string) {
   this.el.nativeElement.style.backgroundColor = color;
 }
}
```

PART 2: Use the Directive in a Component

Step 3: Modify `app.component.html`

Replace the default content with:

```
<h2>Custom Directive Example</h2>

<p appHighlight="lightgreen">Hover over this text to see the highlight!</p>

<p appHighlight="lightcoral">Hover here for a different color.</p>

<p appHighlight>Hover here (uses default yellow)</p>
```

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { HighlightDirective } from './highlight.directive';

@Component({
 selector: 'app-root',
 imports: [HighlightDirective],
 templateUrl: './app.component.html',
 styleUrl: './app.component.css'
})
export class AppComponent {
 title = 'lab-4-custom-directive';
}
```

Step 4: Serve the App

```
ng serve
```

Test the behavior:

- Hovering over each `<p>` tag should change the background color.
- Mouse leave should restore the original background.

Bonus Exercise: Add Click-to-Toggle Highlight

Step 5: Extend the directive

Add this below `@HostListener('mouseleave')`:

```
@HostListener('click') onClick() {
  this.el.nativeElement.style.backgroundColor =
 this.el.nativeElement.style.backgroundColor ? '' : this.highlightColor;
 }
```

Observation:

- Clicking toggles the highlight on/off
- You now have a directive that responds to **hover and click**

Lab Summary

Feature	Learned
@Directive()	Declare a custom directive
ElementRef	Access native DOM element
@Input()	Accept input value from HTML

@HostListener()	Listen to events like hover

Source code reference:

https://github.com/wanmuz86/angular-int-adv-lab4-customdirective

Angular Standalone Components vs NgModules

https://angular.dev/guide/components

What Are Standalone Components?

Standalone components were introduced in **Angular 14** and promoted as a first-class pattern in **Angular 16+** to simplify component organization and reduce reliance on NgModules.

> **Standalone components** declare their own dependencies (imports), and do **not** need to be declared in a module.

Traditional NgModule-Based Architecture

```
@NgModule({
  declarations: [UserComponent],
  imports: [CommonModule],
  exports: [UserComponent]
})
export class UserModule {}
```

Components must be declared in a module and can only use dependencies from `imports`.

Standalone Component Approach

```
@Component({
  selector: 'app-user',
  standalone: true,
  imports: [CommonModule],
  template: `<p>Hello User!</p>`
})
export class UserComponent {}
```

Everything needed is self-contained. The component is immediately usable and testable.

Key Differences

Feature	NgModule-Based Approach	Standalone Component Approach
Declaration required	Yes, inside `@NgModule.declarations`	No — declared as `standalone: true`
Module needed to use it	Yes	No (directly importable)
Reusability	Scoped by module	Globally importable
Dependency management	Managed via module `imports`	Managed via `@Component.imports`
Lazy loading support	Yes	Yes (`loadComponent`)
Tree-shakability	Moderate (full module imported)	Higher (only what's used is imported)
Learning curve	Steeper (module mental overhead)	Simpler (especially for small apps)
Preferred in Angular 17+	Still valid	Highly recommended

When to Use Standalone Components

- For **feature encapsulation**
- In **small/medium apps**
- When building **highly reusable components**
- When simplifying **testability and portability**
- For **lazy-loaded components** via `loadComponent()`

When NgModules Are Still Useful

- When grouping multiple **non-standalone components** (legacy)
- For **shared** and **core** architectural modules
- For large teams who prefer **logical groupings**
- For backwards compatibility with libraries or tooling

Migration Path (Angular 15+)

- Convert components to `standalone: true`
- Replace `AppModule` with `bootstrapApplication(AppComponent)`
- Use `importProvidersFrom()` in `main.ts` or `app.config.ts`
- Group reusable components in `SharedModule`, even for standalone

Example: Bootstrapping Without NgModules (Angular 16+)

```
bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(HttpClientModule, CoreModule),
    provideRouter(routes)
  ]
});
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `<router-outlet />`
})
export class AppComponent {}
```

Summary

Topic	Standalone Components	NgModules
Syntax	`standalone: true` in component	`@NgModule()`
Registration	Import component directly	Declare in module
Bootstrapping	`bootstrapApplication()`	`@NgModule.bootstrap`
Scope	Self-contained	Scoped to module
Angular 17+ Style	Preferred	Still supported (less preferred)

Lab: Angular Standalone Components vs NgModules

Objective

Learn to build and use Angular components using:

- Traditional **NgModule-based** declaration
- Modern **Standalone Component** architecture

Compare their structure, dependency imports, and usage in routing or bootstrapping.

Step 0: Create a New Angular Project

```
ng new standalone-vs-module-lab --routing
cd standalone-vs-module-lab
```

- Select **"Standalone API"** when prompted

Part A: Create a Component Using NgModule

1. Generate a `legacy` module and component:

```
ng generate module legacy
ng generate component legacy/user-panel
```

2. Update `legacy/user-panel.component.ts`:

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

@Component({
  selector: 'app-user-panel',
  template: `<p>User Panel (NgModule-based)</p>`,
  standalone: false
})

export class UserPanelComponent {}
```

3. Update [`legacy.module.ts`](legacy.module.ts):

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { UserPanelComponent } from './user-panel/user-panel.component';

@NgModule({
  declarations: [UserPanelComponent],
  imports: [CommonModule],
  exports: [UserPanelComponent]
})
export class LegacyModule {}
```

Part B: Create a Standalone Component

```
ng generate component modern/user-card --standalone
```

1. Update user-card.component.ts:

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

@Component({
  selector: 'app-user-card',
  standalone: true,
  template: `<p>User Card (Standalone)</p>`
})
export class UserCardComponent {}
```

Step C: Define Routes to Compare Both

1. Update `app.routes.ts`:

```
import { Routes } from '@angular/router';
import { UserCardComponent } from './modern/user-card/user-card.component';
import { UserPanelComponent } from './legacy/user-panel/user-panel.component';

export const routes: Routes = [
  { path: 'user-panel', component: UserPanelComponent },
  { path: 'user-card', component: UserCardComponent }
];
```

Step D: Bootstrap Application with Standalone AppComponent

1. Update `app.config.ts`:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { provideRouter } from '@angular/router';
import { importProvidersFrom } from '@angular/core';
import { routes } from './app.routes';
import { LegacyModule } from './legacy/legacy.module';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    importProvidersFrom(LegacyModule) // only needed for NgModule component
  ]
});
```

2. Ensure `AppComponent` is standalone:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, RouterLink],
  template: `
    <h1>Standalone vs NgModule Demo</h1>
    <a routerLink="/user-card">Standalone User Card</a> |
    <a routerLink="/user-panel">NgModule User Panel</a>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

Step E: Run the App

```
ng serve
```

Visit:

- http://localhost:4200/user-card → shows **Standalone** component
- http://localhost:4200/user-panel → shows **NgModule-based** component

Lab Summary

Comparison	Standalone Component	NgModule Component
Declared in Module	Not needed	Required
Routing	Directly in `component:`	Requires module + provider
Imports	Defined in `@Component.imports`	Declared in NgModule
Bootstrapping	Simple with `bootstrapApplication`	Requires `importProvidersFrom()`
Angular 16+ Style	Preferred	Still supported

Smart vs Dumb Components (Container vs Presentational)

https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/

Separating your UI into **smart (container)** and **dumb (presentational)** components is a common architectural pattern that improves **scalability, testability, and reusability**.

Smart Components (Container Components)

Characteristics:

- Handle **business logic** and **data fetching**
- Know **how data is retrieved and changed**
- Pass data down to dumb components via `@Input()`
- Handle events from dumb components via `@Output()`
- Often connected to services (e.g., `HttpClient`, `Store`, etc.)

Example Use Case:

```
<app-user-detail [user]="selectedUser" (delete)="handleDelete($event)"></app-user-detail>
```

```
@Component({ ... })
export class UserPageComponent {
  selectedUser = this.userService.getUser();
  handleDelete(id: number) {
    this.userService.deleteUser(id).subscribe();
  }
}
```

Pros:

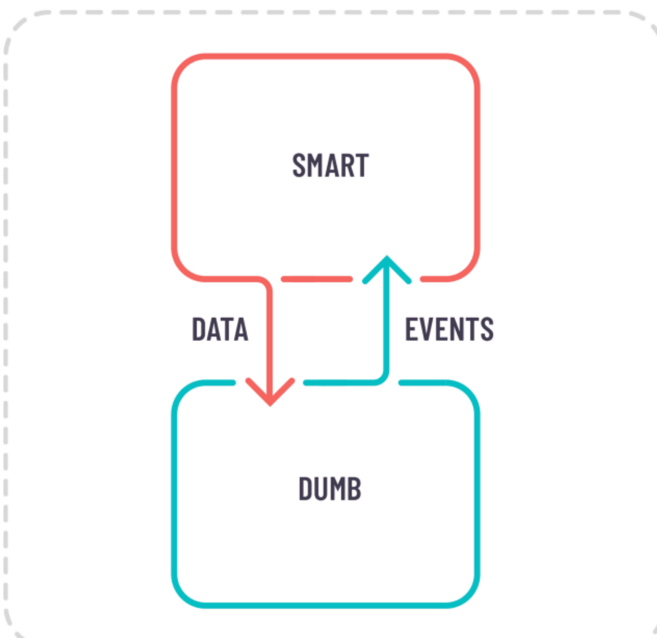
- Centralizes business logic
- Easier to manage workflows and state transitions

- Facilitates integration with services and external data

Dumb Components (Presentational Components)

Characteristics:

- Focus on **display only**
- Use `@Input()` to receive data
- Use `@Output()` to emit events (like button clicks)
- Have **no knowledge** of how data is fetched or stored
- Highly **reusable and testable**



Example Use Case:

```
@Component({ ... })
export class UserDetailComponent {
```

```
  @Input() user!: User;
  @Output() delete = new EventEmitter<number>();

  onDelete() {
    this.delete.emit(this.user.id);
  }
}
```

```
<div>
  <h2>{{ user.name }}</h2>
  <button (click)="onDelete()">Delete</button>
</div>
```

Pros:

- Simple and focused
- Easy to reuse across different contexts
- Easy to write unit tests for

Benefits of Separation

Benefit	Explanation
Reusability	Dumb components can be reused across pages
Testability	Easier to unit test presentational logic
Maintainability	Business logic is centralized in containers
Separation of Concerns	Clean split between logic and UI

Summary Table

Type	Smart Component	Dumb Component
Role	Handles data, logic	Displays UI
Uses Services?	Yes	No
Has Inputs?	Pass to dumb component	Accepts `@Input()`
Emits Events?	Listens to `@Output()`	Emits via `@Output()`
Reusable?	Specific to use case	Highly reusable

Best Practices

- Keep dumb components **stateless** and **service-free**
- Smart components should coordinate data, state, and side effects
- Use dumb components for forms, cards, buttons, dialogs, and other UI fragments
- This pattern scales well with feature modules and state management tools like NgRx or Signals

Lab: Smart vs Dumb Components in Angular

Objective:

Learn to separate logic and UI by building:

- A **Smart (container)** component that fetches and manages data
- A **Dumb (presentational)** component that displays and emits events

Create a new Angular project (if needed):

```
ng new smart-dumb-lab
cd smart-dumb-lab
```

Scenario:

You will build a simple **User List** app where:

- The **smart component** fetches user data and handles deletion
- The **dumb component** displays the list and emits delete events

Step 1: Generate Components

```
ng generate component user-list
ng generate component user-item
ng generate service user
```

Step 2: Create the User Service

`user.service.ts`

```
import { Injectable } from '@angular/core';
import { of } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class UserService {
  private users = [
```

```
    { id: 1, name: 'Ali' },
    { id: 2, name: 'Fatimah' },
    { id: 3, name: 'Zaid' },
  ];

  getUsers() {
    return of(this.users);
  }

  deleteUser(id: number) {
    this.users = this.users.filter(u => u.id !== id);
    return of(this.users);
  }
}
```

Step 3: Build the Dumb Component (`UserItemComponent`)

`user-item.component.ts`

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

@Component({
  selector: 'app-user-item',
  template: `
    <div class="user-card">
      {{ user.name }}
      <button (click)="deleteUser()">Delete</button>
    </div>
  `,
})
export class UserItemComponent {
  @Input() user: any;
  @Output() delete = new EventEmitter<number>();

  deleteUser() {
    this.delete.emit(this.user.id);
  }
}
```

Step 4: Build the Smart Component (`UserListComponent`)

user-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService } from '../user.service';
import { UserItemComponent } from '../user-item/user-item.component';
```

```
@Component({
 selector: 'app-user-list',
 imports: [UserItemComponent],
 templateUrl: './user-list.component.html',
 styleUrl: './user-list.component.css'
})
export class UserListComponent {
 users: any[] = [];

 constructor(private userService: UserService) {}

 ngOnInit() {
   this.loadUsers();
 }

 loadUsers() {
   this.userService.getUsers().subscribe(data => this.users = data);
 }

 handleDelete(id: number) {
   this.userService.deleteUser(id).subscribe(data => this.users = data);
 }
}
```

```
<h2>User List (Smart Component)</h2>
<app-user-item
 *ngFor="let user of users"
 [user]="user"
 (delete)="handleDelete($event)">
</app-user-item>
```

Step 5: Add `<app-user-list>` to `app.component.html`

```
<app-user-list></app-user-list>
```

Step 6: Run the App

```
ng serve
```

Test:

- You should see a list of users
- Clicking “Delete” on any user removes it from the list
- UI comes from the dumb component
- Logic is handled entirely by the smart component

Source code: https://github.com/wanmuz86/angular-int-adv-lab5-todo-smartdumb.git

Discussion

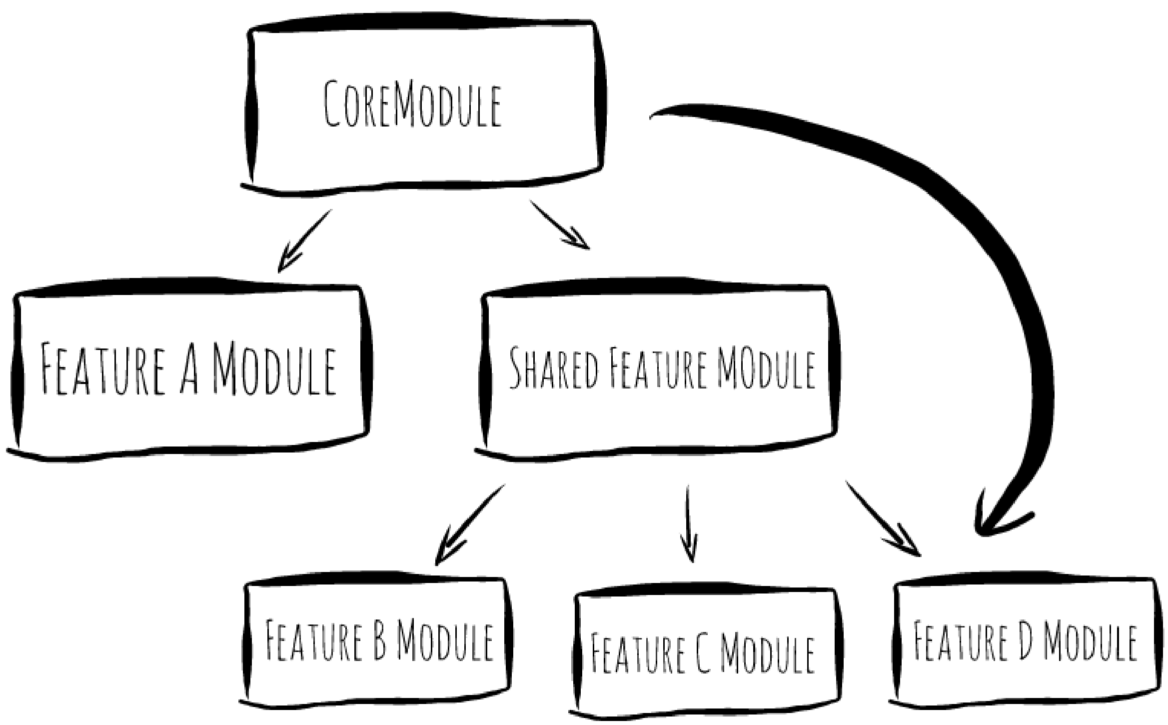
Aspect	UserListComponent	UserItemComponent
Type	Smart (Container)	Dumb (Presentational)
Uses Services?	Yes	No
Accepts `@Input()`	No	Yes (`user`)
Emits `@Output()`	Handles it	Emits `delete`
Reusable?	Specific use case	Reusable UI

Angular Modules: Feature, Shared, and Core

Angular applications are modular by nature. Even with the introduction of **standalone components**, NgModules still play an important role in organizing **cross-cutting concerns**, **shared logic**, and **feature encapsulation**.

Modular architecture improves:

- Reusability
- Maintainability
- Lazy loading & performance
- Team collaboration via feature boundaries



Feature Modules

What is a Feature Module?

A **Feature Module** contains code related to a specific domain, UI section, or functionality (e.g., `UserModule`, `ProductModule`). It may include:

- Standalone or non-standalone components
- Feature-specific services
- Routes and sub-routes

Key Characteristics

- Focused on **one domain** or **section** of the app
- Typically includes its own **routing module**
- Can be **eagerly loaded** or **lazy-loaded**
- Helps with **separation of concerns**

Example:

```
ng generate module features/products --routing
ng generate component features/products/product-list --standalone
```

products.module.ts

```
// features/products/products.module.ts
@NgModule({
  imports: [
    CommonModule,
    SharedModule,
    ProductsRoutingModule
  ]
})
export class ProductsModule {}
```

Shared Module

What is a Shared Module?

A **Shared Module** holds reusable **UI elements** like:

- Standalone components
- Pipes
- Directives

Used across multiple Feature Modules or Standalone Components.

Does not include:

- Singleton services
- Global providers (put those in CoreModule/app.config.ts)

Key Characteristics

- Re-export commonly used standalone components/pipes
- Helps avoid duplication
- Avoid importing it into `CoreModule` or app-wide bootstrap config to prevent overuse

Example:

```
ng generate module shared
ng generate component shared/custom-button --standalone
ng generate pipe shared/capitalize --standalone
ng generate directive shared/highlight --standalone
```

shared.module.ts

```
// shared/shared.module.ts
@NgModule({
  imports: [
    CommonModule,
    CustomButtonComponent,
    CapitalizePipe,
```

```
HighlightDirective
  ],
  exports: [
    CommonModule,
    CustomButtonComponent,
    CapitalizePipe,
    HighlightDirective
  ]
})
export class SharedModule {}
```

You don't **declare** standalone components — just **import and export** them.

Core Module

What is a Core Module?

The **Core Module** contains services and configurations that should **exist only once** in the app — global logic.

Includes:

- Singleton services (e.g., `AuthService`, `LoggerService`)
- HTTP interceptors
- Route guards
- App-wide providers

Never import CoreModule in Feature Modules or SharedModule.

Example:

```
ng generate module core
ng generate service core/auth
```

core.module.ts

```
// core/core.module.ts
import { NgModule, Optional, SkipSelf } from '@angular/core';
import { AuthService } from './auth.service';
```

```
@NgModule({
  providers: [AuthService]
})
export class CoreModule {
  constructor(@Optional() @SkipSelf() parentModule: CoreModule) {
    if (parentModule) {
      throw new Error('CoreModule is already loaded. Import only once in bootstrap configuration.');
    }
  }
}
```

How They Work Together

Module	Contains	Used In	Purpose
main.ts + app.config.ts	Bootstrap logic, routes, providers	App startup	Bootstraps and wires everything
CoreModule	Singleton services, interceptors, guards	Imported once in **importProvidersFrom(...)**	Global services
SharedModule	Standalone UI elements (pipes/components/directives)	Imported in Feature Modules or standalone components	Reusable logic
FeatureModule	Domain-specific routing, UI, and logic	**Lazy-loaded** or eagerly imported	Feature encapsulation

Best Practices for Angular 16+ (Standalone API)

CoreModule

- Use for singleton services (e.g., Auth, API Config, Logging)
- Import only once using `importProvidersFrom(CoreModule)` in `main.ts` or `app.config.ts`

SharedModule

- Export **standalone** components/pipes/directives only
- Do **not** add services here
- Use only where needed (not in app-wide config)

Feature Modules

- Use per-domain or per-workflow (e.g., AdminModule, ProductModule)
- Can include standalone components or traditional components with declarations
- Consider lazy-loading with `loadChildren`

General Rules

- Don't **re-import CoreModule**
- Don't **duplicate component declarations**
- Don't **overload SharedModule** — keep it focused on UI building blocks
- Prefer **standalone components** for UI pieces in Angular 16+

Lab: Structuring an Angular App with Core, Shared, and Feature Modules

Objective:

Learn to organize a scalable Angular application using:

- **Feature Modules** (e.g., Product)
- **Shared Module** for reusable UI elements
- **Core Module** for singleton services and application-wide providers

Step 0: Setup a New Angular Project

```
ng new module-lab --routing
cd module-lab
```

Choose **Standalone API** when prompted.

Step 1: Create Shared and Core Modules

```
ng generate module shared
ng generate module core
```

Step 2: Add a Shared Component & Pipe

Create a shared component , pipe & directive

```
ng generate component shared/custom-button
ng generate pipe shared/capitalize
ng generate directive shared/highlight
```

Update `shared.module.ts`

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CustomButtonComponent } from './custom-button/custom-button.component';
```

```
import { CapitalizePipe } from './capitalize.pipe';
import { HighlightDirective } from './highlight.directive';




@NgModule({
 imports: [
   CommonModule,
   CustomButtonComponent,
   CapitalizePipe,
HighlightDirective

 ],
 exports: [CustomButtonComponent, CapitalizePipe, CommonModule, HighlightDirective]
})
export class SharedModule { }
```

- Do not declare standalone components or pipes. Import them instead.

Update the CapitalizePipe as follows:

https://angular.dev/tutorials/learn-angular/24-create-a-pipe

https://angular.dev/guide/templates/pipes#creating-custom-pipes

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

@Pipe({
 name: 'capitalize',
 standalone: true //  include this if you're using standalone components
})
export class CapitalizePipe implements PipeTransform {

 transform(value: unknown): string {
   if (typeof value !== 'string') return '';

   return value.charAt(0).toUpperCase() + value.slice(1);
 }

}
```

Update the custom-button.component.ts to be as follows:

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

@Component({
  selector: 'app-custom-button',
  standalone: true,
  template: `
    <button [type]="type" class="custom-button" (click)="handleClick()">
      {{ label }}
    </button>
  `,
  styles: [`
    .custom-button {
      padding: 0.5rem 1rem;
      font-size: 1rem;
      background-color: #1976d2;
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
      transition: background-color 0.3s ease;
    }

    .custom-button:hover {
      background-color: #125aa0;
    }
  `]
})
export class CustomButtonComponent {
  @Input() label = 'Click';
  @Input() type: 'button' | 'submit' = 'button';

  handleClick() {
    console.log(`Custom button clicked: ${this.label}`);
  }
}
```

Step 3: Create a Core Service

Generate AuthService

```
ng generate service core/auth
```

Add the code to simulate log in

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

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

 private isLoggedIn = false;

 login(username: string, password: string): boolean {
 if (username === 'admin' && password === 'password') {
   this.isLoggedIn = true;
   return true;
 }
 return false;
 }

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

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

Update `core.module.ts`

```
import { NgModule, Optional, SkipSelf } from '@angular/core';
import { AuthService } from './auth.service';

@NgModule({
  providers: [AuthService]
})
export class CoreModule {
  constructor(@Optional() @SkipSelf() parent: CoreModule) {
    if (parent) {
      throw new Error('CoreModule should only be imported in AppModule!');
    }
  }
}
```

A **provider** tells Angular **how to create or deliver a dependency** (usually a service).

`AuthService` will now be injected as a singleton across the app.

Step 4: Create a Feature Module (e.g., Product, Orders)

```
ng generate module features/product --routing
ng generate component features/product/product-list --standalone
ng generate component features/product/product-detail

ng generate module features/orders --routing
ng generate component features/orders/order-list
```

Update `product.module.ts`

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductRoutingModule } from './product-routing.module';
import { SharedModule } from '../../shared/shared.module';

@NgModule({
 declarations: [],
 imports: [
   CommonModule,
   ProductRoutingModule,
   SharedModule
 ]
})
export class ProductModule { }
```

Step 5: Use Shared Component & Pipe in Feature Component

Update `product-list.component.ts`

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CustomButtonComponent } from
'../../../shared/custom-button/custom-button.component';
import { CapitalizePipe } from '../../../shared/capitalize.pipe';

@Component({
 selector: 'app-product-list',
 standalone: true,
```

```
 imports: [CommonModule, CustomButtonComponent, CapitalizePipe],
 templateUrl: './product-list.component.html'
})
export class ProductListComponent {
 products = ['apple', 'banana', 'orange'];
}
```

Update `product-list.component.html`

```
<h2>Product List</h2>
<app-custom-button label="Click me!"></app-custom-button>

<ul>
  <li *ngFor="let p of products">{{ p | capitalize }}</li>
</ul>
```

Step 6: Bootstrap the App in main.ts

Create app.routes.ts:

```
import { Routes } from '@angular/router';
import { OrderListComponent } from './features/orders/order-list/order-list.component';

export const routes: Routes = [

   // Example of lazy load
   // During this first load of the application, the module is not loaded
   // It will only be loaded when the user navigates to the 'products' path
   // When you build the application, the module will be bundled into a separate chunk (We )
   {
      path:'products',
      loadChildren:()=>
      import('./features/product/product.module').then(m=>m.ProductModule)
   },


   // Example of Eager Load
   // The url will be loadad immediately when the application starts
   // Impact on the load time of opening the application first time (first load)

   {
      path:'orders',
      component:OrderListComponent
   }
];
```

```
const routes: Routes = [
{
 path:'',
 component: ProductListComponent
},
// Lazy load component when the url matches ':id' / products/1, products/2, etc.
{
 path:':id',
 loadComponent: () => import('./product-detail/product-detail.component').then(m =>
m.ProductDetailComponent)
}
];
```

Update config.ts:

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

import { routes } from './app.routes';
import { importProvidersFrom } from '@angular/core';
import { CoreModule } from './core/core.module';
import { ProductModule } from './features/product/product.module';


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

Ensure AppComponent is standalone:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {}
```

Step 7: Run the App

```
ng serve
```

You Should See:

- A product list with names like Apple, Banana, Orange capitalized
- A custom button rendered from the Shared module

Lab Deliverables

Module / File	Purpose
SharedModule	Exports reusable UI components and pipes (standalone)
CoreModule	Provides singleton services (e.g., `AuthService`)
ProductModule	Encapsulates product-related UI & logic

app.routes.ts	Defines routes for the app
main.ts	Bootstraps the app and imports modules
AppComponent	Standalone root component with routing enabled

Code for this exercise:

https://github.com/wanmuz86/angular-int-adv-lab6-modular-structure

Run npm run build to see the chunk created as well

```
anmuz@wans-MacBook-Air browser % ls
hunk-GWOVFRVO.js        chunk-XYHG6AH6.js        main-TQTGDSLJ.js
hunk-LI6V7GCM.js        favicon.ico              polyfills-B6TNHZQ6.js
hunk-WP622WL3.js        index.html               styles-5INURTSO.css
```

Lab: Build a Smart/Dumb Component Structure with OnPush Change Detection

Objective

By the end of this lab, you will:

- Build a **smart (container)** component that manages data and logic
- Build a **dumb (presentational)** component that displays UI
- Use `@Input()` and `@Output()` for communication
- Use `ChangeDetectionStrategy.OnPush` in the dumb component
- Observe the performance behavior of OnPush vs Default change detection

Step 1: Create Angular Project

```
ng new smart-dumb-onpush-lab --routing=false --style=css
cd smart-dumb-onpush-lab
```

Step 2: Generate Components and Service

```
ng generate component user-list  --standalone # smart component
ng generate component user-item  --standalone  # dumb component
ng generate service user
```

Project Structure

```
src/
├── app/
│   ├── user-list/       # Smart
│   ├── user-item/       # Dumb (OnPush)
│   ├── user.service.ts  # Mock user service
│   └── app.component.ts
```

Step 3: Set Up User Service

`user.service.ts`

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

export interface User {
  id: number;
  name: string;
}

@Injectable({ providedIn: 'root' })
export class UserService {
  private users = new BehaviorSubject<User[]>([
    { id: 1, name: 'Ali' },
    { id: 2, name: 'Fatimah' },
    { id: 3, name: 'Zaid' }
  ]);

  getUsers(): Observable<User[]> {
    return this.users.asObservable();
  }

  updateUserName(id: number, newName: string) {
    const updated = this.users.value.map(u =>
      u.id === id ? { ...u, name: newName } : u
    );
    this.users.next(updated);
  }
}
```

Step 4: Build Dumb Component (`UserItemComponent`)

`user-item.component.ts`

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

```
import { User } from '../user.service';

@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserItemComponent {
  @Input() user!: User;
  @Output() update = new EventEmitter<number>();

  updateUserName() {
    this.update.emit(this.user.id);
  }
}
```

user-item.component.html

```
<div class="card">
  <p><strong>{{ user.name }}</strong></p>
  <button (click)="updateUserName()">Change Name</button>
</div>
```

Step 5: Build Smart Component (`UserListComponent`)

user-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService, User } from '../user.service';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html'
})
export class UserListComponent implements OnInit {
  users: User[] = [];

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.userService.getUsers().subscribe(data => {
      this.users = data;
    });
  }
```

```
  changeName(id: number) {
    const newName = prompt('Enter new name:');
    if (newName) {
      this.userService.updateUserName(id, newName);
    }
  }
}
```

user-list.component.html

```
<h2>User List (Smart Component)</h2>
<app-user-item
  *ngFor="let user of users"
  [user]="user"
  (update)="changeName($event)">
</app-user-item>
```

Step 6: Wire It in AppComponent

app.component.html

```
<app-user-list></app-user-list>
```

Step 7: Run the App

```
ng serve
```

Test & Observe Behavior

Expected Behavior

- The list of users is displayed using a **dumb (presentational) component**.
- Clicking **"Change Name"** updates only the relevant user in the list.
- With **OnPush change detection**, the view should update **only** when necessary — based on input reference changes.

OnPush Change Detection Test

Test In-Place Mutation (What Not to Do)

In `UserService`, try updating a user by **mutating the object directly**, like this:

```
this.users.value[0].name = 'Mutated Ali';
this.users.next(this.users.value); // emits the same array reference
```

Result:

If you're using `ChangeDetectionStrategy.OnPush` in the component that receives the user list (via `@Input()`), this change **may not be detected**.

Why?
Because:

- Angular's OnPush strategy only checks for **reference changes** on `@Input()` properties.
- Here, you are **mutating the object in place**, and the array reference (`this.users.value`) remains the same — so Angular won't detect any difference.

Use Immutable Update (What You Should Do)

Instead, update the user's name by creating a **new object** and emitting a **new array reference**, like this:

```
const updated = this.users.value.map(u =>
  u.id === id ? { ...u, name: newName } : u
);
this.users.next(updated);
```

Result:

- The array reference has changed.
- The updated user object is a **new object**.
- Angular's `OnPush` strategy **detects this change** and the view updates as expected.

Summary

Update Type	Reference Changed?	OnPush Detects?
In-place mutation	No	No
Immutable update	Yes	Yes

Key Learnings

Concept	Applied In
Smart Component	`UserListComponent`
Dumb Component	`UserItemComponent`
`@Input()` / `@Output()`	Data and event communication
`ChangeDetectionStrategy.OnPush`	Improves performance in dumb component
Immutable update	Required to trigger OnPush change detection

Bonus Challenge

- Add a “Reset All Names” button in the smart component
- Track rendering using `ngDoCheck()` in both components
- Show how OnPush avoids unnecessary re-renders

Efficient Change Detection with trackBy

https://angular.dev/api/core/TrackByFunction

What is trackBy?

In Angular, when using `*ngFor` to loop through a list, Angular by default tracks changes using **object identity**.

This means even if the data hasn't changed, but the **reference** to the array is new (e.g., after a `.slice()` or spread `[...]`), Angular re-renders all items.

Problem:

- Updating or refreshing a list (even with the same content) causes all items to be **re-created in the DOM**.
- This is **inefficient** for large lists or frequent updates.

Solution: Use `trackBy`

The `trackBy` function helps Angular identify **which items have actually changed**, using a **unique identifier** like `id`.

When a list iterated by *ngFor changes (items are added, removed, or reordered), Angular's default change detection mechanism re-renders all the DOM elements associated with the list if object references change. This can be inefficient, especially for large lists or frequently changing data.
The trackBy function provides a way to tell Angular how to uniquely identify each item in the list. Instead of relying on object identity, you can specify a unique key (e.g., an id property) for each item.

Syntax:

```
<li *ngFor="let item of items; trackBy: trackByFn">
  {{ item.name }}
</li>
```

In Component:

```
trackByFn(index: number, item: any): number {
```

```
  return item.id;
}
```

Without `trackBy`

Angular will:

- Compare by **object reference**
- Re-render all items if a new array is passed, even with same data
- Lose element state (e.g., input focus, animations, scroll position)

With `trackBy`

Angular will:

- Track by **unique value** (e.g., `item.id`)
- Re-render only modified/new items
- **Preserve** DOM elements that haven't changed
- Improve performance and memory usage

Example:

Inefficient:

```
users = [...this.users]; // re-renders all items
```

Efficient:

```
<li *ngFor="let user of users; trackBy: trackByUserId">{{ user.name }}</li>
```

```
trackByUserId(index: number, user: any): number {
  return user.id;
```

```
}
```

Benefits of `trackBy`

Benefit	Description
Optimized rendering	Only re-renders items that actually change
Performance boost	Useful in large lists or high-frequency updates
DOM stability	Preserves user input state, animations, scroll
Debug-friendly	Easier to track re-renders in console

Best Practices

- Always use `trackBy` when:
  - You render a **list of objects**
  - You update the array with new references (e.g., after `.map()`, `.filter()`)
- Use **unique IDs** as the tracking key
- Avoid using `index` as a fallback — only use when items are truly fixed

Lab: Efficient Change Detection with `trackBy` in Angular

Objective

Learn how Angular handles DOM rendering with `*ngFor` and how using the `trackBy` function improves performance by avoiding unnecessary re-renders of DOM elements.

Step 1: Create Angular Project

```
ng new trackby-lab --routing=false --style=css
cd trackby-lab
```

Step 2: Generate a Component

```
ng generate component users
```

Folder Structure

```
src/
└── app/
    ├── users/
    │   ├── users.component.ts
    │   ├── users.component.html
    │   └── users.component.css
    └── app.component.html
```

Step 3: Create Sample Data and Update Method

users.component.ts

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

@Component({
  selector: 'app-users',
  templateUrl: './users.component.html'
})
export class UsersComponent {
  users = [
    { id: 1, name: 'Ali' },
    { id: 2, name: 'Fatimah' },
    { id: 3, name: 'Zaid' }
  ];
```

```
  refreshList() {
    // simulate data refresh with same content but new object references
    this.users = [...this.users];
  }

  changeName(index: number) {
    const updated = [...this.users];
    updated[index].name += ' 🔁';
    this.users = updated;
  }

  // Optional trackBy function
  trackByUserId(index: number, user: any): number {
    return user.id;
  }
}
```

Step 4: Display Users with `*ngFor`

Without `trackBy` (First test)

`users.component.html`

```
<h2>User List</h2>
<ul>
  <li *ngFor="let user of users">
    {{ user.name }}
    <button (click)="changeName(users.indexOf(user))">Change Name</button>
  </li>
</ul>

<button (click)="refreshList()">Refresh (same data)</button>
```

Observe:

1. Open browser dev tools.
2. Add a `console.log()` in the template:

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

And in `users.component.ts`:

```
log(value: string) {
  console.log('Rendering:', value);
  return '';
}
```

3. Click **Refresh**.
   Even though the data is the same, Angular re-renders every item.

Step 5: Use `trackBy` for Optimization

Update `*ngFor` with `trackBy`

`users.component.html`

```
<li *ngFor="let user of users; trackBy: trackByUserId">
  {{ log(user.name) }} {{ user.name }}
</li>
```

Test Again

- Click the **Refresh** button.
- Now only changed items (if any) will be re-rendered.
- DOM elements are preserved; improves performance in large lists.

Explanation

Angular by default tracks items by **object identity**.
When `trackBy` is used, Angular uses the **unique `id`** to track which items have changed.

Summary

Test Case	Without `trackBy`	With `trackBy`
Refresh same data	All re-rendered	DOM reused

Update one item	Only updated rendered	Only updated rendered

Bonus Challenge

- Add 1,000 fake users using `Array.from()` and test performance

```
users = Array.from({ length: 1000 }, (_, i) => ({
   id: i + 1,
   name: `User ${i + 1}`,
   renderCount: 0
 }));
```

- Open your browser's **DevTools (F12)**
- Go to the **Performance** tab
- Click the **Record** button
- Click the **Refresh List** button in the app
- Stop recording after rendering completes
- Analyze:
    - Frame rate
    - DOM updates
    - Scripting time

Then do the same **without `trackBy`** and compare.

DAY 2

Advanced Reactive Forms in Angular

https://angular.dev/api/forms/FormBuilder

https://angular.dev/api/forms/FormArray

https://angular.dev/guide/forms/reactive-forms

Reactive Forms offer a **model-driven**, scalable approach to handling form inputs, validation, and dynamic controls.

In Angular 16+, you can also use them seamlessly in **standalone components**.

1. `FormBuilder` – Simplified Form Creation

What it is:

A service that helps you **create form controls and groups** with less boilerplate.

Syntax:

```
import { FormBuilder, FormGroup } from '@angular/forms';

constructor(private fb: FormBuilder) {}

form: FormGroup = this.fb.group({
  name: [''],
  email: [''],
});
```

Advantages:

- Cleaner syntax
- Useful when building **complex or dynamic forms**
- Can include validators directly

Example with Validators:

https://angular.dev/api/forms/Validators

```
this.fb.group({
  name: ['', [Validators.required]],
  email: ['', [Validators.required, Validators.email]],
});
```

2. `FormArray` – Managing Dynamic Fields

What it is:

`FormArray` allows you to **dynamically add or remove groups of form controls** — ideal for repeating fields (e.g., list of skills, addresses, phone numbers).

Usage:

```
form = this.fb.group({
  users: this.fb.array([]) // <-- FormArray
});

get users(): FormArray {
  return this.form.get('users') as FormArray;
}

addUser() {
  this.users.push(this.fb.group({
    name: ['', Validators.required],
    age: ['']
  }));
}
```

Template:

```
<div formArrayName="users">
  <div *ngFor="let user of users.controls; let i = index" [formGroupName]="i">
    <input formControlName="name">
    <input formControlName="age">
    <button (click)="removeUser(i)">Remove</button>
  </div>
</div>
```

3. Async Validators – Server-Side or Delayed Checks

What it is:

Async validators are used to **validate form values against asynchronous operations**, like checking username/email availability via HTTP.

Syntax:

```
this.fb.group({
  username: ['', {
    validators: [Validators.required],
    asyncValidators: [this.checkUsernameAvailability()],
    updateOn: 'blur' // trigger async check only on blur
  }]
});
```

Example Validator:

```
checkUsernameAvailability(): AsyncValidatorFn {
  return (control: AbstractControl): Observable<ValidationErrors | null> => {
    return this.http.get(`/api/users?username=${control.value}`).pipe(
      map(user => user ? { usernameTaken: true } : null),
      catchError(() => of(null))
    );
  };
}
```

Template:

```
<input formControlName="username">
<div *ngIf="form.get('username')?.errors?.['usernameTaken']">
  Username is already taken.
</div>
```

4. Control Value & Status Access

```
this.form.get('username')?.value;      // Read current value
this.form.get('username')?.status;    // VALID / INVALID / PENDING
this.form.get('username')?.pending;   // true or false
```

5. Reset and Disable Forms

```
this.form.reset();      // Clear values and states
this.form.disable();    // Disable entire form
this.form.enable();     // Enable again
```

6. UX: Conditional Styling

```
<input formControlName="email" [class.invalid]="form.get('email')?.invalid &&
form.get('email')?.touched">
```

```
.invalid {
  border: 1px solid red;
}
```

Summary Table

Feature	Purpose	Key APIs / Usage
FormBuilder	Create form groups & controls easily	`fb.group()`, `fb.control()`
FormArray	Handle dynamic repeatable fields	`FormArray`, `.push()`, `.removeAt()`
Async Validator	Validate against async operations	`AsyncValidatorFn`, `updateOn: 'blur'`
updateOn	Control validation timing	`'change'`, `'blur'`, `'submit'`

Best Practices for Angular 16+

- Use `FormBuilder` for clean and readable setup
- Always cast dynamic arrays: `as FormArray`
- Use `updateOn: 'blur'` for async validations
- In standalone components, import **`ReactiveFormsModule`** directly
- Avoid using `ngModel` and reactive form APIs together
- Use `.pending` state in templates for async feedback

Lab: Advanced Reactive Forms in Angular

Objective

By the end of this lab, you will:

- Create a reactive form using `FormBuilder`
- Dynamically manage multiple input sections using `FormArray`
- Validate a field using a custom **asynchronous validator** simulating a server check

Step 1: Setup Project & Generate Component

```
ng new reactive-advanced-lab --routing=false --style=css
cd reactive-advanced-lab
ng generate component register-form
```

Update `app.component.html` to:

```
<app-register-form></app-register-form>
```

Step 2: Use ReactiveFormsModule in Standalone Component

Edit `src/app/register-form/register-form.component.ts`:

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder, FormGroup, Validators, FormArray,
  AsyncValidatorFn, AbstractControl, ValidationErrors, ReactiveFormsModule
} from '@angular/forms';
import { CommonModule } from '@angular/common';
import { Observable, of } from 'rxjs';
import { delay, map } from 'rxjs/operators';

@Component({
  selector: 'app-register-form',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
```

```
  templateUrl: './register-form.component.html'
})
export class RegisterFormComponent implements OnInit {
  registerForm!: FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit(): void {
    this.registerForm = this.fb.group({
      username: ['', {
        validators: [Validators.required],
        asyncValidators: [this.usernameTakenValidator()],
        updateOn: 'blur'
      }],
      emails: this.fb.array([
        this.fb.control('', [Validators.required, Validators.email])
      ])
    });
  }

  get emails(): FormArray {
    return this.registerForm.get('emails') as FormArray;
  }

  addEmail(): void {
    this.emails.push(this.fb.control('', [Validators.required, Validators.email]));
  }

  removeEmail(index: number): void {
    this.emails.removeAt(index);
  }

  usernameTakenValidator(): AsyncValidatorFn {
    const takenUsernames = ['admin', 'user', 'test'];
    return (control: AbstractControl): Observable<ValidationErrors | null> => {
      return of(takenUsernames.includes(control.value)).pipe(
        delay(1000),
        map(isTaken => isTaken ? { usernameTaken: true } : null)
      );
    };
  }

  onSubmit(): void {
    console.log(this.registerForm.value);
  }
}
```

Step 3: Template

```
<!-- src/app/register-form/register-form.component.html -->
<form [formGroup]="registerForm" (ngSubmit)="onSubmit()">

  <div>
    <label>Username:</label>
    <input formControlName="username">
    <div *ngIf="registerForm.get('username')?.pending">Checking availability...</div>
    <div *ngIf="registerForm.get('username')?.errors?.['usernameTaken']">
      Username is already taken.
    </div>
  </div>

  <div formArrayName="emails">
    <label>Emails:</label>
    <div *ngFor="let emailCtrl of emails.controls; let i = index">
      <input [formControlName]="i">
      <button type="button" (click)="removeEmail(i)" *ngIf="emails.length >
1">Remove</button>
    </div>
    <button type="button" (click)="addEmail()">Add Email</button>
  </div>

  <br>
  <button type="submit" [disabled]="registerForm.invalid ||
registerForm.pending">Register</button>
</form>
```

Step 4: Test the App

Run the app:

```
ng serve
```

Try:

- Typing a valid username: form allows submission
- Typing "admin" or "test": shows async validation error
- Adding/removing multiple email inputs
- Submitting form and viewing logged data

Summary of Concepts Practiced

Feature	Purpose
`FormBuilder`	Simplifies form creation
`FormArray`	Allows dynamic input fields (e.g., multiple emails)
Async Validator	Checks username availability asynchronously
`updateOn`	Optimizes when async validator is triggered

Bonus Challenge

- Add a **password + confirm password** field with custom synchronous validator
- Mark invalid fields in red using conditional CSS classes
- Add submit confirmation below the form

Source code for this lab:

https://github.com/wanmuz86/lab-7-reactive-form

Lab: Create a Form with Nested `FormGroup` and Custom Validators

Objective

By the end of this lab, you will:

- Create a reactive form with **nested `FormGroup`**
- Build and use **custom synchronous validators**
- Display error messages based on form validation state

Step 1: Create a New Angular Project and Component

```
ng new nested-form-lab --routing=false --style=css
cd nested-form-lab
ng generate component registration-form
```

Update `app.component.html` to:

```
<app-registration-form></app-registration-form>
```

Step 2: Enable Reactive Forms

`Update app.config.ts:`

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { importProvidersFrom } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';

export const appConfig: ApplicationConfig = {
 providers: [provideZoneChangeDetection({ eventCoalescing: true }),
   [importProvidersFrom(ReactiveFormsModule)]
};
```

Step 3: Create the Nested FormGroup

In `registration-form.component.ts`:

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder, FormGroup, Validators,
  AbstractControl, ValidationErrors,
  ReactiveFormsModule
} from '@angular/forms';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-registration-form',
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule],
  templateUrl: './registration-form.component.html',
  styleUrls: ['./registration-form.component.css']
})
export class RegistrationFormComponent implements OnInit {
  registerForm!: FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit(): void {
    this.registerForm = this.fb.group({
      personal: this.fb.group({
        firstName: ['', Validators.required],
        lastName: ['', Validators.required]
      }),
      account: this.fb.group({
        username: ['', [Validators.required, this.noAdminValidator]],
        password: ['', [Validators.required, Validators.minLength(6)]],
        confirmPassword: ['']
      }, { validators: [this.passwordMatchValidator] })
    });
  }

  noAdminValidator(control: AbstractControl): ValidationErrors | null {
    const forbidden = control.value?.toLowerCase() === 'admin';
    return forbidden ? { forbiddenName: true } : null;
  }

  passwordMatchValidator(group: AbstractControl): ValidationErrors | null {
    const password = group.get('password')?.value;
    const confirmPassword = group.get('confirmPassword')?.value;
    return password === confirmPassword ? null : { passwordsNotMatch: true };
```

```
  }

  onSubmit(): void {
    console.log(this.registerForm.value);
  }
}
```

Step 4: Create the Template

registration-form.component.html

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

  <fieldset formGroupName="personal">
    <legend>Personal Info</legend>

    <label>First Name</label>
    <input formControlName="firstName">
    <div *ngIf="registerForm.get('personal.firstName')?.invalid &&
registerForm.get('personal.firstName')?.touched">
      First name is required.
    </div>

    <label>Last Name</label>
    <input formControlName="lastName">
    <div *ngIf="registerForm.get('personal.lastName')?.invalid &&
registerForm.get('personal.lastName')?.touched">
      Last name is required.
    </div>
  </fieldset>

  <fieldset formGroupName="account">
    <legend>Account Info</legend>

    <label>Username</label>
    <input formControlName="username">
    <div *ngIf="registerForm.get('account.username')?.hasError('forbiddenName')">
      "admin" is not allowed as a username.
    </div>

    <label>Password</label>
    <input type="password" formControlName="password">
    <div *ngIf="registerForm.get('account.password')?.invalid &&
registerForm.get('account.password')?.touched">
```

```
    Password must be at least 6 characters.
  </div>

  <label>Confirm Password</label>
  <input type="password" formControlName="confirmPassword">
  <div *ngIf="registerForm.get('account')?.errors?.['passwordsNotMatch']">
    Passwords do not match.
  </div>
</fieldset>

<button type="submit" [disabled]="registerForm.invalid">Register</button>
</form>
```

Step 5: Run and Test

```
ng serve
```

Try:

- Leaving required fields empty
- Typing `"admin"` as username → shows custom validation error
- Typing mismatched passwords → shows `passwordsNotMatch` error
- Submitting the form prints form data in the console

Summary

Feature	Where Used
Nested `FormGroup`	`personal` and `account` groups
Synchronous Validators	`Validators.required`, `minLength`

Custom Field Validator	`noAdminValidator` (username)
Custom Group Validator	`passwordMatchValidator`

Bonus Challenge

- Add an **email** field with built-in email validator
- Add a **phone number group** with `countryCode` and `number`
- Convert to use **FormBuilder arrays** for multiple addresses

Answer for challenge exercise

registration-form.component.ts

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  Validators,
  AbstractControl,
  ValidationErrors,
  FormArray
} from '@angular/forms';

@Component({
  selector: 'app-registration-form',
  templateUrl: './registration-form.component.html'
})
export class RegistrationFormComponent implements OnInit {
  registerForm!: FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit(): void {
    this.registerForm = this.fb.group({
      personal: this.fb.group({
        firstName: ['', [Validators.required]],
        lastName: ['', [Validators.required]],
        email: ['', [Validators.required, Validators.email]] // ✅ Email field
      }),
      account: this.fb.group(
        {
          username: ['', [Validators.required, this.noAdminValidator]],
          password: ['', [Validators.required, Validators.minLength(6)]],
          confirmPassword: ['']
        },
        { validators: [this.passwordMatchValidator] }
      ),
      phone: this.fb.group({ //  Phone group
        countryCode: ['+60', Validators.required],
        number: ['', Validators.required]
      }),
      addresses: this.fb.array([this.createAddress()]) //  FormArray of addresses
    });
  }

  // Helper to create address form group
  createAddress(): FormGroup {
```

```
    return this.fb.group({
      street: ['', Validators.required],
      city: ['', Validators.required],
      zip: ['']
    });
  }

  get addresses(): FormArray {
    return this.registerForm.get('addresses') as FormArray;
  }

  addAddress() {
    this.addresses.push(this.createAddress());
  }

  removeAddress(index: number) {
    this.addresses.removeAt(index);
  }

  noAdminValidator(control: AbstractControl): ValidationErrors | null {
    const forbidden = control.value?.toLowerCase() === 'admin';
    return forbidden ? { forbiddenName: true } : null;
  }

  passwordMatchValidator(group: AbstractControl): ValidationErrors | null {
    const password = group.get('password')?.value;
    const confirmPassword = group.get('confirmPassword')?.value;
    return password === confirmPassword ? null : { passwordsNotMatch: true };
  }

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

Template Additions: `registration-form.component.html`

Email Field

```
<label>Email</label>
<input formControlName="email" type="email" />
<div *ngIf="registerForm.get('personal.email')?.invalid &&
registerForm.get('personal.email')?.touched">
  Enter a valid email address.
</div>
```

Phone Group

```html
<fieldset formGroupName="phone">
  <legend>Phone Number</legend>
  <label>Country Code</label>
  <input formControlName="countryCode" />
  <label>Number</label>
  <input formControlName="number" />
</fieldset>
```

Addresses (FormArray)

```html
<fieldset formArrayName="addresses">
  <legend>Addresses</legend>
  <div *ngFor="let address of addresses.controls; let i = index" [formGroupName]="i">
    <label>Street</label>
    <input formControlName="street" />
    <label>City</label>
    <input formControlName="city" />
    <label>ZIP</label>
    <input formControlName="zip" />
    <button type="button" (click)="removeAddress(i)" *ngIf="addresses.length >
1">Remove</button>
    <hr />
  </div>
  <button type="button" (click)="addAddress()">Add Address</button>
</fieldset>
```

Summary of Bonus Features

Feature	Implementation
Email field	`Validators.email`
Phone group	Nested `FormGroup` (`countryCode`, `number`)
Multiple addresses	`FormArray` with dynamic add/remove

Angular Routing: Key Concepts

https://angular.dev/guide/routing

Routing in Angular allows navigation between different views or components based on the URL.

1. Basic Routing

Setup

In `app-routing.module.ts`:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
 { path: 'products', component: ProductComponent },
 { path: 'products/:id', component: ProductDetailComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: '**', component: NotFoundComponent }
];
```

Router Outlet

In template:

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

2. Lazy Loading Modules

https://angular.dev/reference/migrations/route-lazy-loading

What is it?

Lazy loading loads feature modules **only when needed**, improving initial load time.

Setup:

Step 1: In App Routing:

```
const routes: Routes = [
  {
    path: 'products',
    loadChildren: () =>
      import('./features/products/products.module').then(m => m.ProductsModule)
  }
```

```
];
```

Step 2: In the lazy-loaded module (`products-routing.module.ts`):

```
const routes: Routes = [
  { path: '', component: ProductListComponent },
  { path: ':id', component: ProductDetailComponent }
];
```

3. Nested (Child) Routes

Purpose:

Render nested views within a parent route (e.g., tabs, subpages).

Example:

parent-routing.module.ts

```
const routes: Routes = [
  {
    path: 'settings',
    component: SettingsComponent,
    children: [
      { path: 'profile', component: ProfileComponent },
      { path: 'security', component: SecurityComponent }
    ]
  }
];
```

parent.component.html

```
<nav>
  <a routerLink="profile">Profile</a>
  <a routerLink="security">Security</a>
</nav>
<router-outlet></router-outlet>
```

4. Route Guards

https://angular.dev/guide/routing/route-guards

Purpose:

Control access to routes (e.g., authentication, roles, unsaved changes).

Types of Guards:

Guard	Trigger
`CanActivate`	Before entering a route
`CanDeactivate`	Before leaving a route
`CanLoad`	Before loading a module
`CanActivateChild`	For child routes

Example: Auth Guard

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (!this.authService.isLoggedIn()) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }
}
```

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

5. Route Resolvers

https://angular.dev/guide/routing/data-resolvers

Purpose:

Preload data **before route activates**, ensuring the component has what it needs.

Example:

UserResolver

```
@Injectable({ providedIn: 'root' })
export class UserResolver implements Resolve<User> {
  constructor(private userService: UserService) {}

  resolve(route: ActivatedRouteSnapshot): Observable<User> {
    const id = route.paramMap.get('id');
    return this.userService.getUser(id!);
  }
}
```

Routing

```
{ path: 'users/:id', component: UserDetailComponent, resolve: { user: UserResolver } }
```

Component Access

```
this.route.data.subscribe(data => {
  this.user = data['user'];
});
```

Summary Table

Feature	Description	Example Use Case

Lazy Loading	Load modules only when needed	Product, Admin, User modules
Nested Routes	Child routes rendered inside parents	Settings > Profile, Security
Guards	Control navigation access	Auth checks, leave confirmations
Resolvers	Preload data before activating routes	Load user profile before view

Best Practices

- Use lazy loading for **feature modules** to optimize performance
- Use guards to enforce **authorization, unsaved changes, etc.**
- Prefer resolvers for **critical preload data**
- Use `canLoad` instead of `canActivate` to **block module loading**

Lab: Angular Routing – Lazy Loading, Nested Routes, Guards, Resolvers

Objective

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

- Configure routing for a feature module using **lazy loading**
- Set up **nested routes** inside a parent component
- Implement a **route guard** to protect access
- Use a **resolver** to preload data before route activation

Step 1: Create a New Angular Project

```
ng new routing-lab --routing=true --style=css
cd routing-lab
```

This sets up the Angular project with routing support.

Step 2: Generate Feature Module (Lazy Loaded)

```
ng generate module features/users --routing
ng generate component features/users/user-list
ng generate component features/users/user-detail
```

2. Add this to `app.routes.ts` (or `main.ts` if using `provideRouter` directly):

```
{
  path: 'users',
  loadChildren: () =>
    import('./features/users/users.module').then(m => m.UsersModule)
}
```

This will:

- Create a **lazy-loaded UsersModule**
- Automatically update the `AppRoutingModule`

Step 3: Add Nested Routes

Update `users-routing.module.ts`

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { UserListComponent } from './user-list/user-list.component';
import { UserDetailComponent } from './user-detail/user-detail.component';

const routes: Routes = [
 {
   path: '',
   component: UserListComponent,
   children: [
     { path: ':id', component: UserDetailComponent }
   ]
 }

];
```

Template for Parent (`user-list.component.html`)

```
<h2>User List</h2>
<ul>
  <li *ngFor="let id of [1, 2, 3]">
    <a [routerLink]="[id]">User {{ id }}</a>
  </li>
</ul>

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

Step 4: Add a Route Guard (CanActivate)

Generate Guard

```
ng generate guard auth
```

Edit [auth.guard.ts]()

```
import { CanActivateFn } from '@angular/router';

export const authGuard: CanActivateFn = (route, state) => {
  const loggedIn = confirm('Are you logged in?');
  return loggedIn;
};
```

Protect the route

Update `users-routing.module.ts`:

```

const routes: Routes = [
 {
   path: '',
   component: UserListComponent,
   canActivate: [authGuard], // Add your auth guard here if needed
   children: [
     { path: ':id', component: UserDetailComponent }
   ]
 }

];
```

Step 5: Add a Resolver

Generate Resolver

```
ng generate resolver user
```

Edit [user.resolver.ts]()

```
import { ResolveFn } from '@angular/router';
import { of } from 'rxjs';

export const userResolver: ResolveFn<any> = (route, state) => {
  const id = route.paramMap.get('id');
```

```
  return of({ id, name: 'User ' + id });
};
```

Apply Resolver in Route

In `users-routing.module.ts`:

```
{
  path: ':id',
  component: UserDetailComponent,
  resolve: { user: userResolver }
}
```

Use Resolved Data in [`user-detail.component.ts`]

```
import { ActivatedRoute } from '@angular/router';

export class UserDetailComponent {
  user: any;
  constructor(private route: ActivatedRoute) {}

 ngOnInit(): void {
   this.route.data.subscribe((data) => {
     this.user = data['user'];
   });
 }

}
```

Template

```
<h3>User Detail</h3>
<p>ID: {{ user.id }}</p>
<p>Name: {{ user.name }}</p>
```

Step 6: Run and Test

```
ng serve
```

Try:

- Navigating to `/users`
- Clicking a user link (`/users/1`)
- Trying access after declining the **guard prompt**
- Observing resolved data in the user detail page

Summary of What You've Built

Feature	Where Implemented
Lazy Loading	`UsersModule` in `AppRoutingModule`
Nested Routes	`UserListComponent` with `<router-outlet>`
Route Guard	`AuthGuard` used on `users` route
Resolver	`UserResolver` used on `:id` route

Bonus Challenge

- Create an **AdminModule** with its own guard (`AdminGuard`)
- Add a **NotFoundComponent** and handle unknown routes (`**`)
- Add `CanDeactivate` to warn on leaving form without saving

⚖️ When to use which?

Use Case	**Use Resolver**	**Use `ngOnInit()`**
Data is **required before** showing the page	Yes	No
Data is **optional** or can be loaded after component appears	No	Yes
Need to show a **spinner** or **partial UI** while loading	Difficult	Easy
SEO or SSR (Angular Universal) important	Yes	Less ideal
Route guards or authentication check with data	Yes	No

Source code for this exercise: https://github.com/wanmuz86/angular-int-adv-lab8-routing-lab

State Management in Angular

State management is how an application stores, updates, and shares data between components.

1. Global vs Local State

Type	Description	Examples
Local State	Data relevant only within a component or small tree	Form values, modal open/close flag
Global State	Shared across multiple features or the entire app	Logged-in user, cart, settings

Local State

- Managed inside a component (`let`, `@Input()`, `signals`)
- Reacts to user interaction, rarely shared

Global State

- Managed via services, signals, stores (NgRx, Akita, etc.)
- Used in multiple places (e.g., auth status, language, theme)

2. RxJS-based State vs Service-based State

Angular traditionally uses **services** and **RxJS observables** for state management.

A. Service-Based State (Imperative)

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

```
  isLoggedIn = false;

  login() {
    this.isLoggedIn = true;
  }
}
```

- Simple and easy to follow
- Not reactive (components must **manually subscribe or check** changes)

B. RxJS-based State (Reactive)

```
@Injectable({ providedIn: 'root' })
export class CartService {
  private cartSubject = new BehaviorSubject<CartItem[]>([]);
  cart$ = this.cartSubject.asObservable();

  addToCart(item: CartItem) {
    const updated = [...this.cartSubject.value, item];
    this.cartSubject.next(updated);
  }
}
```

- Uses `BehaviorSubject`, `ReplaySubject`, or `Observable`
- Fully reactive: components subscribe and auto-update
- Can use `scan`, `switchMap`, `combineLatest`, etc. for power logic

Feature	**Service State**	**RxJS State**
Reactivity	Manual	Observable streams
Change propagation	Imperative	Automatic via subscriptions

Suited for complex flows	No	Yes

3. Introduction to Angular Signals (New Reactive State)

What are Signals?

Signals are a **reactive primitive** introduced in Angular 16+ to simplify state handling with **fine-grained reactivity** and **minimal boilerplate**.

Basic Usage:

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

const counter = signal(0);
counter.set(counter() + 1); // increment
console.log(counter());     // access value
```

Signals in a Component:

```
@Component({ ... })
export class CounterComponent {
  count = signal(0);

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

Derived Signals:

```
fullName = computed(() => `${this.first()} ${this.last()}`);
```

Effects:

```
effect(() => {
  console.log('count changed:', this.count());
});
```

Signals vs RxJS vs Service State

Feature	Signals	RxJS + Observable	Plain Service State
Reactivity	Fine-grained	Stream-based	Manual
Boilerplate	Minimal	More setup	Minimal
Async handling	Needs `fromObservable`	Native via pipe/map	Not reactive
Component usage	Direct binding	Async pipe	Manual binding
Learning curve	Easy to moderate	Steeper with RxJS ops	Easy

Summary

Concept	Purpose	Key Tools Used
Local State	Inside one component	`let`, `@Input()`, `signals`
Global State	Shared across app	Services, RxJS, Signals

RxJS State	Reactive, observable-driven	`BehaviorSubject`, `Observable`
Service State	Central logic, not reactive	Plain class properties
Signals	New fine-grained reactive system	`signal()`, `computed()`, `effect()`

State Management with RxJS

RxJS provides a powerful and reactive approach to manage application state using **Observables**, which enable real-time data flow and reactive updates.

Why Use RxJS for State Management?

- Reactive and composable
- Great for **centralized state sharing** across components/services
- Ideal for **handling async data** and side-effects (API calls, WebSocket, etc.)
- Lightweight compared to full libraries like NgRx

`BehaviorSubject` – The Core Reactive Store

What is `BehaviorSubject`?

- A **type of Subject** that stores the latest emitted value.
- On subscription, it **immediately emits the last value** (unlike regular `Subject`).
- Useful for **holding and emitting state**.

Syntax

```
import { BehaviorSubject } from 'rxjs';

const state$ = new BehaviorSubject<number>(0); // initial value is 0

state$.next(5);  // update state
state$.subscribe(value => console.log(value)); // logs: 5
```

Using BehaviorSubject for App State

Example: User Auth State

```
// auth.service.ts
export class AuthService {
```

```
  private authState$ = new BehaviorSubject<boolean>(false);

  get isLoggedIn$(): Observable<boolean> {
    return this.authState$.asObservable();
  }

  login() {
    this.authState$.next(true);
  }

  logout() {
    this.authState$.next(false);
  }
}
```

`async` Pipe in Templates

The `async` pipe automatically:

- **Subscribes** to an observable
- **Unsubscribes** when the component is destroyed
- **Emits values directly in the template**

Example:

```
<!-- app.component.html -->
<div *ngIf="authService.isLoggedIn$ | async; else loggedOut">
  Logged In!
</div>
<ng-template #loggedOut>Not Logged In</ng-template>
```

Component Integration

```
// app.component.ts
@Component({ ... })
export class AppComponent {
  isLoggedIn$ = this.authService.isLoggedIn$;

  constructor(private authService: AuthService) {}
```

```
}
```

No manual subscription needed thanks to `async` pipe

Best Practices

Do	Avoid
Use `BehaviorSubject` for state with initial values	Using `Subject` without initial values for shared state
Use `.asObservable()` to expose state safely	Exposing raw `BehaviorSubject` directly
Use `async` pipe in templates to auto-subscribe	Manually subscribing in templates or risking memory leaks
Use operators like `map`, `switchMap`, `filter` for state transformation	Overusing `.subscribe()` in components

Summary

Concept	Purpose
`BehaviorSubject<T>`	Reactive state container with latest value
`Observable<T>`	Stream interface for state consumption

async pipe	Auto manage observable subscriptions in template

Lab: User Theme Toggle with RxJS and BehaviorSubject

Objective

By the end of this lab, you will:

- Use `BehaviorSubject` to manage application state
- Create a simple `ThemeService` that toggles between Light and Dark mode
- Use `async` pipe in the component template to reflect changes

Step 1: Create a New Angular Project

```
ng new rxjs-state-lab --routing=false --style=csscd rxjs-state-lab
ng generate component ThemeToggle
ng generate service Theme
```

Step 2: Create the ThemeService

```
// src/app/theme.service.ts
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

export type ThemeMode = 'light' | 'dark';

@Injectable({
  providedIn: 'root',
})
export class ThemeService {
  private themeMode$ = new BehaviorSubject<ThemeMode>('light');

  get currentTheme$(): Observable<ThemeMode> {
    return this.themeMode$.asObservable();
  }

  toggleTheme() {
    const current = this.themeMode$.getValue();
    const next = current === 'light' ? 'dark' : 'light';
    this.themeMode$.next(next);
  }
}
```

Step 3: Update the Component

```
import { Component, OnInit } from '@angular/core';
import { ThemeService } from '../theme.service';
import { Observable } from 'rxjs';
import { AsyncPipe } from '@angular/common';
import { NgClass } from '@angular/common';
@Component({
 selector: 'app-theme-toggle',
 imports: [AsyncPipe, NgClass],
 templateUrl: './theme-toggle.component.html',
 styleUrl: './theme-toggle.component.css'
})
export class ThemeToggleComponent implements OnInit {

 theme$!: Observable<string>;

 constructor(private themeService: ThemeService) {}


 ngOnInit() {
   this.theme$ = this.themeService.currentTheme$;
 }

 toggle() {
   this.themeService.toggleTheme();
 }


}
```

Step 4: Create the HTML Template

```
<!-- src/app/theme-toggle/theme-toggle.component.html -->
<div [ngClass]="theme$ | async">
  <p>Current Theme: <strong>{{ theme$ | async }}</strong></p>
  <button (click)="toggle()">Toggle Theme</button>
</div>
```

Step 5: Add Theme Styles

```
/* src/app/theme-toggle/theme-toggle.component.css */
```

```css
.light {
  background-color: #ffffff;
  color: #333;
  padding: 20px;
  text-align: center;
}

.dark {
  background-color: #333;
  color: #ffffff;
  padding: 20px;
  text-align: center;
}
```

Step 6: Use the Component in App

```html
<!-- src/app/app.component.html -->
<app-theme-toggle></app-theme-toggle>
```

Result

You can now:

- See the current theme displayed and styled accordingly
- Toggle between "light" and "dark" mode using a button
- Enjoy a clean reactive state managed with `BehaviorSubject` and `async` pipe

Lab: Angular State Management – Local, Global, RxJS, and Signals

Objective

By the end of this lab, you will:

- Understand the difference between **local** and **global** state
- Build a simple state service using **RxJS BehaviorSubject**
- Compare it with a **non-reactive service-based** approach
- Use **Angular Signals** to manage reactive local state inside a component

Create a new Angular project:

```
ng new state-lab --routing=false --style=css
cd state-lab
```

Step 1: Generate a Counter Component & Services

```
ng generate component counter
ng generate service state/counterStore   # RxJS-based
ng generate service state/counterStatic  # Service-based
```

Step 2: Set Up Local State in Component

counter.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
})
export class CounterComponent {
  count = 0; // Local state
```

```
  increment() {
    this.count++;
  }
}
```

counter.component.html

```
<h2>Local State</h2>
<p>Count: {{ count }}</p>
<button (click)="increment()">Increment</button>
<hr>
```

Step 3: Global State Using Plain Service (Non-reactive)

[`counter-static.service.ts`]

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

@Injectable({ providedIn: 'root' })
export class CounterStaticService {
  count = 0;

  increment() {
    this.count++;
  }

  getValue() {
    return this.count;
  }
}
```

Inject & Use in [`counter.component.ts`]

```
constructor(public staticCounter: CounterStaticService) {}

incStatic() {
  this.staticCounter.increment();
}
```

Update Template

```
<h2>Global (Service-Based)</h2>
<p>Static Count: {{ staticCounter.getValue() }}</p>
<button (click)="incStatic()">Increment Static</button>
<hr>
```

Step 4: Global State Using RxJS (Reactive)

counter-store.service.ts

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class CounterStoreService {
  private _count = new BehaviorSubject<number>(0);
  count$: Observable<number> = this._count.asObservable();

// Expose the double counter as an Observable
 doubleCounter$: Observable<number> = new Observable(observer => {
   this._counter.subscribe(value => {
     observer.next(value * 2);
   });
 });


  increment() {
    this._count.next(this._count.value + 1);
  }
}
```

Use in Component:

```
import { Component } from '@angular/core';
import { CounterStoreService } from '../state/counter-store.service';
import { Observable } from 'rxjs';
@Component({
 selector: 'app-counter',
 imports: [],
 templateUrl: './counter.component.html',
 styleUrl: './counter.component.css'
})
export class CounterComponent {
 countRx$!: Observable<number>;
```

```
 constructor(private rxStore: CounterStoreService) {}

 ngOnInit() {
   this.countRx$ = this.rxStore.count$;
 }

 incRx() {
   this.rxStore.increment();
 }



}
```

Template:

https://angular.dev/api/common/AsyncPipe

The Angular AsyncPipe is a built-in pipe that simplifies working with asynchronous data sources like Observables and Promises directly within component templates.

It provides a convenient way to subscribe to these sources and display their emitted values, while also handling important aspects of subscription management.

```
<h2>Global (RxJS-Based)</h2>
<p>Reactive Count: {{ countRx$ | async }}</p>
<button (click)="incRx()">Increment Reactive</button>
<hr>
```

Static vs Reactive State in Angular

Aspect	Static (Non-Reactive)	Reactive (RxJS)
Definition	A plain value (e.g. `count = 0`)	A value wrapped in an observable (`BehaviorSubject` etc.)

Change Detection	Does **not** trigger UI updates automatically	☐ Triggers UI updates **automatically** when value changes
UI Binding	Must call a method to get value: `{{ staticCounter.getValue() }}`	Can bind directly: `{{ countRx$ |async}}
Manual Updates	You must manually call `getValue()` again after updates	View updates automatically via reactive bindings
State Sharing	Difficult for many components to observe changes	Easy — all subscribers/components get the update
Use Case	For simple logic, config, or values that don't change often	For UI data, shared state, or anything dynamic and user-facing

Step 5: Local State with Angular Signals

Use Signals (Angular 16+)

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

signalCount = signal(0);

incSignal() {
  this.signalCount.update(n => n + 1);
}
```

Template:

```
<h2>Local (Signals)</h2>
<p>Signal Count: {{ signalCount() }}</p>
<button (click)="incSignal()">Increment Signal</button>
```

Step 6: Update AppComponent to Use Counter

`app.component.html`

```
<app-counter></app-counter>
```

Step 7: Run & Observe

```
ng serve
```

Test:

- All 4 state management styles:
  - Local state (primitive)
  - Service (non-reactive global)
  - RxJS observable (reactive global)
  - Signals (local reactivity)
- Try multiple button clicks, inspect behavior and re-rendering

Summary

Method	Scope	Reactive ?	Tools Used
Local primitive	Component	No	`let` variable
Static Service	Global	No	`get/set` in service

RxJS BehaviorSubject	Global	Yes	`BehaviorSubject`, `async`
Signals	Component	Yes	`signal()`, `.update()`

Bonus Challenge

- Use `computed()` to derive a double of the signal count
- Add `effect()` to log when signal value changes
- Add `decrement()` functionality to all states

```
import { Component, computed, effect, signal } from '@angular/core';

@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
})
export class CounterComponent {
  //  Reactive signal
  signalCount = signal(0);

  //  Computed signal
  doubleCount = computed(() => this.signalCount() * 2);

  //  Effect: side effect when signal changes
  logEffect = effect(() => {
    console.log('Signal count changed:', this.signalCount());
  });

  //  Methods
  incSignal() {
    this.signalCount.update(n => n + 1);
  }

  decSignal() {
    this.signalCount.update(n => n - 1);
  }
```

```
}
```

```html
<h2>Local State with Signals</h2>
<p>Signal Count: {{ signalCount() }}</p>
<p>Double Count (computed): {{ doubleCount() }}</p>

<button (click)="incSignal()">Increment</button>
<button (click)="decSignal()">Decrement</button>
```

Feature / Aspect	**Static (Non-Reactive)**	**RxJS (Reactive)**	**Signal (Reactive)**
Definition	Plain variable (count = 0)	Observable stream (BehaviorSubject, Observable)	Reactive primitive (signal(0))
Reactivity	No — Angular won't auto-update UI	Yes — `	async` triggers updates
UI Update Trigger	Manual or external CD	Auto with `	async`
Syntax Simplicity	Simple but limited	Verbose (next(), subscribe(), etc.)	Clean (signal(), .set(), .update())
Change Propagation	No	Yes — any subscriber gets updated	Yes — effects, computed, and bindings all update
Boilerplate	Low	Medium to High (Subjects, unsubscriptions)	Low (minimal setup)
Derived State	Manual calculations	map, combineLatest, etc.	computed(() => ...)
Side Effects	Manual	subscribe()	effect(() => ...)

Best Use Case	Very simple/local counters or flags	Complex async flows, data streams, shared state	Local/global state with clean syntax and auto reactivity

https://github.com/wanmuz86/lab-9-state-lab

DAY 3

Angular Signals

https://angular.dev/guide/signals
https://blog.angular-university.io/angular-signals/

Signals are a new reactive primitive in Angular that enable:

- Fine-grained reactivity (fewer unnecessary re-renders)
- Better performance (especially in zone-less apps)
- Simpler state management than RxJS or `@Input()` + `EventEmitter`

1. `signal()` – Reactive State Container

Purpose:

Create a **reactive value** that can be read and updated. When a signal's value changes, any dependent computations or views automatically update.

Syntax:

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

const count = signal(0);        // Create signal

count.set(5);                   // Set new value
count.update(n => n + 1);       // Increment

console.log(count());           // Access value
```

Use Cases:

- Local component state
- Toggle buttons, counters, filters
- Input form bindings (with [ngModel])

2. `computed()` – Derived Reactive Values

Purpose:

Create a **readonly computed value** that automatically updates when **dependent signals** change.

Syntax:

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

const firstName = signal('Ali');
const lastName = signal('Zain');

const fullName = computed(() => `${firstName()} ${lastName()}`);

console.log(fullName()); // → Ali Zain
```

Use Cases:

- Dynamic display values
- Labels, display summaries
- Filtering or conditional UI

3. `effect()` – Reactive Side Effects

Purpose:

Run **side effects** when signal values change (e.g., logging, API calls, DOM interaction).

Syntax:

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

const count = signal(0);

effect(() => {
  console.log('Count changed to:', count());
});
```

Use Cases:

- Trigger animations, logs, analytics

- Update non-template logic
- Sync state across services

Example in Component

```
@Component({ selector: 'app-counter', template: `{{ count() }}` })
export class CounterComponent {

  count = signal(0);
  double = computed(() => this.count() * 2);

  constructor() {
    effect(() => {
      console.log('Double:', this.double());
    });
  }

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

Advanced APIs

`toSignal()` – Convert RxJS Observable to Signal

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

const time$ = interval(1000);
const time = toSignal(time$); // Signal that updates every second
```

`input()` – Reactive @Input

Declares an input property as a signal, meaning its value is reactive and changes can be automatically propagated.

```
@Input({ required: true }) name = input<string>();
```

`model()` – Bi-directional binding (@Input/@Output replacement)

```
@Input({ alias: 'value', transform: model<string>() }) value = signal('');
```

Forms with Signals

This won't work:

```
<input [(ngModel)]="name()" /> <!-- ERROR -->
```

Use this instead:

```
<input [ngModel]="name()" (ngModelChange)="name.set($event)" />
```

You must use `[ngModel]` with `(ngModelChange)` explicitly for Signals.

RxJS vs Signals Comparison

Feature	Signals	RxJS Observables
Syntax	Simple (`count()`)	Verbose (`subscribe()`)
Reactivity Model	Pull-based	Push-based
Side Effects	`effect()`	`subscribe()`
Derived Values	`computed()`	`map()`, `combineLatest()`
Cleanup	Automatic	Manual (`unsubscribe()`)

Zone-less Support	Native (Angular 17+)	Needs workaround

Best Practices

- Use `signal()` for local, mutable state
- Use `computed()` for pure transformations
- Use `effect()` for side-effects only **(not state updates)**
- Use `toSignal()` to bridge from `Observable`
- Avoid circular dependencies between `effect()` and `signal.set()`
- Prefer Signals in zone-less and performance-critical apps

Summary Table

API	Description	Read/Write	Best For
`signal()`	Reactive variable	Yes/Yes	Component state
`computed()`	Derived reactive expression	Yes/No	Calculations, display values
`effect()`	Reactive side effect on dependency	No/No	Logging, sync, animations
`toSignal()`	Observable ➔ Signal bridge	Yes/No	Interop with RxJS

`input()`	Reactive @Input()	Yes/Yes	Inputs with signals
`model()`	Bi-directional input/output	Yes/Yes	Controlled component inputs

Lab: Angular Signals — Employee CRUD with signal(), computed(), effect()

Objective

By the end of this lab, you will:

- Use `signal()` to manage a list of employees
- Use `computed()` to filter/search employees
- Use `effect()` to react to state changes (e.g. log when employee list changes)
- Use ngModel and onModelChanged with signal
- Differentiate global and local state

Step 1: Create a New Angular Project

```
ng new employee-signal-lab --routing=false --style=css
cd employee-signal-lab
```

Step 2: Generate a Component and Service

```
ng generate component employee-manager-component
ng g service employee-service
```

Update `app.component.html` to:

```
<app-employee-manager></app-employee-manager>
```

Step 3: Setup Employee State with `signal()`

ng g interface models/employee

```
export interface Employee {
   id:number;
   name:string;
   department:string;
}
```

```
import { Injectable, signal , effect} from '@angular/core';
// Best practice : put inside models folder
// ng g interface models/employee

import { Employee } from './employee';

// Define the Employee interface
//
@Injectable({
 providedIn: 'root'
})

export class EmployeeService {
  // signal that will hold the list of employees
 // For encapsulation, we will not expose the signal directly
 // Instead, we will expose a readonly version of the signal
 private _employees = signal<Employee[]>([]);
 employees = this._employees.asReadonly();

 // signal that will hold the last id used
 // Everytime we add a new employee, we will increment this id => update
 private _lastId = signal<number>(1);

// use effect to log the employees whenever they change
 constructor() {
   effect(() => {
     console.log('Employees changed:', this._employees());
   });
 }


 addEmployee(name:string, department:string){
   const newEmployee:Employee = {
     id: this._lastId(),
     name: name,
     department: department
   }
   // Update the last id
   this._lastId.update(prev => prev+1)
```

```
   // Add the new employee to the list of signal employees
   // Array spread operator to create a new array with the new employee
   this._employees.update(prev=> [...prev, newEmployee]);
 }

 deleteEmployee(id:number){
   // Remove the employee with the given id from the list of employees
   this._employees.update(prev=> prev.filter(val => val.id !== id));
 }


}
```

```
ng g component employee-add-component
ng g component employee-list-component
```

```html
<h2>Add Employee</h2>
<!-- Usage of signal in form , using [ngModel] and (ngModelChange) as of
Angular 19 as [(ngModel)]/FormModules is not supported/stable with signals -->

<input type="text" placeholder="Name" name="name" [ngModel]="name()"
(ngModelChange)="name.set($event)"/>

<input type="text" placeholder="Department" name="department"
[ngModel]="department()" (ngModelChange)="department.set($event)"/>

<button (click)="addEmployee()">Add new Employee</button>
```

```typescript
import { Component, signal } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { EmployeeService } from '../employee-service';

@Component({
```

```
 selector: 'app-employee-add-component',
 imports: [FormsModule],
 templateUrl: './employee-add-component.html',
 styleUrl: './employee-add-component.css'
})
export class EmployeeAddComponent {

 constructor(private employeeService: EmployeeService) {

 }

 // Signals to hold the input values for name and department [ ngModel]
 name = signal<string>('');
 department = signal<string>('');

 addEmployee() {
   this.employeeService.addEmployee(this.name(), this.department());
   // Reset the input fields after adding the employee
   this.name.set('');
   this.department.set('');
 }

}
```

```
<app-employee-add-component></app-employee-add-component>
<hr>
<app-employee-list-component></app-employee-list-component>
```

```
import { Component } from '@angular/core';
import { EmployeeService } from '../employee-service';
import { Employee } from '../employee';
import { Signal } from '@angular/core';
@Component({
 selector: 'app-employee-list-component',
```

```
 imports: [],
 templateUrl: './employee-list-component.html',
 styleUrl: './employee-list-component.css'
})
export class EmployeeListComponent {

 employees!:Signal<Employee[]>;

 constructor(private employeeService:EmployeeService){
   // Retieve the employees from the service inside constructor
   this.employees = this.employeeService.employees;
 }



}
```

```
<h2>Employee List</h2>
@if(employees().length === 0){
   <p>No employee at the moment. Please add new employee</p>
}
@else {
@for (employee of employees(); track employee.id) {
<div>
   <p>Name: {{employee.name}}</p>
   <p>Department: {{employee.department}}</p>
   <button (click)="deleteEmployee(employee.id)">Delete Employee
   </button>
</div>
}
}
```

ng g component employee-list-item-component

```
import { Component, input } from '@angular/core';
import { Employee } from '../employee';


@Component({
 selector: 'app-employee-list-item-component',
 imports: [],
 templateUrl: './employee-list-item-component.html',
 styleUrl: './employee-list-item-component.css'
})
export class EmployeeListItemComponent {

 // Input property to receive the employee data from the parent component
 // input (with small i) is used to define an input property in Angular that
supports signals
 employee = input<Employee>();

 deleteEmployee(id: number) {
   // Emit an event to delete the employee
   // This method should be implemented in the parent component
   console.log(`Delete employee with id: ${id}`);
 }
 }
```

```
<div>
   <p>Name: {{employee()?.name}}</p>
   <p>Department: {{employee()?.department}}</p>
   <button (click)="deleteEmployee(employee()?.id!)">Delete Employee
   </button>
</div>
```

```
<h2>Employee List</h2>
@if(filteredEmployees().length === 0){
<p>No employee at the moment. Please add new employee</p>
}
```

```
@else {
<!-- @for have a track option to track the changes in the list built in -->
<div>
   <!-- Filter the list via onChange event on the input -->
   <!-- ngModel + derived signal example -->
   <input type="text" placeholder="Search by name"
   [ngModel]="searchTerm()"
   (ngModelChange)="searchTerm.set($event)" />
</div>

@for (employee of filteredEmployees(); track employee.id) {
<app-employee-list-item-component
[employee]="employee"></app-employee-list-item-component>
}
}
```

```
import { Component } from '@angular/core';
import { EmployeeService } from '../employee-service';
import { Employee } from '../employee';
import { Signal, signal,computed } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { EmployeeListItemComponent } from
'../employee-list-item-component/employee-list-item-component';
@Component({
 selector: 'app-employee-list-component',
 imports: [EmployeeListItemComponent, FormsModule],
 templateUrl: './employee-list-component.html',
 styleUrl: './employee-list-component.css'
})
export class EmployeeListComponent {

 // employees!:Signal<Employee[]>;
  // To hold the search term in the input field
 searchTerm = signal<string>('');

 filteredEmployees = computed(()=>
 this.employeeService.employees().filter(employee=>
employee.name.toLowerCase().includes(
   this.searchTerm().toLowerCase()
```

```
 }))

 constructor(private employeeService:EmployeeService){
   // Retieve the employees from the service inside constructor
  // this.employees = this.employeeService.employees;
 }



}
```

```
// src/app/employee-manager/employee-manager.component.ts
import { Component, signal, computed, effect } from '@angular/core';

interface Employee {
  id: number;
  name: string;
  department: string;
}

@Component({
  selector: 'app-employee-manager',
  templateUrl: './employee-manager.component.html',
  styleUrls: ['./employee-manager.component.css'],
})
export class EmployeeManagerComponent {
  // State signals
  employees = signal<Employee[]>([]);
  name = signal('');
  department = signal('');
  searchTerm = signal('');

  nextId = signal(1); // For simple ID generation

  constructor() {
    // Log effect when list changes
    effect(() => {
      console.log('Employee list changed:', this.employees());
    });
  }

  // Computed: filtered employees
  filteredEmployees = computed(() =>
    this.employees().filter(emp =>
      emp.name.toLowerCase().includes(this.searchTerm().toLowerCase())
    )
```

```
  );

  // Actions
  addEmployee() {
    const newEmployee: Employee = {
      id: this.nextId(),
      name: this.name(),
      department: this.department(),
    };

    this.employees.update((prev) => [...prev, newEmployee]);
    this.nextId.update(id => id + 1);
    this.name.set('');
    this.department.set('');
  }

  deleteEmployee(id: number) {
    this.employees.update((list) => list.filter(emp => emp.id !== id));
  }
}
```

Step 4: Create the Template

```
<!-- src/app/employee-manager/employee-manager.component.html -->
<h2>Employee Manager (Signals)</h2>


<input type="text" [ngModel]="name()" (ngModelChange)="name.set($event)"
placeholder="Name">
<input type="text" [ngModel]="department()" (ngModelChange)="department.set($event)"
placeholder="Department">

<button (click)="addEmployee()">Add Employee</button>

<hr>
<input type="text" [ngModel]="searchTerm()" (ngModelChange)="searchTerm.set($event)"
placeholder="Search...">

<h3>Employee List (Filtered)</h3>
<ul>
 <li *ngFor="let emp of filteredEmployees()">
   {{ emp.id }} - {{ emp.name }} ({{ emp.department }})
   <button (click)="deleteEmployee(emp.id)">Delete</button>
 </li>
</ul>
```

Step 5: Run and Test

```
ng serve
```

Test:

- Adding employees
- Searching by name
- Deleting employees
- Watching `console.log` triggered by `effect()`

Summary of Concepts Practiced

Concept	Description	Example
signal()	Reactive variable for local state	employees = signal<Employee[]>([])
computed()	Derived value reacting to another signal	filteredEmployees = computed(...)
effect()	Executes side-effects when a signal changes	effect(() => console.log(...))
model()	Bi-directional binding to simplify i@Input/@Output pairs	Input({ alias: 'value', transform: model<string>() }) value = signal('')

Bonus Challenge

1. **Add `selectedId = signal<number | null>()` to highlight or edit an employee**
2. **Add `isListEmpty = computed(() => employees().length === 0)`**
3. **Use `effect()` to `alert()` when the employee list reaches 5 people**
4. **Add update functionality (edit + save)**

[employee-manager.component.ts]()

```
import { Component, signal, computed, effect } from '@angular/core';

interface Employee {
  id: number;
  name: string;
  department: string;
}

@Component({
  selector: 'app-employee-manager',
  templateUrl: './employee-manager.component.html',
  styleUrls: ['./employee-manager.component.css']
})
export class EmployeeManagerComponent {
  // Signals
  employees = signal<Employee[]>([]);
  name = signal('');
  department = signal('');
  searchTerm = signal('');
  nextId = signal(1);
  selectedId = signal<number | null>(null); // Bonus
  isEditing = signal(false); // Bonus

  constructor() {
    // Log when employee list changes
    effect(() => {
      console.log('Employee list changed:', this.employees());
    });

    // Bonus: Alert when list reaches 5 employees
    effect(() => {
      if (this.employees().length === 5) {
        alert('🎉 5 employees reached!');
      }
    });
  }

  // Computed: Filtered list
  filteredEmployees = computed(() =>
    this.employees().filter(emp =>
      emp.name.toLowerCase().includes(this.searchTerm().toLowerCase())
    )
  );

  // Bonus: check if list is empty
  isListEmpty = computed(() => this.employees().length === 0);
```

```
  addEmployee() {
    if (this.name() && this.department()) {
      const newEmployee: Employee = {
        id: this.nextId(),
        name: this.name(),
        department: this.department(),
      };
      this.employees.update((prev) => [...prev, newEmployee]);
      this.nextId.update(id => id + 1);
      this.name.set('');
      this.department.set('');
    }
  }

  deleteEmployee(id: number) {
    this.employees.update((list) => list.filter(emp => emp.id !== id));
    if (this.selectedId() === id) {
      this.selectedId.set(null);
      this.isEditing.set(false);
    }
  }

  selectEmployee(emp: Employee) {
    this.selectedId.set(emp.id);
    this.name.set(emp.name);
    this.department.set(emp.department);
    this.isEditing.set(true);
  }

  saveEmployee() {
    this.employees.update((list) =>
      list.map(emp =>
        emp.id === this.selectedId()
          ? { ...emp, name: this.name(), department: this.department() }
          : emp
      )
    );
    this.cancelEdit();
  }

  cancelEdit() {
    this.selectedId.set(null);
    this.name.set('');
    this.department.set('');
    this.isEditing.set(false);
  }
}
```

employee-manager.component.html

```html
<h2>Employee Manager (Signals)</h2>

<input
  type="text"
  [ngModel]="name()"
  (ngModelChange)="name.set($event)"
  placeholder="Name"
/>
<input
  type="text"
  [ngModel]="department()"
  (ngModelChange)="department.set($event)"
  placeholder="Department"
/>

<button *ngIf="!isEditing()" (click)="addEmployee()">Add Employee</button>
<button *ngIf="isEditing()" (click)="saveEmployee()">Save</button>
<button *ngIf="isEditing()" (click)="cancelEdit()">Cancel</button>

<hr />

<input
  type="text"
  [ngModel]="searchTerm()"
  (ngModelChange)="searchTerm.set($event)"
  placeholder="Search..."
/>

<h3>Employee List (Filtered)</h3>
<p *ngIf="isListEmpty()">No employees yet. Please add some!</p>
<ul>
  <li
    *ngFor="let emp of filteredEmployees()"
    [style.fontWeight]="emp.id === selectedId() ? 'bold' : 'normal'"
  >
    {{ emp.id }} - {{ emp.name }} ({{ emp.department }})
    <button (click)="selectEmployee(emp)">Edit</button>
    <button (click)="deleteEmployee(emp.id)">Delete</button>
  </li>
</ul>
```

https://github.com/wanmuz86/angular-int-adv-lab11-employee

Lifecycle Control in Angular Signals

Focus: `untracked()` and `onCleanup()` (formerly mislabeled as `cleanup()`)

As Angular signals introduce fine-grained reactivity, controlling how signals track dependencies and how effects clean up becomes essential for performance and correctness.

1. `untracked()` – Skip Dependency Tracking

Purpose

Access a signal's value **without registering it as a dependency** of `computed()` or `effect()`.

Syntax

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

effect(() => {
  const current = untracked(() => count());
  console.log('Count at effect init:', current);
});
```

Use Cases

- Snapshot values once without reactive updates
- Avoid unnecessary re-renders in `computed()`
- Compare previous and current values in `effect()`

Example

```
effect(() => {
  const prev = untracked(() => count());
  if (count() > prev + 5) {
    console.log('Count jumped by more than 5');
  }
});
```

2. `onCleanup()` – Teardown Logic for Effects

Purpose

Register **cleanup logic** in `effect()` that runs:

- Before the next execution, and
- When the effect is destroyed (e.g., on component teardown)

Correct Syntax

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

effect((onCleanup) => {
  console.log('Effect started');

  const interval = setInterval(() => {
    console.log('Interval running...');
  }, 1000);

  //  Cleanup logic
  onCleanup(() => {
    clearInterval(interval);
    console.log('Effect cleaned up');
  });
});
```

Use Cases

- Canceling timers, subscriptions, event listeners
- Resetting component-local state
- Cleaning up non-Angular side effects (DOM, sockets, etc.)

`untracked()` vs `onCleanup()`

Feature	Purpose	Used In

`untracked()`	Access signal without tracking	`computed()`, `effect()`
`onCleanup()`	Run teardown logic before re-run	`effect()` only

Best Practices

- Use `untracked()` when you want to read a signal **without making the effect depend on it**
- Use `onCleanup()` inside `effect()` when:
  - Starting timers or subscriptions
  - Binding to external events (DOM, sockets)
  - Managing temporary or non-Angular state

Summary

API	Purpose	Typical Use Case
`untracked()`	Skip tracking signal dependencies	Snapshot value, avoid side effects
`onCleanup()`	Register teardown logic	Unsubscribe, clear timers, cleanup

Lab: Angular Signals — Conditional Tracking with `untracked()` and `onCleanup()`

Objective

By the end of this lab, you'll:

- Understand **how reactive tracking works conditionally**
- Use `untracked()` to prevent reactivity in parts of an `effect()`
- Use `cleanup()` to **safely manage side effects** like intervals

Step 1: Create a New Angular Project

```
ng new signal-conditional-tracking-lab --routing=false --style=css
cd signal-conditional-tracking-lab
```

Step 2: Generate the Component

```
ng generate component signal-lifecycle
```

Update `app.component.html`:

```
<app-signal-lifecycle></app-signal-lifecycle>
```

Step 3: Setup Conditional Tracking with Cleanup

Open `signal-lifecycle.component.ts`:

```
import { Component, signal, effect, untracked } from '@angular/core';

@Component({
 selector: 'app-signal-lifecycle',
 templateUrl: './signal-lifecycle.component.html',
})
export class SignalLifecycleComponent {
 counter = signal(0);
```

```
 constructor() {
   effect((onCleanup) => {
     const value = this.counter(); // this makes the effect reactive

     if (value > 5) {
       console.log('[IF] Tracked: Counter is above 5 ->', value);

       const interval = setInterval(() => {
         console.log('[Interval] Running while counter > 5:', this.counter());
       }, 1000);

       // cleanup
       onCleanup(() => {
         clearInterval(interval);
         console.log('[Cleanup] Interval cleared because counter changed');
       });

     } else {
       const snapshot = untracked(() => this.counter());
       console.log('[ELSE] Untracked snapshot (5 or below):', snapshot);

       console.log('[ELSE] Direct read (still tracked!):', this.counter());
     }
   });
 }

 increment() {
   this.counter.update(n => n + 1);
 }

 reset() {
   this.counter.set(0);
 }
}
```

Step 4: Update the Template

In `signal-lifecycle.component.html`:

```
<h2>Conditional Signal Tracking + Cleanup</h2>
<p>Counter: {{ counter() }}</p>
<button (click)="increment()">Increment</button>
<button (click)="reset()">Reset</button>
```

Step 5: Run the App and Observe

```
ng serve
```

Try this:

1. Open the browser console.
2. Click **Increment** to increase the counter.
3. Once the value passes **5**, an interval starts and logs every second.
4. Click **Reset** to bring it back to 0.

What to Observe:

- The interval only starts when the counter goes above 5.
- `cleanup()` clears the previous interval every time the effect re-runs.
- `untracked()` ensures the snapshot in the `else` block does **not** trigger reactivity on its own.
- The condition `if (this.counter() > 5)` is **tracked** and responsible for triggering the effect.

Bonus: Fully Untracked Effect Example (Optional)

If you want the entire effect to be non-reactive (runs only once):

```
constructor() {
  effect(() => {
    const value = untracked(() => this.counter());

    if (value > 5) {
      console.log('[Untracked IF] > 5:', value);
    } else {
      console.log('[Untracked ELSE] <= 5:', value);
    }
  });
}
```

- This version will **only run once** on component creation.

Summary Table

Concept	Description	Reactive ?
`this.counter()` inside `effect()`	Tracked read – triggers re-runs	Yes
`untracked(() => this.counter())`	Snapshot – no tracking	No
`cleanup(() => { ... })`	Clears resources on effect teardown	N/A
`setInterval()` inside effect	Simulates a side effect needing cleanup	N/A

Using Signals in Templates & Services

Angular Signals introduce a **reactive and fine-grained** state management approach. They work naturally in both **templates** and **services**, enhancing reactivity without needing external tools like RxJS or NgRx for simpler use cases.

1. Using `signal()` in Templates

Template Binding

When you use signals in components, you can access them directly in templates using **function call syntax**.

Example:

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

@Component({
  selector: 'app-counter',
  template: `<p>Count: {{ count() }}</p>`
})
export class CounterComponent {
  count = signal(0);

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

Rules:

- You **must use** `()` to access signal values in templates: `{{ count() }}`
- Signals are **zone-friendly** and automatically update the view

Event Binding:

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

2. Using `signal()` and `computed()` in Services

Purpose:

Use signals in services to create a global **reactive state** that components can subscribe to or use directly.

Example:

```
@Injectable({ providedIn: 'root' })
export class CounterService {
  private _count = signal(0);
  count = this._count.asReadonly(); // expose readonly signal

  increment() {
    this._count.update(n => n + 1);
  }

  reset() {
    this._count.set(0);
  }

  double = computed(() => this._count() * 2);
}
```

In Component:

```
constructor(public counterService: CounterService) {}
```

In Template:

```
<p>Count: {{ counterService.count() }}</p>
<p>Double: {{ counterService.double() }}</p>
<button (click)="counterService.increment()">Increment</button>
```

Benefits of Signals in Templates and Services

Feature	Benefit

Function-call in template	No need for `async` pipe or manual subscriptions
Auto change detection	No zones needed (if zone-less), fine-grained reactivity
Global reactive state	Easier than `BehaviorSubject` for many use cases
Composability	Can use `computed()` and `effect()` for derived logic

What to Avoid

- Avoid mixing signals and observables without converting (`toSignal` / `toObservable`)
- Avoid setting signals directly in templates (write logic in component/service)
- Avoid calling `signal()` inside `computed()` or `effect()` without understanding reactivity cost

Summary

Where	How to Use
In templates	`{{ signalName() }}` – always use parentheses
In services	Use `signal()` for state, `computed()` for derived logic
Sharing state	Inject signal-holding service into component

Lab: Using Signals in Templates & Services

Objective

By the end of this lab, you will:

- Use `signal()` to manage state in a component template
- Use `signal()` and `computed()` in a service to hold global reactive state
- Bind signal values in the HTML template using `signal()`
- Call service methods to modify signal values

Step 1: Create a New Angular Project

```
ng new signal-service-lab --routing=false --style=css
cd signal-service-lab
```

Step 2: Create Component and Service

```
ng generate component counter-component
ng generate service counter-service
```

Update `app.component.html`:

```
<app-counter></app-counter>
```

Step 3: Use `signal()` in Component

counter.component.ts

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

@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html'
})
```

```
export class CounterComponent {
  localCount = signal(0);

  incrementLocal() {
    this.localCount.update(n => n + 1);
  }

  decrementLocal() {
    this.localCount.update(n => n - 1);
  }
}
```

counter.component.html

```
<h2>Local Signal (Component)</h2>
<p>Local Count: {{ localCount() }}</p>
<button (click)="incrementLocal()"> Increment</button>
<button (click)="decrementLocal()"> Decrement</button>
<hr>
```

Step 4: Use `signal()` in Service

counter.service.ts

```
import { Injectable, signal, computed } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class CounterService {
  private _count = signal(10);  // Initial global count
  count = this._count.asReadonly();

  double = computed(() => this._count() * 2);

  increment() {
    this._count.update(n => n + 1);
  }

  reset() {
    this._count.set(10);
  }
}
```

Step 5: Inject Service into Component

counter.component.ts (add below local signal logic)

```
import { CounterService } from '../counter.service';

constructor(public counterService: CounterService) {}
```

counter.component.html (add below local section)

```
<h2>Global Signal (Service)</h2>
<p>Global Count: {{ counterService.count() }}</p>
<p>Double: {{ counterService.double() }}</p>

<button (click)="counterService.increment()">Global Increment</button>
<button (click)="counterService.reset()">Reset</button>
```

Step 6: Run and Test

```
ng serve
```

Try:

- Incrementing both **local** and **service** signal values
- Watching `double` update reactively as `count` changes
- Inspecting how signals automatically update the DOM without `async` pipe or manual subscription

Summary of Concepts Practiced

Feature	Purpose	Location
`signal()`	Local reactive state	Component & Service

`computed()`	Derive value from signal	Service
Template binding	Use `{{ signalName() }}`	HTML
State sharing	Use signal-holding service globally	Injected in component

Bonus Challenge

- Add a third value: `triple = computed(() => this._count() * 3)`
- Create a second component that also uses the `CounterService`
- Style updated values using `[class]` or `ngClass` when count changes

Update `counter.service.ts` to add `triple`

```
import { Injectable, signal, computed } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class CounterService {
  private _count = signal(10); // Initial global count
  count = this._count.asReadonly();

  double = computed(() => this._count() * 2);
  triple = computed(() => this._count() * 3); //  Bonus addition

  increment() {
    this._count.update(n => n + 1);
  }

  reset() {
    this._count.set(10);
  }
}
```

Create Second Component

```
ng generate component counter-viewer
```

counter-viewer.component.ts

```
import { Component } from '@angular/core';
import { CounterService } from '../counter.service';

@Component({
  selector: 'app-counter-viewer',
  templateUrl: './counter-viewer.component.html',
  styleUrls: ['./counter-viewer.component.css'],
})
export class CounterViewerComponent {
  constructor(public counterService: CounterService) {}
}
```

counter-viewer.component.html

```
<h2>Viewer Component</h2>

<p [ngClass]="{ high: counterService.count() >= 15 }">
  Shared Global Count: {{ counterService.count() }}
</p>
<p>Triple: {{ counterService.triple() }}</p>
```

Add styling for class `.high` in `counter-viewer.component.css`

```
.high {
  color: red;
  font-weight: bold;
}
```

Update `app.component.html` to include second component

```
<app-counter></app-counter>
<hr />
<app-counter-viewer></app-counter-viewer>
```

Final enhancement (Optional): Highlight count if value is high in `counter.component.html`

Add to the global count section:

```
<p [class.high]="counterService.count() >= 15">
  Global Count: {{ counterService.count() }}
</p>
```

And add this to `counter.component.css`:

```
.high {
  color: green;
  font-weight: bold;
}
```

Run the app

```
ng serve
```

You'll now see:

- **Triple value** being computed reactively.
- **Second component** reacting to global signal.
- **Conditional styling** (e.g., `class="high"`) based on signal values.

Signal Inputs in Angular 17+

https://angular.dev/guide/components/inputs

What are Signal Inputs?

Signal Inputs allow you to declare `@Input()` properties as **signals**, giving you direct reactive access to input values inside the component.

Syntax

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

@Component({
  selector: 'app-user-card',
  template: `<p>Hello, {{ name() }}</p>`
})
export class UserCardComponent {
  name = input<string>('Guest'); // signal input with default value
}
```

Benefits of Signal Inputs

Feature	Advantage
Reactive by default	Automatically re-evaluates when input changes
No need for `ngOnChanges()`	Access latest value without lifecycle hooks
Composable with `computed`	Use inside computed/effect without extra setup

Simplified API	Cleaner than `@Input()` + `set`/`ngOnChanges`

How It Works

- `input<T>()` creates a **signal-like version** of an input binding
- Can be used like any signal: `name()` to read the current value
- Replaces the need for `@Input() name!: string` + manual tracking

Example: Parent to Child Binding with Signal Input

Parent Component

```
<app-user-card [name]="'Aisyah'"></app-user-card>
```

Child Component (Angular 17+)

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

@Component({
  selector: 'app-user-card',
  template: `<p>Hello, {{ name() }}</p>`
})
export class UserCardComponent {
  name = input<string>();
}
```

Usage with `computed()` and `effect()`

```
fullName = computed(() => `${this.name()} from Angular`);

constructor() {
  effect(() => {
    console.log('Name changed:', this.name());
  });
```

```
}
```

No need for `ngOnChanges()` or manual change detection logic!

Signal Inputs vs Traditional Inputs

Feature	`@Input()`	`input()` (Signal Input)
Reactive Access	Manual (`ngOnChanges`)	Built-in reactivity
Usable in `computed()`	Requires extra logic	Native
Read value in template	`{{ name }}`	`{{ name() }}`
Bind to changes	`ngOnChanges()` required	`effect()` or signal logic

Best Practices

- Use `input()` when:
  - You want reactive access to an input
  - You're using signals, `computed`, or `effect` in the component
- Still use classic `@Input()` if you need compatibility with libraries or decorators (e.g., form integrations)
- Always use `()` to access signal input value

Summary

Concept	Description
`input()`	Angular 17+ API to declare signal inputs
Usage	Use like any signal: `name()`
Benefits	No `ngOnChanges()`, full reactivity
Composable	Works with `computed()` and `effect()`

Source code reference: https://github.com/wanmuz86/angular-int-adv-lab10-signal-service

Angular Signals – Best Practices & Pitfalls

Signals introduce a new **reactivity model** in Angular that's more fine-grained, zone-less friendly, and easier to manage compared to `RxJS` or `ngOnChanges`.

Best Practices

1. Use `signal()` for Local Reactive State

- Ideal for values that live within a component (e.g., counters, form inputs, toggles).

```
count = signal(0);
```

2. Expose Readonly Signals from Services

- Prevents accidental mutation from outside the service.

```
private _count = signal(0);
count = this._count.asReadonly();
```

3. Use `computed()` for Pure Derived Values

- Great for UI labels, summaries, filters.

```
double = computed(() => this.count() * 2);
```

4. Use `effect()` for Side Effects Only

- Logging, subscriptions, syncing DOM or services.
- Avoid changing signals inside `effect()`.

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

5. Use `untracked()` Inside `effect()` or `computed()` When Needed

- Prevents unnecessary dependencies or loops.

```
effect(() => {
  const prev = untracked(() => this.count());
});
```

6. Use onC`leanup()` in `effect()`

- Manage timers, subscriptions, or DOM listeners.

```
effect(() => {
  const id = setInterval(() => { ... }, 1000);
  onCleanup(() => clearInterval(id));
});
```

7. Prefer `input()` over `@Input()` for Reactive Components (Angular 17+)

- Native signal-based input binding.

```
name = input<string>('Guest');
```

Common Pitfalls

1. Forgetting `()` When Accessing Signals

- `{{ count }}` → Wrong
- `{{ count() }}` → Correct

2. Mutating Signals Directly

- Never do `this.count = 5`
- Always use `set()` or `update()`

```
this.count.set(5);
```

```
this.count.update(n => n + 1);
```

3. Using `effect()` to Change Signals

- Causes infinite loops and violates purity.

```
//  Avoid
effect(() => {
  this.count.set(this.count() + 1);
});
```

4. Using Signals in Complex Async Flows Without `toObservable()`

- Signals are synchronous. For async tasks like HTTP, convert to/from observable:

```
toSignal(fromObservable(...))
```

5. Overusing Signals for Everything

- Not everything needs to be reactive.
- Use signals for **reactive UI state**, not for storing config, constants, or rarely-changing values.

6. Using Signals Without Cleanup in Long-lived Effects

- Can cause memory leaks if intervals or subscriptions aren't cleared.

Summary Table

Do	Don't
Use `signal()` for reactive state	Mutate signals directly (`signal = 5`)
Use `computed()` for derived values	Change signals inside `effect()`

Use `untracked()` when skipping dependencies	Forget `()` when accessing signal in template
Use `cleanup()` for side-effect teardown	Leave intervals/subscriptions hanging
Use `input()` for reactive input bindings	Combine with `@Input()` unnecessarily

Angular 17–19 Features

Focus: Standalone Components & Routing

https://angular.dev/guide/components

Angular 17–19 introduced major improvements in **application architecture, performance**, and **developer ergonomics**, especially with **standalone APIs**.

What are Standalone Components?

Definition:

Standalone components are **self-contained Angular components** that **do not require a module (NgModule)** to function.

```
@Component({
  standalone: true,
  selector: 'app-hello',
  template: `<p>Hello!</p>`,
})
export class HelloComponent {}
```

Benefits of Standalone Components

Benefit	Description
No NgModules	Simpler structure, easier onboarding
Better tree-shaking	Smaller bundle size
Faster app startup	Less indirection and DI setup
Native route-level code-splitting	Works seamlessly with standalone routing

Easier code reuse	Easily imported in other components/services

Declaring and Using Standalone Components

```
@Component({
  standalone: true,
  imports: [CommonModule],
  selector: 'app-hello',
  template: `<p>Hello!</p>`,
})
export class HelloComponent {}
```

Used in another component:

```
@Component({
  standalone: true,
  imports: [HelloComponent],
  template: `<app-hello></app-hello>`,
})
export class ParentComponent {}
```

Standalone Routing (Angular 14+ and improved in 17–19)

Standalone routing allows you to **define routes directly using standalone components**, **without NgModules**.

App Route Setup (main.ts or app.config.ts)

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter, Routes } from '@angular/router';
import { AppComponent } from './app/app.component';
import { HomeComponent } from './app/home.component';

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

bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)],
});
```

Standalone Component with Routing

```
@Component({
  standalone: true,
  selector: 'app-home',
  template: `<h1>Home</h1>`,
})
export class HomeComponent {}
```

Angular 17–19 Routing Improvements

Feature	Description
Declarative routing	`provideRouter()` setup in `main.ts`, not NgModules
Page-based routing (Angular 17+)	Auto-routes from `/app/pages` directory (`routes.ts`)
Deferred Loading (17+)	Load routes after interaction using `defer: true`
Signal Inputs + routing	Combine signals and route params easily

Best Practices with Standalone Components

- Prefer `standalone: true` for new components
- Group feature routes using `loadComponent`
- Use `importProvidersFrom()` if you must reuse an NgModule temporarily

- Use `inject()` for cleaner DI in standalone context

Summary Table

Concept	Traditional Angular	Angular 17–19 (Standalone)
Component Setup	Requires `NgModule`	Just `standalone: true`
Route Config	In `AppRoutingModule`	In `main.ts` with `provideRouter()`
Route Components	Must be declared in module	Can be fully standalone
Performance	Slower startup	Smaller, faster, zone-less ready
Use Cases	Enterprise apps	Modern apps, microfrontends, SSR

Lab: Standalone Components & Routing in Angular 17+

Objective

By the end of this lab, you will:

- Create and use a **standalone component**
- Configure **standalone routing** using `provideRouter()`
- Load components using the modern `loadComponent` API
- Understand how Angular apps can be built without `NgModules`

Step 1: Create a New Angular App with Standalone API

```
ng new standalone-lab --standalone --routing=false --style=css
cd standalone-lab
```

Use `--standalone` to scaffold a project without `AppModule`.

Step 2: Create a Home Component

```
ng generate component pages/home --standalone
```

This creates a fully standalone component.

Step 3: Create an About Component

```
ng generate component pages/about --standalone
```

Step 4: Define Standalone Routes

Create a file: `src/app/routes.ts`

```
import { Routes } from '@angular/router';

export const routes: Routes = [
  {
    path: '',
    loadComponent: () =>
      import('./pages/home/home.component').then(m => m.HomeComponent)
  },
  {
    path: 'about',
    loadComponent: () =>
      import('./pages/about/about.component').then(m => m.AboutComponent)
  }
];
```

Step 5: Configure Routing in `main.ts`

Update `main.ts`:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { AppComponent } from './app/app.component';
import { routes } from './app/routes';

bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)]
});
```

Step 6: Modify `AppComponent` to Use `<router-outlet>`

Update `app.component.ts`:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `
    <nav>
      <a routerLink="/">Home</a> |
```

```
    <a routerLink="/about">About</a>
  </nav>
  <hr />
  <router-outlet></router-outlet>
 `
})
export class AppComponent {}
```

Step 7: Run and Test

```
ng serve
```

What to Try:

- Navigate to `/` and `/about`
- Check browser dev tools → Each route loads its component on demand
- Inspect project structure — no `AppModule` used

Summary of Concepts Practiced

Concept	Technique Used
Standalone component	`--standalone` flag in Angular CLI
Routing config	Defined in `routes.ts` using `loadComponent()`
App setup	`bootstrapApplication()` and `provideRouter()`

Component rendering	`<router-outlet>` inside a standalone root

Bonus Challenges

- Add a `NotFoundComponent` for unknown routes
- Add route data (e.g., titles) and display in each component
- Try `defer: true` in route config to experiment with **deferred loading**

Angular Control Flow Syntax

New Block Syntax: @if, @for, @switch (Angular 17+)

Angular 17 introduced **declarative control flow blocks** with a new syntax that improves:

- Readability
- IDE support
- Template type safety
- Performance (via better compile-time optimization)

1. `@if` – Conditional Rendering

Syntax:

```
@if (condition) {
  <p>Condition is true</p>
} @else {
  <p>Condition is false</p>
}
```

Example:

```
@if (isLoggedIn()) {
  <p>Welcome, {{ userName() }}</p>
} @else {
  <button (click)="login()">Login</button>
}
```

Benefits over `*ngIf`:

- Cleaner nesting
- No need for `<ng-container>`
- Fully typed inside blocks

2. `@for` – Iteration Block (Replaces `*ngFor`)

Syntax:

```
@for (item of items; track item.id) {
  <p>{{ item.name }}</p>
}
```

Example:

```
@for (task of tasks(); track task.id) {
  <li>{{ task.title }}</li>
}
```

Destructuring & Index:

```
@for ((task, i) of tasks()) {
  <li>#{{ i }} — {{ task.title }}</li>
}
```

Benefits over `*ngFor`:

- Track by is **built in**
- Index and key access is easier
- Better autocomplete and type inference

3. `@switch` – Conditional Matching (Alternative to `ngSwitch`)

Syntax:

```
@switch (status) {
  @case ('loading') {
    <p>Loading...</p>
  }
  @case ('error') {
    <p>Error occurred!</p>
  }
  @default {
    <p>Unknown state</p>
  }
}
```

Why Use These?

Feature	Benefit
`@if`	Cleaner and fully-typed conditional rendering
`@for`	Built-in `trackBy`, destructuring, better performance
`@switch`	Simpler branching logic compared to `ngSwitch`

Compatibility & Requirements

- Available in **Angular 17+**
- Works only in **standalone-enabled templates**
- Requires `@angular/compiler` and Angular CLI 17+ or above

Migration Example

Old (ngIf + ngFor):

```
<ng-container *ngIf="items.length > 0; else empty">
  <div *ngFor="let item of items">{{ item.name }}</div>
</ng-container>
<ng-template #empty>No items</ng-template>
```

New (Angular 17+):

```
@if (items.length > 0) {
  @for (item of items) {
    <div>{{ item.name }}</div>
  }
} @else {
  <p>No items</p>
}
```

Comparison Table: New Block Syntax vs Traditional Directives

Feature	`@if` / `@for` / `@switch` (Angular 17+)	`*ngIf` / `*ngFor` / `ngSwitch` (Traditional)
Syntax Clarity	Cleaner, block-style, inline `@else`	Verbose, needs `<ng-container>` and `<ng-template>`
Type Safety	Fully typed in template (e.g., `task.title`)	Limited type inference; often needs `as` cast
IDE Support	Better autocomplete, inline errors	Harder for IDEs to infer context types
Performance	Faster runtime (compiled to `InstructionBlock`)	Slightly slower; uses runtime directive creation
TrackBy	Built-in to `@for`, no extra syntax	Requires verbose `trackBy: trackByFn`
Switching Logic	`@switch`/`@case` block-style, scoped vars	`<ng-container *ngSwitch>` + nested `*ngSwitchCase`
Destructuring	Supported: `@for ((item, i) of items)`	Supported but less readable in HTML

Migration	Simple (same logic, new syntax)	Already stable, backward compatible
Standalone Required	Yes (works only in standalone mode)	Works in all templates
Learning Curve	Slightly new syntax to learn	Familiar for existing Angular developers

Performance Insight

Metric	Angular 17+ Block Syntax	Traditional Directives
Template Compilation	Compiled to instruction blocks (faster)	Translates to directives at runtime
Change Detection	More efficient (fewer checks)	Slightly more overhead
DOM Update Granularity	More optimal (compiled trees)	More diffing and patching

Verdict: Angular 17+ block syntax allows **better template optimization during compile time**, leading to **faster rendering and smaller change detection scope**.

Real-World Example: Performance-Friendly List with Signal

```
tasks = signal([
  { id: 1, title: 'Learn Signals' },
  { id: 2, title: 'Refactor Template' }
]);
```

```
<!-- Angular 17+ -->
@for (task of tasks(); track task.id) {
  <li>{{ task.title }}</li>
}
```

- No need for extra `trackBy` function — Angular compiles `track task.id` directly for efficient DOM updates.

Summary Table

Directive	Purpose	Key Feature
`@if`	Conditional block	Cleaner than `*ngIf`, inline `@else`
`@for`	Iteration block	Replaces `*ngFor`, built-in `trackBy`
`@switch`	Branching logic	Simpler than `ngSwitch`

When to Use	**Use `@if`, `@for`, `@switch` When...**
New Projects	You're using Angular 17+ and standalone components
Signal-based Apps	Works best with `signal()` and `computed()`
Performance Focus	You want faster DOM updates and smaller compiled templates
Readability First	Cleaner syntax and easier to read templates

Lab: Mastering Angular 17+ Control Flow Syntax

Objective

By the end of this lab, you will:

- Use `@if` to conditionally render content
- Use `@for` to iterate through arrays with `track`
- Use `@switch`, `@case`, and `@default` for conditional branching
- Compare the new syntax to the traditional `*ngIf` and `*ngFor`

Step 1: Create a New Standalone Angular Project

```
ng new control-flow-lab --standalone --routing=false --style=css
cd control-flow-lab
```

Step 2: Generate a Demo Component

```
ng generate component control-flow-demo --standalone
```

Update `app.component.html`:

```
<app-control-flow-demo></app-control-flow-demo>
```

Step 3: Setup Demo State in the Component

control-flow-demo.component.ts

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

@Component({
  selector: 'app-control-flow-demo',
  standalone: true,
  templateUrl: './control-flow-demo.component.html'
})
```

```
export class ControlFlowDemoComponent {
  isLoggedIn = signal(false);
  userName = signal('Aisyah');

  toggleLogin() {
    this.isLoggedIn.update(v => !v);
  }

  tasks = signal([
    { id: 1, title: 'Buy milk' },
    { id: 2, title: 'Study Angular' },
    { id: 3, title: 'Call mom' }
  ]);

  status = signal<'loading' | 'error' | 'done'>('loading');
}
```

Step 4: Add Template with `@if`

control-flow-demo.component.html

```
<h2>if Example</h2>

@if (isLoggedIn()) {
  <p>Welcome, {{ userName() }}!</p>
  <button (click)="toggleLogin()">Logout</button>
} @else {
  <p>You are not logged in.</p>
  <button (click)="toggleLogin()">Login</button>
}
```

Try toggling the login state and see content change.

Step 5: Add `@for` Loop

```
<h2>for Example</h2>

<ul>
  @for (task of tasks(); track task.id) {
    <li>{{ task.title }}</li>
  }
</ul>
```

Try reordering or modifying `tasks` signal in the code to see `trackBy` in action.

Step 6: Add `@switch` Block

```
<h2>switch Example</h2>

@switch (status()) {
  @case ('loading') {
    <p> Loading...</p>
  }
  @case ('error') {
    <p> Error loading data</p>
  }
  @case ('done') {
    <p>Finished loading!</p>
  }
  @default {
    <p>Unknown status</p>
  }
}

<!-- Manually change the status value in TypeScript for testing -->
```

You can change `status.set('done')` manually in code to see different outputs.

Step 7: Run the App

```
ng serve
```

Navigate to `http://localhost:4200` and test:

- The `@if` login toggle
- The `@for` list rendering
- The `@switch` branching

Summary of Concepts Practiced

Feature	Syntax	Replaces
@if	@if (...) {}	*ngIf, <ng-container>
@for	@for (...) {}	*ngFor + trackBy
@switch	@switch, @case	ngSwitch

Bonus Challenges

- Add a `@for ((task, i) of tasks())` to display index
- Add a button to switch `status` from `loading` → `done` → `error`
- Convert one of the `@if` blocks into the old `*ngIf` syntax and compare readability

Bonus 1: Add `@for ((task, i) of tasks())` to Display Index

Update your `control-flow-demo.component.html`:

```
<h2>for Example with Index</h2>
<ul>
  @for ((task, i) of tasks(); track task.id) {
    <li>#{{ i + 1 }} — {{ task.title }}</li>
  }
</ul>
```

- `i + 1` is used for 1-based indexing (optional, but cleaner for display).

Bonus 2: Add Button to Cycle `status` from `loading` → `done` → `error` → `loading`

Update your [`control-flow-demo.component.ts`](control-flow-demo.component.ts):

```
cycleStatus() {
  const current = this.status();
  if (current === 'loading') this.status.set('done');
  else if (current === 'done') this.status.set('error');
  else this.status.set('loading');
}
```

Update `control-flow-demo.component.html`:

```
<h2>switch Example</h2>
<button (click)="cycleStatus()">Cycle Status</button>

@switch (status()) {
  @case ('loading') {
    <p> Loading...</p>
  }
```

```
  @case ('error') {
    <p>Error loading data</p>
  }
  @case ('done') {
    <p> Finished loading!</p>
  }
  @default {
    <p>❓ Unknown status</p>
  }
}
```

Now clicking the button cycles through each state.

Bonus 3: Convert `@if` block into classic `*ngIf` and compare

Add Classic `*ngIf` Version Below for Comparison

In `control-flow-demo.component.html`:

```
<h2>*ngIf Example (Legacy)</h2>
<ng-container *ngIf="isLoggedIn(); else loggedOut">
  <p>Welcome, {{ userName() }}!</p>
  <button (click)="toggleLogin()">Logout</button>
</ng-container>
<ng-template #loggedOut>
  <p>You are not logged in.</p>
  <button (click)="toggleLogin()">Login</button>
</ng-template>
```

Visual Comparison:

Feature	`@if` Block	`*ngIf` Traditional
Readability	Clear, inline	Split between `ng-container` and `ng-template`
Maintainability	Easier to manage logic flow	Requires two separate tags

Type Safety	Fully typed in block	Not explicitly enforced

Final App Testing Checklist

- Toggle login state using both `@if` and `*ngIf`
- Render tasks with index using `@for ((task, i) of ...)`
- Cycle through status states using `@switch`
- Visually compare legacy syntax to new block syntax

Zoneless Angular: What, Why, and How

- https://angular.dev/guide/zoneless
- https://angularexperts.io/blog/zoneless-angular/

What is Zoneless Angular?

Zoneless Angular refers to running Angular **without Zone.js**, a library that monkey-patches browser APIs (like `setTimeout`, `addEventListener`, `Promise.then`, etc.) so Angular knows when to trigger **global change detection**.

Traditional (Zone.js-enabled)

- Angular automatically runs change detection after *any* async operation.
- Easy but **expensive**, as the entire component tree is checked.

Zoneless Angular

- Removes automatic global change detection
- No monkey-patching = cleaner stack traces and better performance
- You must **explicitly control when CD happens**
  - Preferably using **Signals**
  - Or manually via `ChangeDetectorRef.detectChanges()`

Why Go Zoneless?

Benefit	Description
Performance Boost	Avoids CD triggering on every async task — **huge win** in large apps

Predictable Updates	Only re-renders what you explicitly change — **deterministic UI updates**
Cleaner Stack Traces	No more Zone.js wrapping and confusing async call stacks
Paired with Signals	Signals eliminate the need for `ngOnChanges`, `detectChanges`, and boilerplate
SSR-friendly	Helps fine-tune hydration and lazy rendering on the server

How to Use Angular Without Zone.js

Step 1: Remove Zone.js

In `main.ts`:

```
// Remove or comment this:
import 'zone.js';
```

If using `bootstrapApplication`, use:

```
import { ApplicationConfig, provideExperimentalZonelessChangeDetection } from
'@angular/core';

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

Step 2: Use Signals or ChangeDetectorRef

Since Angular no longer auto-triggers CD, **you have 2 options**:

Option 1: Use Signals (recommended)

```
@Component({
```

```
  standalone: true,
  selector: 'app-clock',
  template: `<p>Time: {{ time() }}</p>`,
})
export class ClockComponent {
  time = signal(new Date().toLocaleTimeString());

  constructor() {
    setInterval(() => {
      this.time.set(new Date().toLocaleTimeString()); // triggers reactivity
    }, 1000);
  }
}
```

- Angular tracks signal dependencies in templates
- When `time()` changes, **only that binding updates**
- No need for zones, `detectChanges`, or `markForCheck`

Option 2: Use `ChangeDetectorRef` Manually

```
constructor(private cd: ChangeDetectorRef) {
  setTimeout(() => {
    this.data = 'Updated!';
    this.cd.detectChanges(); // trigger CD manually
  }, 1000);
}
```

- Needed if you're using plain variables or working with legacy patterns
- You must **decide where and when to run CD**

How Change Detection Works in Zoneless Angular

Case	Triggers CD?	Explanation
`signal().update()`	Yes	Signals are tracked reactively
`(click)="..."`	Yes	Angular wires event bindings manually (even zoneless)
`setTimeout(() => ...)`	No	No auto-CD unless using signals or `detectChanges()`
`Promise.then()`	No	No auto-CD unless signal is used
`nativeElement.addEventListener()`	No	Outside Angular; use signal or `detectChanges()`
`Observable.subscribe()`	No	Manual CD or signal needed
`ChangeDetectorRef.detectChanges()`	Yes	Triggers CD manually at the component level

Pitfalls and Considerations

Issue	Explanation
Manual Work	You are responsible for triggering CD (no auto-magic)

Complex Legacy Code	Old apps might rely on Zone-based behavior deeply
Tests May Break	Unit/integration tests may rely on Zone-based async behavior
Requires Signal Mindset	You must think in terms of **reactive state + explicit triggers**

When to Use Zoneless Angular

Recommended for:

- High-performance dashboards or animation-heavy UIs
- Component libraries (signals = clean inputs/outputs)
- Apps that already use `OnPush` and immutable state
- SSR (server-side rendering) apps — improves hydration control
- Micro frontends or islands of interactivity

Not ideal for:

- Large legacy applications with deep reliance on `ngZone`
- Teams unfamiliar with signals, reactivity, or manual CD
- Codebases mixing `ngModel`, template-driven forms, etc.

Summary

Feature	With Zone.js	Zoneless Angular

Change detection	Automatic (global tree check)	Manual, per-signal/component
Monkey-patching	Yes (`setTimeout`, etc.)	None
View updates	On every async trigger	Only on signal/update/click
Performance	Slower in large apps	Faster + precise
Debuggability	Stack traces polluted	Clean, async-safe
Recommended use cases	Legacy, DX convenience	Modern apps with signals

Lab: Angular Without Zone.js — Comparing Reactivity Sources with and without Signals

Objective

By the end of this lab, you will:

- Compare plain variables vs. signals
- Use different triggers: `setTimeout`, `setInterval`, `Promise.then`, and `addEventListener`

Observe DOM updates in **Zone.js**, **Zoneless + plain variable**, and **Zoneless + Signals**
Understand when Angular re-renders and when it doesn't

Step 1: Create a New Angular App

```
ng new reactivity-comparison-lab --standalone --style=css --routing=false
cd reactivity-comparison-lab
```

Step 2: Remove Zone.js & Enable Zoneless Mode

```
npm uninstall zone.js
```

main.ts — remove `import 'zone.js'`

app.config.ts:

```
import { ApplicationConfig, provideExperimentalZonelessChangeDetection } from
'@angular/core';

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

main.ts:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { appConfig } from './app/app.config';

bootstrapApplication(AppComponent, appConfig);
```

Step 3: Create Component to Compare All Scenarios

`app.component.ts`:

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
})
export class AppComponent {

// Plain variable (non-reactive)
  plainCount = 0;

  // Signal variable (reactive)
  signalCount = signal(0);

  constructor() {
    // setTimeout
    setTimeout(() => {
      this.plainCount++;
      this.signalCount.update(v => v + 1);
      console.log('setTimeout - plain:', this.plainCount, '| signal:', this.signalCount());
    }, 1000);

    // setInterval
    setInterval(() => {
      this.plainCount++;
      this.signalCount.update(v => v + 1);
      console.log('setInterval - plain:', this.plainCount, '| signal:', this.signalCount());
    }, 2000);

    // Promise
    Promise.resolve().then(() => {
      this.plainCount++;
```

```
      this.signalCount.update(v => v + 1);
      console.log('Promise - plain:', this.plainCount, '| signal:', this.signalCount());
    });

    // native DOM event
    setTimeout(() => {
      const btn = document.getElementById('native-button');
      if (btn) {
        btn.addEventListener('click', () => {
          this.plainCount++;
          this.signalCount.update(v => v + 1);
          console.log('Native click - plain:', this.plainCount, '| signal:', this.signalCount());
        });
      }
    });

    // Observe reactivity
    effect(() => {
      console.log('Signal changed to:', this.signalCount());
    });
  }

  getPlainCount() {
    return this.plainCount; // use method to avoid caching
  }

  incrementBoth() {
    this.plainCount++;
    this.signalCount.update(v => v + 1);
    console.log('Manual click - plain:', this.plainCount, '| signal:', this.signalCount());
  }
}
```

Step 4: Template for Visual Comparison

`App.component.html`:

```
<h1>Angular Zoneless Reactivity Comparison</h1>

<section>
  <h2>Manual Click</h2>
  <p>Plain Count: {{ getPlainCount() }}</p>
  <p>Signal Count: {{ signalCount() }}</p>
  <button (click)="incrementBoth()">Increment Manually</button>
</section>
```

```
<hr />

<section>
  <h2>setTimeout, setInterval, Promise</h2>
  <p>Observe console logs for async triggers.</p>
</section>

<hr />

<section>
  <h2>Native DOM Event</h2>
  <button id="native-button">Native Event: Increment</button>
  <p>Plain Count (Native): {{ getPlainCount() }}</p>
  <p>Signal Count (Native): {{ signalCount() }}</p>
</section>
```

Step 5: Run the App

```
npm start
```

What to Observe

Trigger	Plain Variable	Signal	DOM Updates?
`(click)="..."`	Yes	Yes	Both
`setTimeout`	No	Yes	Only signal
`setInterval`	No	Yes	Only signal

`Promise.then()`	No	Yes	Only signal
Native Event	No	Yes	Only signal

Use DevTools and console logs to inspect the behavior.

Bonus: Want to Show with `detectChanges()`?

Add to constructor:

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

constructor(private cdr: ChangeDetectorRef) {
  setTimeout(() => {
    this.plainCount++;
    this.cdr.detectChanges(); // manually force update
  }, 3000);
}
```

Now you'll see DOM update even for plain variables — but only when you explicitly ask Angular to re-render.

Summary

Method	Signals Needed?	DOM Update?	Manual Fix?
`setTimeout`	Yes	No / Yes	Use signal or detectChanges
`Promise`	Yes	No / Yes	Use signal
Native events	Yes	No / Yes	Use signal or detectChanges

`(click)`	No (still works)	Yes	Angular wires it manually

Source code: https://github.com/wanmuz86/angular-int-adv-lab12-reactivity-comparison-lab

Zoneless Change Detection with Signals

Objective

By the end of this lab, you will:

- Understand how Angular works without Zone.js
- Use `signal()`, `computed()`, `effect()` properly
- Bootstrap your Angular app in **zoneless mode**
- Replace traditional `@Input()` + Zone-based CD with Signals-based reactivity
- Connect Signals to async HTTP with `effect()` and **`toSignal()`**

Step 1: Create a New Angular 20 App

```
ng new zoneless-lab --standalone --style=css
cd zoneless-lab
```

- Choose `Yes` when prompted to use zoneless

Step 2: Verify that zone.js is removed (Angular 20) or remove it (Angular 17-19)

Uninstall zone.js

```
npm uninstall zone.js
```

Step 3: Verify Zoneless Mode in **app.config.ts** (Angular 20) or change to zoneless mode (Angular 17-19)

```
import { ApplicationConfig, provideExperimentalZonelessChangeDetection   } from
'@angular/core';

export const appConfig: ApplicationConfig = {
 providers: [provideExperimentalZonelessChangeDetection()]
};
```

Step 4: Create a Signal-based Service

```
ng g service stats
```

```
// src/app/stats.service.ts
import { Injectable, signal } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class StatsService {
  value = signal(0);

  update() {
    setTimeout(() => {
      this.value.set(42); // only the dependent component will update
    }, 1000);
  }
}
```

Step 5: Create a Standalone Signal Component

```
ng generate component stats --standalone
```

Edit `stats.component.ts`:

```
import { Component, computed, effect, inject, signal } from '@angular/core';
import { StatsService } from '../stats.service';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-stats',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>Stats Value: {{ value() }}</p>
    <p>Double: {{ doubleValue() }}</p>
    <button (click)="update()">Update</button>
  `
})
export class Stats {
  private statsService = inject(StatsService);
```

```
  value = this.statsService.value;
  doubleValue = computed(() => this.value() * 2);

  constructor() {
    effect(() => {
      console.log('Stats changed:', this.value());
    });
  }

  update() {
    this.statsService.update();
  }
}
```

Step 6: Use it in AppComponent

```
// app.component.ts
import { Component } from '@angular/core';
import { StatsComponent } from './stats/stats.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [StatsComponent],
  template: `
    <h1>Zoneless Angular 20 </h1>
    <app-stats></app-stats>
  `
})
export class AppComponent {}
```

Step 7: Use `effect()` with HTTP

```
ng generate component user-profile --standalone
```

[app.config.ts]()

```
import { ApplicationConfig, provideExperimentalZonelessChangeDetection   } from
'@angular/core';
import { provideHttpClient } from '@angular/common/http';
export const appConfig: ApplicationConfig = {
 providers: [provideExperimentalZonelessChangeDetection(), provideHttpClient()]
```

```
};
```

user-profie.component.ts

```
import { Component, effect, inject, signal } from '@angular/core';
import { HttpClient } from '@angular/common/http';



@Component({
 selector: 'app-user-profile',
 imports: [],
 templateUrl: './user-profile.component.html',
 styleUrl: './user-profile.component.css'
})
export class UserProfileComponent {

 private http = inject(HttpClient);
 user = signal<any | null>(null);

 constructor() {
   effect(() => {
     this.http.get('https://fakestoreapi.com/users/1').subscribe((data) => {
       this.user.set(data);
     });
   });
 }

}
```

```
<div *ngIf="user() as userData; else loading">
 <div class="profile-card">
   <h2>{{ userData.name.firstname }} {{ userData.name.lastname }}</h2>
   <p><strong>Email:</strong> {{ userData.email }}</p>
   <p><strong>Username:</strong> {{ userData.username }}</p>
   <p><strong>Phone:</strong> {{ userData.phone }}</p>
   <p><strong>Address:</strong>
     {{ userData.address.number }} {{ userData.address.street }},
     {{ userData.address.city }}, {{ userData.address.zipcode }}
   </p>
```

```
 </div>
</div>
<ng-template #loading>
 <div>Loading user profile...</div>
</ng-template>
```

Step 8: Use `toSignal()` with RxJS

```
ng generate component todos --standalone
```

```
import { Component, inject } from '@angular/core';
import { toSignal } from '@angular/core/rxjs-interop';
import { HttpClient } from '@angular/common/http';
import { CommonModule } from '@angular/common';

@Component({
  standalone: true,
  selector: 'products',
  imports: [CommonModule],
  template: `
    <ul *ngIf="products(); else loading">
      <li *ngFor="let product of products()">{{ product.title }}</li>
    </ul>
    <ng-template #loading>Loading...</ng-template>
  `,
})
export class ProductsComponent {
  private http = inject(HttpClient);
  products = toSignal(this.http.get<any[]>('https://fakestoreapi.com/products'), {
    initialValue: [],
  });
}
```

Checklist for Zoneless + Signals

Tip	Example

Use `signal()` for state	`count = signal(0)`
Use `computed()` for derived	`double = computed(() => count() * 2)`
Use `effect()` for reaction	`effect(() => { console.log(count()); })`
Use `markDirty()` for manual	`markDirty(this)` when using non-signal props
Use `toSignal()` for RxJS	`toSignal(http.get(...))`
Test all async integrations	Some libraries may still depend on Zone.js

Summary

- You've successfully disabled Zone.js and replaced it with Signals and fine-grained change detection.
- Components now only update when their reactive signals change.
- Rendering is more **predictable, faster, and testable**.
- **Zoneless Angular** is the future of reactive UI in Angular.

Signal Reactivity Enhancements in Angular 19

Angular 19 continues to refine the **Signals API**, making Angular's reactivity model more robust, composable, and efficient.

Key Enhancements in Angular 19

1. `settable()` Signals (Improved APIs for encapsulation)

Angular 19 introduces a cleaner approach to **encapsulated signal state**, removing the need to manually expose `asReadonly()` in many cases.

```
import { signal, settable } from '@angular/core';

const count = settable(0); // replaces signal() + asReadonly pattern
```

Benefit:

Cleaner API to define a signal that can be updated internally but shared externally as readonly.

2. Dependency-aware `effect()` Optimization

Angular 19 improves how `effect()` reacts to dependencies:

- **Batched updates**: multiple signal changes trigger only **one re-evaluation**
- **Skipped runs**: effects are skipped unless dependencies **actually changed**

```
effect(() => {
  console.log(counter()); // only runs when counter changes
});
```

Benefit:

Better **performance and predictability** — especially in large UIs.

3. Computed Signal Enhancements

Smarter memoization:

- `computed()` will **not re-run** unless one of its **actual dependencies** has changed
- Prevents wasteful recalculations in long signal chains

```
const fullName = computed(() => `${firstName()} ${lastName()}`);
```

4. Improved Debugging & DevTools Integration

Angular 19 introduces:

- **Signal Graph Inspection** (via Angular DevTools)
- **Named signals and effects** for easier debugging:

```
const count = signal(0, { name: 'countSignal' });
```

Benefit:

Trace signal changes and reactive flows visually.

5. Lifecycle Hooks in `effect()` with `cleanup()`

Fully supported and **deterministic** cleanup behavior:

```
effect(() => {
  const id = setInterval(...);
  cleanup(() => clearInterval(id));
});
```

Benefit:

More consistent memory management in reactive flows.

6. Future-facing APIs

Angular 19 paves the way for:

- More **zoneless** integration (e.g., SSR, event handlers)
- **Signal inputs**, **signal outputs**, and **control flow blocks** working together smoothly
- **Signal-based form controls and router integration** in upcoming versions

Summary Table

Feature	Angular 18	Angular 19 Enhancement
Signal creation	`signal()`	`settable()` (better encapsulation)
Computed signal reactivity	Manual memoization	Smarter, dependency-aware re-runs
Effects	Immediate, unbatched	Batched, optimized, skippable
DevTools	Limited support	Full signal graph + naming support
Lifecycle (`cleanup()`)	Experimental	Fully integrated

Best Practices in Angular 19

- Use `settable()` for better encapsulated signal states
- Name your signals and effects for better DevTools tracing
- Avoid nesting signals inside other signals (use `computed()` properly)
- Use `cleanup()` in long-running effects or subscriptions
- Let the **reactivity model** drive your UI — no need for `ngOnChanges` or `EventEmitters` in most cases

Lab: Exploring Signal Reactivity Enhancements in Angular 19

Objectives

By the end of this lab, you will:

- Use `settable()` to define encapsulated reactive state
- Create `computed()` signals with optimized dependencies
- Track and log updates using `effect()` with `cleanup()`
- Inspect signal graph with Angular DevTools
- Build a small reactive counter dashboard

Step 1: Create a New Angular App (Standalone)

```
ng new signal-enhanced-lab --standalone --style=css --routing=false
cd signal-enhanced-lab
```

Step 2: Create a Component

```
ng generate component counter-dashboard --standalone
```

Update `app.component.ts`:

```
import { Component } from '@angular/core';
import { CounterDashboardComponent } from
'./counter-dashboard/counter-dashboard.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CounterDashboardComponent],
  template: `<app-counter-dashboard />`,
})
export class AppComponent {}
```

Step 3: Create Reactive State Using `settable()`

In `counter-dashboard.component.ts`:

```
import { Component, computed, effect, settable, signal } from '@angular/core';
import { cleanup } from '@angular/core/signals';

@Component({
  selector: 'app-counter-dashboard',
  standalone: true,
  templateUrl: './counter-dashboard.component.html',
})
export class CounterDashboardComponent {
  counter = settable(0, { name: 'counter' });

  increment = () => this.counter.update(c => c + 1);
  decrement = () => this.counter.update(c => c - 1);
  reset = () => this.counter.set(0);

  double = computed(() => this.counter() * 2, { name: 'doubleValue' });
  status = computed(() => this.counter() % 2 === 0 ? 'Even' : 'Odd');

  constructor() {
    effect(() => {
      console.log(`Count changed: ${this.counter()} (double: ${this.double()})`);
    }, { name: 'counterLogger' });

    effect(() => {
      const interval = setInterval(() => this.increment(), 2000);
      cleanup(() => clearInterval(interval));
    });
  }
}
```

Step 4: Build the Template

In `counter-dashboard.component.html`:

```
<h2> Signal Counter (Angular 19)</h2>

<p>Value: {{ counter() }}</p>
<p>Double: {{ double() }}</p>
<p>Status: <strong>{{ status() }}</strong></p>

<button (click)="increment()"> Increment</button>
<button (click)="decrement()"> Decrement</button>
<button (click)="reset()">Reset</button>
```

Step 5: Run the App

```
ng serve
```

Open http://localhost:4200

You should see:

- A counter that auto-increments every 2 seconds
- Console logs from `effect()`
- Buttons to interactively change state
- All UI reactivity working **without Zone.js**

Step 6: Use Angular DevTools

1. Open **Angular DevTools** in your browser
2. Go to the **Signals tab**
3. Inspect:
   - `counter` signal
   - `double` and `status` computed signals
   - `counterLogger` effect
   - Signal graph structure

Summary of Concepts Practiced

Feature	Usage Example

`settable()`	`counter = settable(0)`
`computed()`	`double = computed(() => counter() * 2)`
`effect()`	Logs and updates triggered by signals
`cleanup()`	Clears interval inside `effect()`
DevTools signal tracing	Named signals and effects for inspection

Bonus Challenges

- Add another signal for user-defined step size (e.g., increment by N)
- Use `untracked()` to log the value without triggering dependencies
- Add a signal-based timer countdown using `settable()` and `effect()`

Upgrade Angular App to Signals and Native Signal Inputs

Objectives

By the end of this lab, you will:

- Replace RxJS with Signals in a state service
- Use `computed()` and `effect()` for reactive logic
- Pass signals natively using `@Input({ signal: true })`
- Understand smart vs dumb components
- Handle async API calls with `toSignal()`

Enable Zoneless Mode

Step 1: Setup Project

```
ng new signals-upgrade-lab --standalone --style=css --routing=false
cd signals-upgrade-lab
```

Step 0: Enable Zoneless Change Detection (Before Step 1)

1. **Uninstall `zone.js`:**

```
npm uninstall zone.js
```

2. **Update `main.ts`** — no changes needed if you're already using Angular 17+ standalone bootstrap.

3. **Modify `app.config.ts`**:

Create or update `src/app/app.config.ts`:

```
import { ApplicationConfig, provideExperimentalZonelessChangeDetection } from
'@angular/core';

export const appConfig: ApplicationConfig = {
```

```
  providers: [provideExperimentalZonelessChangeDetection()],
};
```

Step 2: Create State Service Using Signals

```
ng generate service services/product-state
```

Update `product-state.service.ts`:

```
import { Injectable, signal, computed, effect } from '@angular/core';
import { toSignal } from '@angular/core/rxjs-interop';
import { HttpClient } from '@angular/common/http';
import { inject } from '@angular/core';

export interface Product {
  id: number;
  title: string;
  price: number;
}

@Injectable({ providedIn: 'root' })
export class ProductStateService {
  private http = inject(HttpClient);

  private rawProducts$ = this.http.get<Product[]>('https://fakestoreapi.com/products');

  private products = toSignal(this.rawProducts$, { initialValue: [] });
  readonly productList = this.products;

  selectedId = signal<number | null>(null);

  readonly selectedProduct = computed(() =>
    this.products()!.find(p => p.id === this.selectedId())
  );

  constructor() {
    effect(() => {
      console.log('Selected Product:', this.selectedProduct());
    });
  }

  select(id: number) {
    this.selectedId.set(id);
```

```
  }
}
```

Step 3: Create Components

```
ng generate component product-dashboard --standalone
ng generate component product-detail --standalone
```

Step 4: Implement Product Dashboard (Smart Component)

product-dashboard.component.ts

```
import { Component } from '@angular/core';
import { ProductStateService } from '../services/product-state.service';
import { ProductDetailComponent } from '../product-detail/product-detail.component';

@Component({
  selector: 'app-product-dashboard',
  standalone: true,
  imports: [ ProductDetailComponent],
  templateUrl: './product-dashboard.component.html'
})
export class ProductDashboardComponent {
  constructor(public productState: ProductStateService) {}

  select(id: number) {
    this.productState.select(id);
  }
}
```

product-dashboard.component.html

```
<h2>Product List</h2>
<ul>
  @for (product of productState.productList(); track product.id) {
    <li>
      {{ product.title }} - ${{ product.price }}
      <button (click)="select(product.id)">View</button>
    </li>
  }
</ul>
```

```
<hr />

<app-product-detail [product]="productState.selectedProduct" />
```

Step 5: Use `@Input({ signal: true })` in Child (Dumb Component)

product-detail.component.ts

```
import { Component, Signal, input } from '@angular/core';
import { Product } from '../services/product-state.service';
@Component({
 selector: 'app-product-detail',
 imports: [],
 templateUrl: './product-detail.component.html',
 styleUrl: './product-detail.component.css'
})
export class ProductDetailComponent {
readonly product = input<Product | undefined>();

}
```

```
<h2>Product List</h2>
<ul>
 @for (product of productState.productList(); track product.id) {
   <li>
     {{ product.title }} - ${{ product.price }}
     <button (click)="select(product.id)">View</button>
   </li>
 }
</ul>
<hr />

<app-product-detail [product]="productState.selectedProduct()" />
```

Step 6: Update AppComponent

app.component.ts

```
import { Component } from '@angular/core';
import { ProductDashboardComponent } from
'./product-dashboard/product-dashboard.component';
import { HttpClientModule } from '@angular/common/http';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [HttpClientModule, ProductDashboardComponent],
  template: `<app-product-dashboard />`
})
export class AppComponent {}
```

Step 7: Run & Observe

```
ng serve
```

You should see:

- Products loaded from API
- Clicking “View” updates the detail section
- Everything reacts instantly — **no manual subscription** needed

Concepts Practiced

Feature	Usage
Replace RxJS	Used `signal()` and `toSignal()`

Derived state	`computed()` for selected product
Side effect	`effect()` to log selection
Smart Component	`ProductDashboardComponent` handles state
Dumb Component	`ProductDetailComponent` uses signal input
Native signal input	`@Input({ signal: true })`

Bonus Challenges

Add Filter Signal

```
filter = signal('');
filteredProducts = computed(() =>
  this.products().filter(p =>
    p.title.toLowerCase().includes(this.filter().toLowerCase())
  )
);
```

Use in Dashboard

```
<input type="text" placeholder="Search..." [ngModel]="productState.filter()"
(ngModelChange)="productState.filter.set($event)" />
@for (product of productState.filteredProducts(); track product.id) { ... }
```

Log Selection with effect()

Already done:

```
effect(() => {
  console.log('Selected Product:', this.selectedProduct());
});
```

Add Price Update Signal (2-way)

```
price = signal(0);
```

Then bind `price()` to an input and update it. Use `effect()` to recalculate total or analytics.

Summary

This lab gave you a fully functioning **zoneless Angular app** with:

- Signal-based state
- Clean smart/dumb separation
- Fast, scoped change detection
- Native `@Input({ signal: true })`

DAY 4

Angular Performance & Testing

Focus: Standalone Components & Optimization Techniques

1. Standalone Component Optimization

Standalone components (introduced in Angular 14, matured in Angular 15+) eliminate the need for `NgModules`, leading to:

Benefit	Description
Faster build and runtime	Less overhead from module resolution and tree shaking
Smaller bundle size	Unused code is more easily tree-shaken
Fine-grained lazy loading	Components can be loaded directly via the router without modules
Simpler structure	Easier to maintain, test, and reason about individually

How to enable:

```
ng generate component my-cmp --standalone
```

Best practice:
Import only what is needed per component — this improves **bundle tree shaking**.

2. Performance Best Practices

Use Signals with OnPush or Zoneless Change Detection

- Minimize global change detection cycles
- Use `@if` / `@for` with signal-based state

Lazy Load by Route or Component

- Use `loadComponent` or `loadChildren` with `standalone: true`:

```
{
  path: 'dashboard',
  loadComponent: () => import('./dashboard.component').then(m => m.DashboardComponent)
}
```

Use `trackBy` in `@for` or `*ngFor`

Avoid full DOM re-rendering by tracking items uniquely.

Avoid Memory Leaks

- Clean up `effect()` with `cleanup()`
- Destroy subscriptions in non-signal logic

3. Testing Standalone Components

Standalone Test Setup

You no longer need `TestBed.configureTestingModule` with declarations. Instead:

```
import { render } from '@testing-library/angular';

await render(MyComponent, {
  componentProperties: { title: 'Hello' }
});
```

Use `TestBed.createComponent` (Optional)

Angular 15+ also supports:

```
const fixture = TestBed.createComponent(MyComponent);
fixture.detectChanges();
```

Mocking Standalone Services

Use `providers` and `imports` at the test level:

```
await render(MyComponent, {
  imports: [HttpClientTestingModule],
  providers: [
    { provide: MyService, useValue: mockService }
  ]
});
```

4. Tools for Performance Optimization

Tool	Purpose
Angular DevTools	Inspect signals, components, and change detection
`source-map-explorer`	Visualize bundle size, identify bloated imports
ESLint + Performance Rules	Enforce OnPush, pure pipes, etc.

Summary Table

Topic	Recommendation / Tip
Standalone components	Use for all new components; allows lazy loading without modules
Signals + Change Detection	Combine with OnPush or zoneless strategies

Lazy loading	Use `loadComponent` for component-level loading
Testing	Prefer `render()` API for simplicity and speed
Cleanup & memory	Use `cleanup()` in `effect()`; remove unused signal chains
Bundle optimization	Tree-shake with minimal imports and no unused module dependencies

Lab: Angular Performance & Testing with Standalone Optimization

Objectives

By the end of this lab, you will:

- Convert a feature module into a **standalone lazy-loaded component**
- Use `signal()`, `computed()`, and `@if` for **efficient reactivity**
- Enable **OnPush change detection** and measure render behavior
- Write **lightweight tests** for a standalone component using `render()`

Step 1: Setup a Standalone App

```
ng new perf-lab --standalone --routing --style=css
cd perf-lab
```

Step 2: Create a Feature Component for Lazy Loading

```
ng generate component dashboard --standalone --flat --skip-tests
```

Edit `src/app/dashboard.component.ts`:

```
import { Component, signal, computed } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  standalone: true,
  selector: 'app-dashboard',
  imports: [CommonModule],
  template: `
    <h2>Dashboard</h2>
    <p @if="counter() > 10"> High Activity!</p>
    <button (click)="increment()">Count: {{ counter() }}</button>
    <p>Double: {{ double() }}</p>
  `
})
export class DashboardComponent {
```

```
  counter = signal(0);
  double = computed(() => this.counter() * 2);

  increment() {
    this.counter.update(v => v + 1);
  }
}
```

Step 3: Lazy-Load Dashboard via `loadComponent()`

Edit `src/app/app.routes.ts`:

```
import { Routes } from '@angular/router';

export const routes: Routes = [
  {
    path: '',
    redirectTo: 'dashboard',
    pathMatch: 'full'
  },
  {
    path: 'dashboard',
    loadComponent: () =>
      import('./dashboard.component').then(m => m.DashboardComponent)
  }
];
```

Update `main.ts`:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { AppComponent } from './app/app.component';
import { routes } from './app/app.routes';

bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)]
});
```

Step 4: Enable OnPush (Optional)

Since we use signals, `OnPush` becomes optional — but enables stricter detection.

Update component metadata:

```
changeDetection: ChangeDetectionStrategy.OnPush
```

> You can import `ChangeDetectionStrategy` from `@angular/core`

Step 5: Profile in DevTools

- Use **Angular DevTools** to verify:
  - Signal reactivity chain
  - View doesn't re-render unless signal changes
- Use browser dev tools → Performance tab → record when clicking "Count"

Step 6: Write a Standalone Unit Test Using `render()`

Create: `src/app/dashboard.component.spec.ts`

```
import { render, screen } from '@testing-library/angular';
import { DashboardComponent } from './dashboard.component';

describe('DashboardComponent', () => {
  it('should render and increment count', async () => {
    await render(DashboardComponent);

    const button = await screen.findByText(/Count: 0/i);
    button.click();

    expect(screen.getByText(/Count: 1/)).toBeTruthy();
  });
});
```

Install testing library if not yet:

```
npm install @testing-library/angular --save-dev
```

Run tests:

```
ng test
```

Concepts Practiced

Concept	Practiced via
Standalone optimization	Component is standalone + lazy loaded
Efficient change detection	Signals + computed + `@if` syntax
Minimal rendering	Signal-driven logic, no full re-render
Lazy loading	`loadComponent()` usage
Testing	`render()` API from Angular Testing Library

Bonus Challenges

- Add a timer (`effect()` + `cleanup()`) that auto-increments the counter every 3s
- Measure bundle size with `source-map-explorer`
- Add another standalone page (`/reports`) and lazy load it too

Angular SSR (Server-Side Rendering)

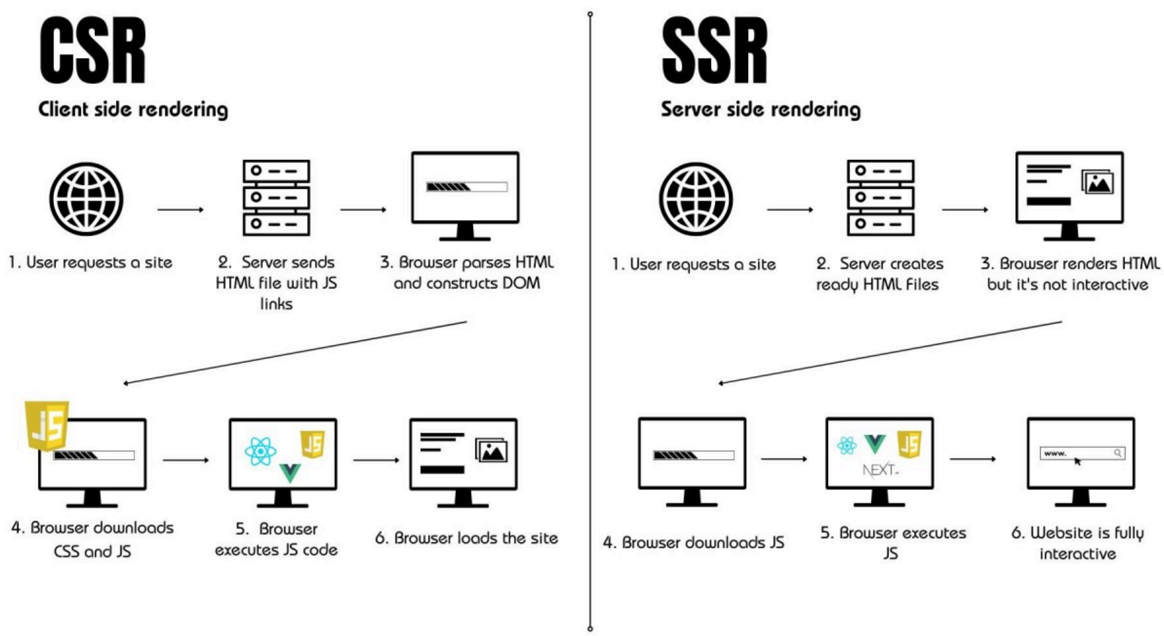
Focus: Hydration + Pre-rendering with Angular Universal (v16–20+)
Docs: https://angular.dev/guide/hydration

What is Angular Universal (SSR)?

Angular Universal enables Angular apps to render HTML on the **server** (Node.js) instead of waiting for the browser to render with JavaScript.

SSR sends **fully rendered HTML** to the browser, resulting in:

- Faster **perceived load time**
- Better **SEO**
- Proper **social media link previews**



Why Use SSR?

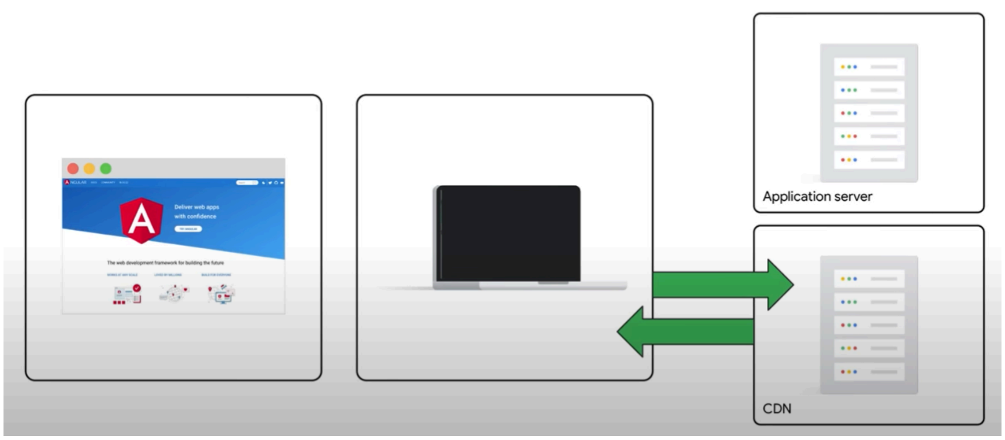
Benefit	Description
Faster LCP	Sends usable HTML immediately (faster **Largest Contentful Paint**)
SEO Optimization	Search engine bots can crawl your app as rendered HTML
Social Sharing	Link previews (e.g., Twitter, Facebook) show real content
Better UX	Reduces blank screen or flickering, improves **Time to Interactive**

Key Concepts

1. **Hydration (Angular 16+)**

Hydration is the process where:

- Server sends **rendered HTML**
- Angular bootstraps and **attaches behavior** to that HTML
- DOM is **not** recreated → No flicker, Faster interaction



Hydration Behavior:

- No duplicate rendering
- State can be preserved
- Signals and inputs supported (Angular 17+)
- Automatic event reattachment

Code Example (main.ts)

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()],
});
```

- Angular now hydrates using the actual server DOM instead of re-rendering it in the browser.

Pre-rendering

Pre-rendering is the process of generating **static HTML at build time** for specific routes.

Best for:

- Marketing pages
- Blogs
- FAQ, About, Landing Pages
- Pages that don’t rely on dynamic user data

Command:

```
ng run your-app-name:prerender
```

Outputs static HTML into:

```
dist/<project>/browser/
```

- These HTML files can be hosted on **any static server or CDN** — no Node.js required.

Hydration vs Pre-rendering

Feature	Hydration	Pre-rendering
Timing	On demand (runtime, server-rendered)	At build time
Use Case	Dynamic pages (auth, dashboards)	Static content (landing, blog)
SEO	Yes	Yes
JS Needed	Yes (to hydrate and interact)	No (for display), yes (for interaction)
Hosting	Node.js server (e.g., Express, Vercel)	Static CDN (Netlify, Firebase, GitHub)

Setup Guide: Enabling SSR + Hydration

1. Add Angular Universal

```
ng add @angular/ssr
```

This will:

- Add `server.ts`, `main.server.ts`, `app.server.module.ts`
- Update `angular.json` with SSR & prerender targets
- Install Express and Angular Universal dependencies

2. Hydration Support

Supported from **Angular 16+**
Improved in **Angular 17, 18, 19**, and **Angular 20**:

- Better signal-based component hydration
- `<img ngSrc>` hydration
- Improved reactivity + zone-less support (optional)

3. Build and Serve with SSR

```
npm run build:ssr
npm run serve:ssr
```

Visit http://localhost:4000
You'll see rendered HTML even with JavaScript disabled.

Pre-rendering Static Routes

In `angular.json → projects → your-app → architect → prerender → options`:

```
"routes": ["/", "/about", "/contact"]
```

Then run:

```
ng run your-app-name:prerender
```

Static `.html` files are written to `dist/<your-app>/browser`

Debug & Validate SSR + Hydration

Tool	What to Check
View Page Source	HTML should contain full DOM content (`<h1>`, etc.)
Chrome DevTools > Elements	DOM should not re-render or flicker
Performance Tab	Look for hydration markers in Angular DevTools
Lighthouse	Check LCP, FCP, SEO score, and Time to Interactive

Bonus Tips & Real-World Use

- **Use TransferState** to avoid duplicate HTTP fetches on client
- **Lazy load routes** with SSR-friendly techniques
- **Use signals with hydration** — tested from Angular 17+

Summary

Concept	Key Takeaway
SSR	Renders Angular HTML on the server (Node.js) for speed and SEO
Hydration	Attaches interactivity to server-rendered HTML without replacing the DOM
Pre-rendering	Generates HTML at build time for static pages (no server required)

Version	SSR supported in all Angular versions; Hydration supported from Angular 16+
Hosting	SSR needs Node.js (Express); Pre-rendered output is deployable anywhere
Use Case	Use **SSR + hydration** for dynamic apps; use **pre-rendering** for static pages

Using `TransferState` for API Integration in SSR

What is TransferState?

`TransferState` is a built-in Angular mechanism that:

- Allows you to **fetch data on the server**
- Embed that data into the rendered HTML
- Automatically **reuses** the data on the client — avoiding **duplicate HTTP calls**

Use Case Example: Fetch from REST API

Imagine we want to fetch blog posts or a list of products.

Step 1: Create a Basic HTTP Service

```
// src/app/services/product.service.ts
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class ProductService {
  constructor(private http: HttpClient) {}

  getProducts() {
    return this.http.get('https://fakestoreapi.com/products');
  }
}
```

Step 2: Use `TransferState` in Component

```
import {
  makeStateKey,
  TransferState
} from '@angular/platform-browser';
import { Component, inject, effect, signal } from '@angular/core';
import { ProductService } from '../services/product.service';

const PRODUCTS_KEY = makeStateKey<any[]>('products');

@Component({
```

```
  selector: 'app-product-list',
  standalone: true,
  imports: [],
  template: `
    <h1>Product List</h1>
    <div *ngFor="let product of products()">
      {{ product.title }}
    </div>
  `,
})
export class ProductListComponent {
  private productService = inject(ProductService);
  private state = inject(TransferState);

  products = signal<any[]>([]);

  constructor() {
    const existing = this.state.get(PRODUCTS_KEY, null);
    if (existing) {
      this.products.set(existing);
    } else {
      this.productService.getProducts().subscribe((res) => {
        this.products.set(res);
        this.state.set(PRODUCTS_KEY, res);
      });
    }
  }
}
```

How This Works with SSR:

- On the server:
    - HTTP call runs
    - Data is embedded in HTML as a script tag
- On the client:
    - Angular checks if data is already transferred
    - No need to re-fetch!

Improves performance, avoids flickering, saves API calls

Helpful Notes

- You can use `makeStateKey<T>()` to store any serializable data
- `TransferState` is only active in **SSR mode**
- Works well with **signals** and **hydrated apps**

Summary Table (Extended)

Feature	Purpose	SSR Impact
SSR	Render HTML on server	SEO, FCP, LCP boost
Hydration	Reuse server DOM, attach events	No flicker, fast TTI
Pre-rendering	Build-time HTML for static routes	CDN-ready pages
TransferState	Share data from server → client to avoid re-fetching	One HTTP call only
REST API	Can be hydrated via TransferState	Dynamic data supported

Lab: Angular SSR with Hydration + Pre-rendering + Dynamic Data

Objectives

By the end of this lab, you will:

- Enable Server-Side Rendering (SSR) and Hydration in an Angular app
- Pre-render static routes at build time
- Create a dynamic page using Signals and TransferState
- Understand CSR vs SSR vs SSG (Pre-rendering)
- Compare performance using Lighthouse and DevTools

Step 1: Create or Use an Existing Angular App

```
ng new ssr-lab --routing --style=css
cd ssr-lab
```

Add basic routes (e.g. Home, About):

```
ng generate component pages/home
ng generate component pages/about
```

Edit app.routes.ts:

```
import { Routes } from '@angular/router';

export const routes: Routes = [
  { path: '', loadComponent: () => import('./pages/home/home.component').then(m =>
m.HomeComponent) },
  { path: 'about', loadComponent: () => import('./pages/about/about.component').then(m =>
m.AboutComponent) }
];
```

Step 2: Verify or add Angular SSR support

```
ng add @angular/ssr
```

This will:

- Create server.ts, **main.server.ts,** and **app.routes.server.ts**
- Install Express and SSR dependencies

Step 3: Verify that Hydration is Enabled

Edit app.config.ts:

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()],
});
```

This ensures Angular will reuse **server-rendered HTML** instead of re-rendering from scratch.

Step 4: Verify SSR is Working

```
npm run build
npm run serve:ssr:ssr-lab
```

Open http://localhost:4000 and:

- View source (you should see full HTML)
- Inspect in **DevTools > Elements** tab
- Look in **DevTools > Performance** tab for hydration markers

Rendering Comparison (SSR vs CSR)

Behavior	CSR (`ng serve`)	SSR (`serve:ssr`)
View Source	Empty `app-root`	Full HTML with actual content
First Paint (without JS)	Blank until JS loads	Immediate content visible
SEO Optimization	Not crawlable by bots	Crawlable, fully indexed
Dynamic Data (Articles Page)	Loaded via HTTP	Loaded and embedded via `TransferState`
DOM Hydration	No reuse	DOM is reused, not re-rendered

Tip: Use Chrome DevTools to compare "Elements" tab and "Performance" traces for both modes.

Step 5: Add Static Pre-rendering

Instead of modifying `angular.json`, use the modern Angular 20 approach via `app.routes.server.ts`.

Edit `server.ts`:

```
import { RenderMode, ServerRoute } from '@angular/ssr';

export const serverRoutes: ServerRoute[] = [
  { path: '', renderMode: RenderMode.Prerender },
  { path: 'about', renderMode: RenderMode.Prerender },
  { path: '**', renderMode: RenderMode.Server },
];
```

Build and run prerendered pages:

```
ng build
 npm run serve:ssr:ssr-lab
```

Check:

- `dist/ssr-lab/browser/index.html`
- `dist/ssr-lab/browser/about/index.html`

These are static HTML files ready to deploy.

Step 6: Create a Dynamic Page (API + TransferState)

6.1 Create Articles Service

```
ng generate service services/articles
```

`articles.service.ts`:

```
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { TransferState, makeStateKey } from '@angular/core';
import { tap } from 'rxjs';

const ARTICLES_KEY = makeStateKey<any[]>('articles');

@Injectable({ providedIn: 'root' })

export class ArticlesService {

 private http = inject(HttpClient);
 private transferState = inject(TransferState);

 getArticles() {
   if (this.transferState.hasKey(ARTICLES_KEY)) {
     const data = this.transferState.get<any[]>(ARTICLES_KEY, []);
     this.transferState.remove(ARTICLES_KEY);
     return data;
   }

   return this.http.get<any[]>('https://jsonplaceholder.typicode.com/posts').pipe(
     tap(data => this.transferState.set(ARTICLES_KEY, data))
```

```
  );
 }
}
```

Add the providerHttp in app.config.ts

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners,
provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient, withFetch } from '@angular/common/http';
import { routes } from './app.routes';
import { provideClientHydration, withEventReplay } from '@angular/platform-browser';

export const appConfig: ApplicationConfig = {
 providers: [
   provideBrowserGlobalErrorListeners(),
   provideZoneChangeDetection({ eventCoalescing: true }),
   provideRouter(routes),
 provideHttpClient(withFetch()),
    provideClientHydration(withEventReplay())
 ]
};
```

6.2 Create Articles Component with Signal

```
ng generate component pages/articles
```

articles.component.ts:

```
import { Component, inject, OnInit, signal } from '@angular/core';
import { ArticlesService} from '../../services/articles';

interface Article {
 userId: number;
 id: number;
 title: string;
 body: string;
}

@Component({
 selector: 'app-articles',
```

```
 imports: [],
 templateUrl: './articles.html',
 styleUrl: './articles.css'
})
export class Articles implements OnInit  {
private service = inject(ArticlesService);
 articles = signal<Article[]>([]);

 ngOnInit() {
   const result = this.service.getArticles();
   if (Array.isArray(result)) {
     this.articles.set(result.slice(0, 5));
   } else {
     result.subscribe(data => this.articles.set(data.slice(0, 5)));
   }
 }

}
```

articles.component.html:

```
 <h2>Latest Articles</h2>
   <ul>
     @for( article of articles(); track article.id) { // This will now work
        <li>{{ article.title }}</li>
     }
   </ul>
```

6.3 Add to Router & Pre-render

Edit app.routes.ts:

```
{ path: 'articles', loadComponent: () => import('./pages/articles/articles').then(m => m.Articles)
}
```

Update app.routes.ts to add asprerender routes:

```
import { RenderMode, ServerRoute } from '@angular/ssr';

export const serverRoutes: ServerRoute[] = [
```

```
 { path: '', renderMode: RenderMode.Prerender },
 { path: 'about', renderMode: RenderMode.Prerender },
 { path: 'articles', renderMode: RenderMode.Prerender },
 { path: '**', renderMode: RenderMode.Server },
];
```

Re-run:

```
ng build
 npm run serve:ssr:ssr-lab
```

You now have a dynamic page using SSR + TransferState + Signals!

1. When a user requests `/articles`, your Node.js server executes your Angular application.
2. The `ArticlesService` fetches the article data from the API while running on the server.
3. This fetched data is then stored in `TransferState`.
4. The server renders the `Articles` component (using the data from the service) into static HTML, including the `TransferState` data embedded within a `<script>` tag.
5. This pre-rendered HTML is sent to the user's browser.
6. The browser immediately displays the content of the `/articles` page.
7. Once the JavaScript bundle loads, the client-side Angular application "hydrates" – it takes over the already rendered HTML.
8. The `ArticlesService` on the client-side *first* checks `TransferState`. Since the data is already there, it retrieves it from `TransferState` instead of making a new HTTP request.
9. The `articles` signal in your component is initialized with this data, and the application becomes fully interactive without any noticeable re-rendering or flickering.

Important: How Data Works with SSR + TransferState

When using **SSR + TransferState + Signals** for the /articles page:

- If you're using **RenderMode.Prerender**, Angular fetches and embeds data **at build time**.
  - ➢ Fast page load, but **new articles won't appear** unless you rebuild your app.

- If you're using **RenderMode.Server**, Angular fetches fresh data on **every request**.
    - ➢ Always up-to-date, supports dynamic data, slightly slower first load.

To show latest data without rebuilding, use RenderMode.Server for /articles:

```
{ path: 'articles', renderMode: RenderMode.Server }
```

Step 7: Understand the Different Types of Rendering

Rendering Type	Description	Use Case
CSR (Client-Side)	Renders entirely in browser after JS loads	SPAs, internal dashboards
SSR (Server-Side)	Renders HTML on server and sends to client	SEO-heavy, content-first pages
Hydration	Angular reuses SSR DOM without re-rendering on client boot	Boosts performance + avoids flicker
Pre-rendering (SSG)	HTML generated at build time for fixed routes	Static blogs, marketing pages

Step 8: Compare the Performance

8.1 With Lighthouse

Open Chrome DevTools → Lighthouse tab:

Run tests on:

- CSR build (ng serve)
- SSR (npm run serve:ssr:ssr-lab)
- Pre-rendered static output (dist/ssr-lab/browser with live-server or Netlify)

Compare:

- Time to First Byte (TTFB)

- First Contentful Paint (FCP)
- Largest Contentful Paint (LCP)

8.2 With Angular DevTools

Open Angular DevTools > Profiler

Check when Hydration begins and ends

Compare interactivity time and hydration cost

Summary Table

Feature	CSR	SSR + Hydration	Pre-rendering (SSG)
SEO Friendly	No	Yes	Yes
First Paint Performance	Slow	Fast	Instant
JavaScript Required	Yes	Yes	No (until hydrated)
Dynamic Data	Yes	Yes(TransferState)	No (requires JS)
Hosting Type	Static/CDN	Node.js server	Static (Netlify, etc)
Use Case	Dashboards	Blogs, CMS	Marketing, Docs

Bonus Challenges

- Add a Contact page and pre-render it
- Use computed() or effect() to display article count
- Add a loading state using signals and show spinner
- Try removing JavaScript (<noscript>) to observe fallback

Bonus Challenge Solutions

1. Add a Contact Page and Pre-render It

1.1 Generate the Contact Page

```
ng generate component pages/contact
```

1.2 Edit app.routes.ts

```
{ path: 'contact', loadComponent: () => import('./pages/contact/contact').then(m => m.Contact)
}
```

1.3 Add Simple HTML (contact.component.html)

```
<h2>Contact Us</h2>
<p>Email: hello@example.com</p>
<p>Phone: +6012-3456789</p>
```

1.4 Add to Pre-rendering Routes in angular.json

```
import { RenderMode, ServerRoute } from '@angular/ssr';

export const serverRoutes: ServerRoute[] = [
 { path: '', renderMode: RenderMode.Prerender },
 { path: 'about', renderMode: RenderMode.Prerender },
 { path: 'articles', renderMode: RenderMode.Server },
 {path: 'contact', renderMode: RenderMode.Prerender },
 { path: '**', renderMode: RenderMode.Server },
];
```

1.5 Re-run Pre-rendering

```
ng build
```

- dist/ssr-lab/browser/contact/index.html should now exist and be viewable without JS.

2. Use computed() or effect() to Display Article Count

Modify [articles.component.ts]()

```
import { Component, computed, effect, inject, OnInit, signal } from '@angular/core';
import { ArticlesService } from '../../services/articles.service';

@Component({
  standalone: true,
  selector: 'app-articles',
  templateUrl: './articles.component.html',
})
export class ArticlesComponent implements OnInit {
  private service = inject(ArticlesService);

  articles = signal<any[]>([]);

  articleCount = computed(() => this.articles().length);

  ngOnInit() {
    const result = this.service.getArticles();
    if (Array.isArray(result)) {
      this.articles.set(result.slice(0, 5));
    } else {
      result.subscribe(data => this.articles.set(data.slice(0, 5)));
    }

    effect(() => {
      console.log('Article count:', this.articleCount());
    });
  }
}
```

Update articles.component.html

```
<h2>Latest Articles ({{ articleCount() }})</h2>
<ul>
  @for (article of articles(); track article.id) {
    <li>{{ article.title }}</li>
  }
</ul>
```

3. Add a Loading State Using signal() and Show a Spinner

Update articles.component.ts

```
loading = signal(true);

ngOnInit() {
  const result = this.service.getArticles();
  if (Array.isArray(result)) {
    this.articles.set(result.slice(0, 5));
    this.loading.set(false);
  } else {
    result.subscribe(data => {
      this.articles.set(data.slice(0, 5));
      this.loading.set(false);
    });
  }
}
```

Update articles.component.html

```
<h2>Latest Articles ({{ articleCount() }})</h2>

@if (loading()) {
  <div>Loading articles...</div>
} @else {
  <ul>
    @for (article of articles(); track article.id) {
      <li>{{ article.title }}</li>
    }
  </ul>
}
```

4. Try Removing JavaScript to Observe Fallback

Steps:

- Use your browser's DevTools → Network tab → Block JS, or
- Use <noscript> block in index.html, or

- Use curl or View Page Source

What You'll Observe:

- Pre-rendered pages (/about, /contact) will display content correctly
- Dynamic routes (/articles) will show content only if pre-rendered or TransferState was used
- No interactivity (like loading spinner or dynamic count) without JS

Summary: Bonus Challenge Outcomes

Challenge	**What You Achieved**
Contact page	Added standalone, pre-rendered route
computed()/effect() usage	Reactive article count + console logging
Loading state	Used signal to show/hide loading spinner
JS disabled test	Verified SSR and pre-rendering fallback compatibility

Angular Signals & Zoneless SSR Compatibility

Why This Matters

Angular's reactivity system is evolving:

- **Signals** (Angular 16+) introduce fine-grained, explicit state tracking.
- **Zoneless Angular** (Angular 17+) removes the need for `Zone.js`, which has historically powered Angular's change detection.
- **SSR** (Server-Side Rendering) becomes more deterministic and lightweight with Signals.

Together, these innovations lead to:

- Better performance and predictability
- Simplified mental model (less magic from Zone.js)
- More control over hydration and rendering behavior

How Signals Empower Zoneless SSR

Concept	Traditional Angular (with Zone.js)	Signals-based Angular (Zoneless)
Change Detection	Implicit via patched async events	Explicit via `signal()`, `effect()`, etc.
Reactivity Model	Dirty-checking and lifecycle-based	Dependency-tracked reactive graph
SSR Bootstrapping	Needs Zone patches	Native via scheduling/reactivity

Hydration	Often requires DOM patches or rerendering	Seamless – DOM preserved and wired to state
Debugging	Hard to trace due to global patching	Transparent reactivity flow with Signals
Performance	Overhead from dirty checks and zones	Optimized and event-driven

How to Enable Zoneless SSR with Signals (Angular 17+)

1. Build With Signals

Use `signal()`, `computed()`, `effect()` for app state:

```
count = signal(0);
double = computed(() => count() * 2);
effect(() => console.log('Count changed:', count()));
```

2. Enable Hydration

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()],
});
```

3. Disable Zone.js (Optional)

If you want to go fully **zoneless**, simply remove this line:

```
// REMOVE THIS:
import 'zone.js';
```

Angular will automatically switch to scheduler-based change detection.

Why Zoneless Angular Matters

- Removes global `Zone.js` dependency
- Easier to reason about app behavior
- Boosts performance by removing dirty-check cycles
- Aligns Angular with modern reactive frameworks like React (hooks), Svelte (stores), Solid.js (fine-grained reactivity)

Caveats & Best Practices

DO	AVOID
Use signal() for state	Avoid @Input() for primitives
Use @Input({ signal: true })	Avoid EventEmitter for component outputs
Use effect() for derived side effects	Don't rely on ngOnChanges or ngDoCheck
Use untracked() when avoiding reactivity	Avoid mutating DOM before hydration
Track hydration in DevTools	Avoid manual change detection calls

SSR + Signals Compatibility Checklist

Feature	Supported in Angular 17+ / 18 / 19
`signal()` during SSR	Yes
`computed()` in SSR	Yes
`effect()` during SSR	Yes (use sparingly)

Hydration of signals	Yes
Pre-rendering with signals	Yes
SSR without Zone.js	Yes
Server-side reactivity graphs	Yes
Integration with `TransferState`	Yes

Summary

Term	Description
Signals	Explicit, dependency-tracked reactive state
Zoneless Angular	Angular running without Zone.js, more efficient and modern
Hydration	Reusing server-rendered HTML without client re-render
SSR + Signals	Ideal combo for fast, predictable rendering
Angular 19+	Fully optimized for signals-first and zone-free workflows

Extra Notes

- Signals are not just a performance feature — they're a **paradigm shift** in how Angular apps are structured and reasoned about.
- The **debugging experience** improves with a visible reactivity graph (via DevTools in future releases).
- Signals also enable **Web Component compatibility** and better **lazy-loading strategies** since they avoid zone interference.

Lab: Angular Signals + Zoneless SSR

Objectives

By the end of this lab, you will:

- Set up an Angular app without `Zone.js`
- Use Angular **Signals** for state management
- Enable **SSR with Hydration** in zoneless mode
- Verify DOM preservation and reactive updates without dirty checking

Step 1: Create a New Angular App

```
ng new zoneless-ssr-lab --standalone --routing --style=css
cd zoneless-ssr-lab
```

Step 2: Add Angular Universal (SSR)

```
ng add @angular/ssr
```

This will:

- Generate `server.ts`, `main.server.ts`, `app.server.module.ts`
- Add SSR build targets to `angular.json`
- Install `@nguniversal/express-engine` and dependencies

*Select Yes for zoneless and SSR

Step 3: Verify that hydration is added

Edit `main.ts`:

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()],
});
```

Hydration will reuse DOM on the client without rerendering it.

Step 4: Verify that Zone.js is removed(Zoneless Mode)

Open `main.ts` and **delete or comment out** this line:

```
// REMOVE THIS LINE:
import 'zone.js';
```

Angular will now use scheduler-based change detection (no Zone patches).

Step 5: Create Signal-Based Component

```
ng generate component pages/counter --standalone
```

Edit `counter.component.ts`:

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

@Component({
  selector: 'app-counter',
  standalone: true,
  templateUrl: './counter.component.html',
})
export class CounterComponent {
  count = signal(0);
  double = signal(0);

  constructor() {
    effect(() => {
      this.double.set(this.count() * 2);
      console.log('Double updated to:', this.double());
    });
  }
```

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

Edit `counter.component.html`:

```
<h2>Counter</h2>
<p>Count: {{ count() }}</p>
<p>Double: {{ double() }}</p>
<button (click)="increment()">Increment</button>
```

Step 6: Add Route

Update `app.routes.ts`:

```
{ path: 'counter', loadComponent: () => import('./pages/counter/counter.component').then(m
=> m.CounterComponent) }
```

Step 7: Run With SSR and Hydration

```
npm run build
npm run serve:ssr:zoneless-ssr-lab
```

Visit `http://localhost:4000/counter`

Verify:

- In **View Source**, the counter DOM is fully rendered
- In **DevTools** → **Performance**, look for **Hydration Start**
- Click **Increment** – it should update without full rerender

Step 8: Add `computed()` and `effect()` (Optional)

In `counter.component.ts`, refactor:

```
import { computed } from '@angular/core';
double = computed(() => this.count() * 2);
```

Use `effect()` to log changes:

```
effect(() => {
  console.log(`Count is ${this.count()}, double is ${this.double()}`);
});
```

Step 9: Track DOM Behavior

Open Chrome DevTools and:

- Inspect `<p>` values – observe no DOM replacements
- Measure hydration time and interaction responsiveness

Completion Checklist

Task	Done?
Angular SSR enabled	Yes
Zone.js removed	Yes
Signal-based component used	Yes
Hydration verified (source + DevTools)	Yes
computed() and effect() used correctly	Yes

Reflection: Why This Lab Matters

- Signals enable Angular to **work without Zone.js**, making behavior explicit and efficient
- Zoneless SSR + Hydration gives **blazing-fast** performance and predictable output
- This setup mirrors modern frameworks like **React Server Components** or **Solid.js SSR**

Bonus Exercises

- Add a `reset()` method using `count.set(0)`
- Add a `signal<string>` to show a status like `"Even"` or `"Odd"`
- Try adding a `setTimeout` and observe change detection in zoneless mode
- Add a signal input using `@Input({ signal: true })` in a child component

Bonus Exercise 1: Add a `reset()` method

Update `counter.component.ts`:

```
reset() {
  this.count.set(0);
}
```

Update `counter.component.html`:

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

Bonus Exercise 2: Add a `signal<string>` status ("Even" or "Odd")

Update `counter.component.ts`:

Add this signal:

```
status = computed(() => this.count() % 2 === 0 ? 'Even' : 'Odd');
```

Update `counter.component.html`:

```
<p>Status: {{ status() }}</p>
```

Bonus Exercise 3: Add a `setTimeout()` and observe reactivity

Update `counter.component.ts` (inside constructor or ngOnInit):

```
setTimeout(() => {
  this.count.update(v => v + 5);
  console.log('setTimeout triggered count += 5');
}, 2000);
```

In **zoneless mode**, this will still work because Angular now uses a **scheduler** that tracks signals without needing Zone.js.

You'll see:

- DOM updates after 2 seconds
- No Zone patching required

Bonus Exercise 4: Use `@Input({ signal: true })` in a child component

1. Generate a child component

```
ng generate component components/display --standalone
```

2. `display.component.ts`:

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

@Component({
  selector: 'app-display',
  standalone: true,
  template: `<p>From Child: Count is {{ count() }}</p>`,
})
export class DisplayComponent {
```

```
  @Input({ signal: true }) count!: Signal<number>;
}
```

3. Use in `counter.component.ts`:

Import the child:

```
import { DisplayComponent } from '../../components/display/display.component';
```

Add it to the `@Component` metadata:

```
@Component({
  ...
  imports: [CommonModule, DisplayComponent],
})
```

4. Update counter.component.html:

```
<app-display [count]="count" />
```

Result

- `app-display` receives `count` as a **live signal**, not a snapshot
- Any update to `count` is reflected instantly in the child
- No need for `@Input()` + `ngOnChanges()` boilerplate

What is Incremental Hydration?

https://angular.dev/guide/incremental-hydration

Incremental Hydration is a modern Angular SSR feature that allows **deferred hydration of individual components**, rather than hydrating the entire DOM at once on page load.

Why It Matters

Traditional Hydration:

- After server renders HTML, Angular **hydrates (reconnects)** all components immediately on client boot.
- This can **block the main thread**, especially on large apps.
- All interactivity is paused until the hydration finishes.

Incremental Hydration:

- Angular hydrates **only part of the DOM**, and **defers other components** until needed.
- Reduces Time-to-Interactive (TTI)
- Improves performance and user experience (especially on mobile and low-end devices)

Key Concept: `@defer` Blocks

Angular 17+ introduces the `@defer` syntax, which enables incremental hydration:

```
@defer (when visible) {
  <app-heavy-component />
}
```

This tells Angular:

> "Don't hydrate this component until it scrolls into view."

Supported Hydration Triggers

Syntax	Description	Use Case Example
`@defer (on idle)`	Wait until browser is idle	Charts, metrics, analytics sections
`@defer (when visible)`	Hydrate when component becomes visible in viewport	Footer, testimonials, pricing cards
`@defer (when click)`	Hydrate when a user triggers a signal or interaction	Modals, tabs, "Load More" buttons
`@defer (after 2s)`	Hydrate after a fixed delay (e.g. 2 seconds)	Non-critical UI (ads, banners)

Example

```
<!-- home.component.html -->

<h1>Welcome!</h1>

@defer (on idle) {
  <app-news-feed />
}

@defer (when visible) {
  <app-footer />
}
```

The rest of the app becomes interactive immediately, but:

- `<app-news-feed>` waits until the main thread is idle
- `<app-footer>` is hydrated only when user scrolls down

What Happens in the Browser?

Angular initially renders **placeholders** in place of deferred components (e.g., `<ng-defer-placeholder>`).

When the hydration trigger is met:

- Angular **hydrates the component**
- Placeholder is replaced with the **interactive Angular component**
- You can inspect this behavior in **DevTools > Elements** and **Performance tab**

Benefits of Incremental Hydration

Benefit	Why It Matters
Faster Time-to-Interactive	Only critical components are hydrated immediately
Reduced JS execution	Less work on initial boot
Better UX on slow devices	Defers low-priority UI components
Explicit control	Developers decide *when* hydration happens

Limitations / Caveats

- You **must use standalone components** to enable defer blocks
- Deferred components **should not block critical user interactions**
- SSR and `@defer` work best with **hydration enabled** via `provideClientHydration()`

- Lazy loading is not the same as deferred hydration (but they can be combined)

Summary

Term	Description
Hydration	Reconnecting static HTML to Angular's reactivity
Incremental Hydration	Hydrating parts of the app on-demand
`@defer`	Angular directive to control hydration timing and triggers
Goal	Faster, smarter, more interactive web apps post-SSR

Lab: Try Incremental Hydration with @defer

Objective

You will:

- Use Angular's `@defer` to delay hydration
- Apply `when visible` to hydrate a component only when it's scrolled into view
- Observe hydration behavior using DevTools

Step 1: Create a New Standalone Component

```
ng generate component pages/slow-widget --standalone
```

Update `slow-widget.component.ts`:

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

@Component({
  standalone: true,
  selector: 'app-slow-widget',
  templateUrl: './slow-widget.component.html',
})
export class SlowWidgetComponent implements OnInit {
  message = signal('Loading data...');

  ngOnInit(): void {
    setTimeout(() => {
      this.message.set(' Widget hydrated and data ready!');
    }, 1000);
  }
}
```

`slow-widget.component.html`:

```
<h3>Deferred Widget</h3>
<p>{{ message() }}</p>
```

Step 2: Use It with `@defer` in Home Page

Open `home.component.html` (or `app.component.html`) and add:

```
<h1>Welcome to the Homepage</h1>

<div style="height: 100vh; background: #f5f5f5;">
  <p>Scroll down to see the deferred widget</p>
</div>

@defer (when visible) {
  <app-slow-widget />
}
```

Step 3: Enable Hydration (if not yet)

In `main.ts`:

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()],
});
```

Step 4: Serve with SSR

```
npm run build
npm run serve:ssr:ssr-lab
```

Open `http://localhost:4000`

Step 5: Test and Observe

- Do **not scroll** — the `<app-slow-widget>` should not be hydrated yet
- Scroll down → after visibility, the component should load
- Open **DevTools** → **Elements** → you'll see `ng-defer-placeholder` replaced
- Open **Performance tab** to track deferred hydration

Bonus: Try Other Triggers

Replace `(when visible)` with:

- `(on idle)`
- `(after 2s)`
- `(when showWidget)` with a signal condition

Testing Signals in Angular

Unit Testing - Reactive Flows -Best Practices

What Are Angular Signals?

- Signals are **reactive primitives** used to manage state in a predictable and fine-grained way.
- They replace and simplify patterns previously handled by `RxJS`, `EventEmitter`, or `@Input()`/`@Output()`.

1. Unit Testing Signal State

Test a Basic Signal

You can directly test signal values using `.()` and `.set()` or `.update()`:

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

describe('Signal basics', () => {
  it('should initialize and update correctly', () => {
    const count = signal(0);
    count.set(5);
    expect(count()).toBe(5);

    count.update(c => c + 2);
    expect(count()).toBe(7);
  });
});
```

Test a `computed()`

```
import { computed, signal } from '@angular/core';

describe('Computed signal', () => {
  it('should recompute when dependency changes', () => {
    const price = signal(100);
    const tax = computed(() => price() * 0.1);

    expect(tax()).toBe(10);

    price.set(200);
```

```
    expect(tax()).toBe(20);
  });
});
```

Test `effect()` with Tracking

```
import { signal, effect } from '@angular/core';

describe('Signal effect', () => {
  it('should trigger when the signal changes', () => {
    const logs: number[] = [];
    const count = signal(1);

    effect(() => {
      logs.push(count());
    });

    count.set(2);
    count.set(3);

    expect(logs).toEqual([1, 2, 3]);
  });
});
```

2. Testing Signal Reactivity in Components

Use Angular Testing Library or `TestBed` for DOM behavior:

Example Component

```
@Component({
  standalone: true,
  selector: 'app-counter',
  template: `
    <p>Count: {{ count() }}</p>
    <button (click)="increment()">Increment</button>
  `
})
export class CounterComponent {
  count = signal(0);
  increment() {
    this.count.update(c => c + 1);
  }
}
```

Test with Testing Library

```
import { render, screen } from '@testing-library/angular';
import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  it('should increment signal state on click', async () => {
    await render(CounterComponent);

    const button = screen.getByText(/Increment/);
    button.click();

    expect(screen.getByText(/Count: 1/)).toBeTruthy();
  });
});
```

This tests both:

- Signal reactivity
- DOM updates based on reactive flow

3. Best Practices for Signal Testing

Practice	Why it Matters
Use `.()` to read signal value	Always access the current value via `.()`
Use `.set()` / `.update()`	Avoid mutating signal state directly
Test `computed()` independently	Don’t always rely on UI for derived tests
Use `effect()` in a test context	Validate reactivity or logging behavior

Clean up side effects	Use manual teardown if effects are used

Extra: Mocking Signal Inputs in Components (Angular 17+)

If a child component uses:

```
@Input({ signal: true }) data!: Signal<number>;
```

You can pass a test signal like this:

```
await render(ChildComponent, {
  componentInputs: {
    data: signal(10)
  }
});
```

Summary Table

Signal Feature	Test Strategy
`signal()`	Check value with `.()`, use `.set()`
`computed()`	Test output updates as dependencies change
`effect()`	Track logs or reactivity manually
DOM Integration	Use Angular Testing Library for interaction
Signal Inputs	Use `@Input({ signal: true })` and mock

Lab: Testing Signals — Unit & Component Integration

Objectives

By the end of this lab, you will:

- Write unit tests for `signal()`, `computed()`, and `effect()`
- Test a component that uses signals for state
- Validate DOM reactivity using **Testing Library**
- Practice mocking signal inputs for isolated testing

Prerequisites

- Install Angular Testing Library (if not already):

```
npm install @testing-library/angular --save-dev
```

Step 1: Set Up Angular App

```
ng new signals-testing-lab --standalone --routing=false --style=css
cd signals-testing-lab
```

Step 2: Create a Signal-Based Component

```
ng generate component counter --standalone
```

Edit `counter.component.ts`:

```
import { Component, signal, computed } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  standalone: true,
  selector: 'app-counter',
```

```
  imports: [CommonModule],
  template: `
    <h2>Counter</h2>
    <p>Count: {{ count() }}</p>
    <p>Double: {{ double() }}</p>
    <button (click)="increment()">Increment</button>
  `
})
export class CounterComponent {
  count = signal(0);
  double = computed(() => this.count() * 2);

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

Step 3: Unit Test Signal Logic (Pure Functions)

Create `src/app/counter.signal.spec.ts`:

```
import { signal, computed, effect } from '@angular/core';

describe('Signal basics', () => {
  it('should update the signal correctly', () => {
    const count = signal(0);
    count.set(5);
    expect(count()).toBe(5);
  });

  it('should compute values from a signal', () => {
    const price = signal(100);
    const tax = computed(() => price() * 0.1);
    expect(tax()).toBe(10);
    price.set(200);
    expect(tax()).toBe(20);
  });

  it('should trigger effect when signal changes', () => {
    const log: number[] = [];
    const s = signal(1);
    effect(() => log.push(s()));
    s.set(2);
    s.set(3);
    expect(log).toEqual([1, 2, 3]);
  });
});
```

Run tests:

```
ng test
```

Step 4: Integration Test with DOM Using Testing Library

Create `src/app/counter.component.spec.ts`:

```
import { render, screen } from '@testing-library/angular';
import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  it('should display initial count and double', async () => {
    await render(CounterComponent);
    expect(screen.getByText(/Count: 0/)).toBeTruthy();
    expect(screen.getByText(/Double: 0/)).toBeTruthy();
  });

  it('should increment count when button is clicked', async () => {
    await render(CounterComponent);
    const button = screen.getByText(/Increment/);
    button.click();
    expect(screen.getByText(/Count: 1/)).toBeTruthy();
    expect(screen.getByText(/Double: 2/)).toBeTruthy();
  });
});
```

Step 5: BONUS – Test Component with Signal Input

If you have a child component like:

```
@Component({
  standalone: true,
  selector: 'app-child',
  template: `<p>Signal value: {{ inputSignal() }}</p>`,
})
export class ChildComponent {
  @Input({ signal: true }) inputSignal!: Signal<number>;
}
```

Then test it like this:

```
await render(ChildComponent, {
  componentInputs: {
    inputSignal: signal(42)
  }
});
expect(screen.getByText(/Signal value: 42/)).toBeTruthy();
```

What You Learned

Concept	Tool Used
`signal()` and `.update()`	Jasmine/Karma
`computed()`	Unit test validation
`effect()`	Logged results, observed behavior
DOM reactivity	Angular Testing Library
Signal inputs	Mocked with `componentInputs`

Bonus Challenge

- Add a **timer** using `effect()` and `setInterval()` with `cleanup()`
- Test memory safety by **cleaning up effects** manually
- Create a shared `CounterService` using signals and test multiple consumers

Signals with SSR (Server Side Rendering)

1. How Signals Behave in SSR

Angular Signals are **synchronous and deterministic**, making them a natural fit for SSR. When rendering on the server:

- `signal()`, `computed()`, and `@Input({ signal: true })` work **predictably**
- Angular evaluates templates based on the current values of signals
- The result is a **fully rendered HTML DOM**, ready for hydration on the client

Benefits:

- No Zones required
- No asynchronous change detection
- Reusable and readable state snapshots

2. Best Practices for Hydration

Hydration = merging static HTML from the server with the live Angular app on the client (without re-rendering).

Best Practices

Practice	Why?
Use `provideClientHydration()`	Enables DOM reuse & reduces flicker
Use `ChangeDetectionStrategy.OnPush`	Prevents unnecessary template checks

Avoid using mutable global state	Ensures SSR output matches client startup
Render static routes for pre-render	Enables instant paint and crawlable HTML
Use `signal()` for page state	No Zone dependency, deterministic
Delay effects until after hydration	Avoid running client-side logic too early

Example

```
bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()],
});
```

This is required for hydration compatibility with signals.

3. Using `toSignal()` in Server-Rendered Templates

`toSignal()` bridges `Observable` to `Signal` – often used for:

- Data fetched via `HttpClient`
- Route parameters (`ActivatedRoute.params`)
- `BehaviorSubject` in services

Example (Safe SSR pattern):

```
@Component({
  template: `<p *ngIf="user() as u">Hello, {{ u.name }}</p>`
})
export class UserComponent {
  user = toSignal(this.userService.user$); // converts observable to signal
```

```
}
```

- **SSR-safe** because it captures the latest value synchronously
- Be careful with `async` observables during SSR — they may not emit in time

4. Challenges with `effect()` and SSR Lifecycle

Problem:

- `effect()` runs eagerly at setup time
- During SSR, you may **not want effects to run**
- Effects might reference `window`, `localStorage`, or APIs not available server-side

Anti-Pattern:

```
effect(() => {
  localStorage.setItem('theme', theme());
});
```

This fails during SSR: `localStorage is not defined`

Mitigation Strategies

Strategy	Explanation
Use `isPlatformBrowser()`	Check environment before running code
Use `inject(PLATFORM_ID)`	Server/client context detection
Defer `effect()` until `ngOnInit()`	Helps run effects only on client

Use `setTimeout` or `requestIdleCallback`	Delay until browser is ready
Use `cleanup()` inside `effect()`	Clean up timers or side effects

Example:

```
constructor() {
  if (isPlatformBrowser(this.platformId)) {
    effect(() => {
      console.log('Signal value:', this.someSignal());
    });
  }
}
```

Summary Table

Feature	Signal SSR Support	Notes
`signal()`	Yes	Fully synchronous during SSR
`computed()`	Yes	Derived values resolved before render
`effect()`	With caution	Avoid side effects during server render
`toSignal()`	Yes	Safe for bridging observables to templates

Hydration	Yes	Works with `provideClientHydration()`
Lifecycle (`ngOnInit`, etc.)	Post-hydration	SSR runs without lifecycle hooks

Developer Tips

- **Use signals for all app state** if possible
- **Guard effects** with platform checks (`isPlatformBrowser`)
- **Pre-render static routes** with `ng run app:prerender`
- **Use hydration and avoid full DOM rebuilds**
- **Avoid any `window`, `document`, or browser-only APIs** in global signal logic

Lab: Angular Signals with SSR — Hydration, `toSignal()`, and Lifecycle

Objectives

By the end of this lab, you will:

- Set up SSR and hydration using Angular Universal
- Use `signal()`, `computed()`, and `toSignal()` in a server-rendered template
- Test hydration with real HTML output
- Handle SSR lifecycle issues with `effect()` safely

Step 1: Create a New Angular App

```
ng new signal-ssr-lab --standalone --routing --style=css
cd signal-ssr-lab
```

Step 2: Add Angular SSR

```
ng add @angular/ssr
```

This will generate:

- `server.ts`, `app.server.module.ts`
- Preconfigured Express server and hydration setup

Step 3: Enable Hydration

Open `src/main.ts` and ensure this line is present:

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
```

```
  providers: [provideClientHydration()]
});
```

Step 4: Create a Component Using Signals

```
ng generate component pages/dashboard --standalone
```

Update `dashboard.component.ts`:

```
import { Component, signal, computed } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  standalone: true,
  selector: 'app-dashboard',
  imports: [CommonModule],
  template: `
    <h2>📊 Server-Rendered Dashboard</h2>
    <p>Counter: {{ counter() }}</p>
    <p>Double: {{ double() }}</p>
    <button (click)="increment()">Increment</button>
  `
})
export class DashboardComponent {
  counter = signal(1);
  double = computed(() => this.counter() * 2);

  increment() {
    this.counter.update(c => c + 1);
  }
}
```

Step 5: Use `toSignal()` for Observable Bridge

Edit `dashboard.component.ts`:

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

export class DashboardComponent {
  // Existing signals...
  now = toSignal(interval(1000), { initialValue: 0 }); // Updates every second
}
```

Update the template:

```
<p>Now: {{ now() }}</p>
```

- This bridges an observable (e.g., polling or streaming) into a server-compatible signal.

Step 6: Add Safe `effect()` Usage

```
import { effect, inject, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

export class DashboardComponent {
  // Existing signals...

  constructor() {
    const platformId = inject(PLATFORM_ID);

    if (isPlatformBrowser(platformId)) {
      effect(() => {
        console.log('[Browser Only] Counter changed:', this.counter());
      });
    }
  }
}
```

- This avoids executing `effect()` logic during SSR.

Step 7: Build and Serve SSR App

```
npm run build:ssr
npm run serve:ssr
```

Visit: http://localhost:4000

- Open **View Page Source** — you should see full HTML (hydrated content)
- Open **DevTools > Console** — logs appear only on the client (not SSR)

Bonus: Simulate Error

Try moving `effect()` outside the platform check and reload the server:

```
effect(() => {
  console.log(localStorage.getItem('theme'));
});
```

- You should get an error like `ReferenceError: localStorage is not defined` — this demonstrates **why guards are necessary** in SSR.

Summary of What You've Done

Feature	Completed?
Added SSR and hydration	Yes
Used `signal()` and `computed()`	Yes
Used `toSignal()` for observable	Yes
Implemented safe `effect()`	Yes
Verified HTML pre-rendering	Yes

Challenge (Optional)

- Create a service with a `signal()` and inject it into multiple components
- Pre-render routes with `ng run app:prerender`
- Use `cleanup()` inside an `effect()` tied to `setInterval`

NgRx Signals Store – Modern State Management with Angular Signals

1. Overview of NgRx Signals Store

NgRx Signals Store is a lightweight and reactive state management approach introduced as part of the modernized NgRx ecosystem (v16+), built around Angular's `signal()` primitive.

Purpose:

- Replace boilerplate-heavy reducers/actions/effects with simple `signal()`-based stores.
- Improve **developer experience** and **runtime performance**.
- Align NgRx more closely with Angular's native reactivity model.

2. Signal-based Selectors and Stores

Instead of `createReducer()` + `createAction()` + `createSelector()`, you define a **store class** using `signal()`, `computed()`, and methods to mutate state.

Example: `CounterStore`

```
import { Injectable, computed, signal } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class CounterStore {
  private count = signal(0);

  readonly double = computed(() => this.count() * 2);

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

  getCount() {
    return this.count;
  }
}
```

Usage in Component:

```
@Component({ ... })
export class CounterComponent {
  readonly count = this.counterStore.getCount();
  readonly double = this.counterStore.double;

  constructor(private counterStore: CounterStore) {}

  inc() {
    this.counterStore.increment();
  }
}
```

- Benefits:
  - No reducers
  - No action types
  - No need to wire up selectors or feature slices

3. Migration from Traditional NgRx to Signals-based Approach

Traditional NgRx	NgRx Signals Store
`createAction()`	Replace with store method (e.g. `set()`)
`createReducer()`	Replace with `signal()` state variable
`createSelector()`	Replace with `computed()`
`dispatch(action)`	Replace with method call (`increment()`)

`select(state => ...)`	Access signal directly in template

Migration Steps:

1. Identify simple slices (e.g. counter, toggle, UI state)
2. Create a new StoreService using `signal()`/`computed()`
3. Replace `select()` calls in components with direct signal usage
4. Phase out `actions.ts`, `reducer.ts`, `selectors.ts` if no longer needed

You can still use **traditional reducers and actions** for **complex workflows**, and use Signals Store for **simple state**.

4. When to Choose Signals Store vs Full NgRx

Scenario	Recommended Approach
Simple UI State (e.g., toggles, counters)	NgRx Signals Store
Local or Feature Module State	NgRx Signals Store
Global Shared State (auth, cart, etc.)	Either (based on scale)
Complex Side Effects (API, retry, debounce, etc.)	Full NgRx + Effects
You already use `signal()` extensively	Use Signals Store

Need time-travel debugging or DevTools	Full NgRx
Want minimal learning curve	Signals Store

Summary

Concept	Explanation
NgRx Signals Store	Angular-native state management with `signal()`
Selectors replaced by	`computed()`
Dispatch replaced by	Direct method calls (e.g., `store.increment()`)
Works great with	Angular Standalone Components + OnPush
Best for	Small to mid-sized apps, UI-level state
Still use classic NgRx for	Enterprise-wide state, orchestrating complex effects

Lab: Build and Use NgRx Signals Store in an Angular App

Objectives

By the end of this lab, you will:

- Create a lightweight `SignalStore` using Angular's `signal()` and `computed()`
- Use the store in a standalone component
- Replace traditional NgRx boilerplate (action, reducer, selector)
- Understand when and how to use Signals Store

Step 1: Create New Angular Standalone App

```
ng new signals-store-lab --standalone --routing --style=css
cd signals-store-lab
```

Step 2: Create a Signals Store

Create a file `src/app/stores/counter.store.ts`:

```
import { Injectable, signal, computed } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class CounterStore {
  private count = signal(0);

  readonly double = computed(() => this.count() * 2);

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

  decrement() {
    this.count.update(c => c - 1);
  }

  reset() {
```

```
    this.count.set(0);
  }

  getCountSignal() {
    return this.count;
  }
}
```

Step 3: Create a Counter Component to Consume the Store

```
ng generate component counter --standalone
```

Update `counter.component.ts`:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CounterStore } from '../stores/counter.store';

@Component({
  selector: 'app-counter',
  standalone: true,
  imports: [CommonModule],
  template: `
    <h2> Counter</h2>
    <p>Count: {{ count() }}</p>
    <p>Double: {{ double() }}</p>
    <button (click)="store.increment()">Increment</button>
    <button (click)="store.decrement()"> Decrement</button>
    <button (click)="store.reset()">Reset</button>
  `
})
export class CounterComponent {
  count = this.store.getCountSignal();
  double = this.store.double;

  constructor(public store: CounterStore) {}
}
```

Step 4: Route the Component in `app.routes.ts`

```
import { Routes } from '@angular/router';
import { CounterComponent } from './counter/counter.component';

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

- Visit http://localhost:4200 and test the UI.

Step 5: Compare to Classic NgRx (Optional)

Create a dummy `counter.actions.ts`, `counter.reducer.ts`, and compare:

Traditional NgRx	**Signals Store**
`createAction()`	Store methods (`increment()`)
`createReducer()`	`signal()` variable (`count`)
`createSelector()`	`computed()` (`double`)
`dispatch(action)`	Direct call to `store.method()`
`select(state => …)`	Access `store.signal()` in template

Summary: What You Built

Feature	**Completed**

Signals-based state with `signal()`	Yes
Derived state with `computed()`	Yes
Full feature store with methods	Yes
Component binding to signals	Yes

Bonus Challenge

- Create a `TodoStore` with `todos = signal<Todo[]>()`
- Add filtering logic with `computed()` (e.g., `completedTodos`)
- Allow toggling todos from the component

Web Components Integration with Angular Signals

Angular Elements • Signals in Web Components • Cross-Framework Interop

1. Angular Elements with Standalone Components

Angular allows packaging components as **custom elements** (aka **Web Components**) via the `@angular/elements` package.

Why Use Angular Elements?

- Embed Angular components in **non-Angular apps** (e.g., React, Vue, static HTML)
- Isolate Angular logic for **micro frontends**
- Deliver reusable UI modules via web standards

Setup Overview

```
npm install @angular/elements
```

Then convert a **standalone component** into a custom element:

```
import { createCustomElement } from '@angular/elements';
import { bootstrapApplication } from '@angular/platform-browser';

bootstrapApplication(MyComponent).then(appRef => {
  const element = createCustomElement(MyComponent, { injector: appRef.injector });
  customElements.define('my-widget', element);
});
```

- Angular 14+ supports `standalone: true` for easier packaging.

2. Using Signals in Angular Elements

Angular Signals are **perfectly compatible** with Angular Elements, since they are:

- **Self-contained** (state + logic)
- **Reactive** without zones or change detection overhead

- **Composable** via `signal()`, `computed()`, `effect()`

Example: Signal-based Counter Element

```
@Component({
  selector: 'signal-counter',
  standalone: true,
  template: `
    <p>Count: {{ count() }}</p>
    <button (click)="count.update(v => v + 1)">+</button>
  `
})
export class SignalCounterComponent {
  count = signal(0);
}
```

- Package `SignalCounterComponent` as a custom element and reuse it anywhere (even outside Angular).

3. Embedding Signal-based Angular Components in Non-Angular Apps

You can now embed your Signal-powered Angular components inside:

- **Static HTML pages**
- **React/Vue/Preact** apps
- **Microfrontend shells (e.g., Webpack Module Federation)**

Example in HTML:

```
<html>
  <body>
    <signal-counter></signal-counter>
    <script src="signal-widget.js"></script>
  </body>
</html>
```

- The JavaScript bundle (via Angular CLI `output-hashing: false`) will register the component automatically.

4. Communication Between Web Components and Angular Components

- **Input Binding via Attributes / Properties**

```
@Input({ alias: 'label', transform: v => v.toUpperCase() })
label = signal('Default');
```

In HTML:

```
<signal-button label="Click Me"></signal-button>
```

- **Output Communication**

Use custom events:

```
@Output() clicked = new EventEmitter<void>();

<button (click)="clicked.emit()">Click</button>
```

Or dispatch manually:

```
this.elRef.nativeElement.dispatchEvent(new CustomEvent('my-event', { detail: { value: 42 } }));
```

Signals → DOM Update = (auto)

DOM updates automatically react to signal changes — **no extra change detection** needed.

Best Practices

Practice	Why It Matters
Use `standalone` components	Easier packaging, fewer dependencies
Use `signal()` for internal state	Keeps element reactive without `NgZone`
Define custom `@Input({ signal: true })`	Reactive props from host

Emit DOM events via `@Output()`	Enables interop with parent apps
Keep bundle self-contained	Avoid dependencies outside component scope
Lazy load element module if possible	Smaller footprint

Summary

Topic	Supported?	Notes
Standalone Component → Custom Element	Yes	Use `@angular/elements`
`signal()` in custom element	Yes	Fully compatible with DOM updates
Use in React/Vue/static HTML	Yes	No Angular host needed
Input/Output bindings	Yes	via attributes and CustomEvents
Interop-safe, reactive UI	Yes	Works across frameworks

Web Components Integration with Angular Signals

Angular Elements • Signals in Web Components • Cross-Framework Interop

1. Angular Elements with Standalone Components

Angular allows packaging components as **custom elements** (aka **Web Components**) via the `@angular/elements` package.

Why Use Angular Elements?

- Embed Angular components in **non-Angular apps** (e.g., React, Vue, static HTML)
- Isolate Angular logic for **micro frontends**
- Deliver reusable UI modules via web standards

Setup Overview

```
npm install @angular/elements
```

Then convert a **standalone component** into a custom element:

```
import { createCustomElement } from '@angular/elements';
import { bootstrapApplication } from '@angular/platform-browser';

bootstrapApplication(MyComponent).then(appRef => {
  const element = createCustomElement(MyComponent, { injector: appRef.injector });
  customElements.define('my-widget', element);
});
```

- Angular 14+ supports `standalone: true` for easier packaging.

2. Using Signals in Angular Elements

Angular Signals are **perfectly compatible** with Angular Elements, since they are:

- **Self-contained** (state + logic)
- **Reactive** without zones or change detection overhead

- **Composable** via `signal()`, `computed()`, `effect()`

Example: Signal-based Counter Element

```
@Component({
  selector: 'signal-counter',
  standalone: true,
  template: `
    <p>Count: {{ count() }}</p>
    <button (click)="count.update(v => v + 1)">+</button>
  `
})
export class SignalCounterComponent {
  count = signal(0);
}
```

- Package `SignalCounterComponent` as a custom element and reuse it anywhere (even outside Angular).

3. Embedding Signal-based Angular Components in Non-Angular Apps

You can now embed your Signal-powered Angular components inside:

- **Static HTML pages**
- **React/Vue/Preact** apps
- **Microfrontend shells (e.g., Webpack Module Federation)**

Example in HTML:

```
<html>
  <body>
    <signal-counter></signal-counter>
    <script src="signal-widget.js"></script>
  </body>
</html>
```

- The JavaScript bundle (via Angular CLI `output-hashing: false`) will register the component automatically.

---

4. Communication Between Web Components and Angular Components

Input Binding via Attributes / Properties

```
@Input({ alias: 'label', transform: v => v.toUpperCase() })
label = signal('Default');
```

In HTML:

```
<signal-button label="Click Me"></signal-button>
```

Output Communication

Use custom events:

```
@Output() clicked = new EventEmitter<void>();

<button (click)="clicked.emit()">Click</button>
```

Or dispatch manually:

```
this.elRef.nativeElement.dispatchEvent(new CustomEvent('my-event', { detail: { value: 42 } }));
```

Signals → DOM Update = (auto)

DOM updates automatically react to signal changes — **no extra change detection** needed.

Best Practices

Practice	Why It Matters
Use `standalone` components	Easier packaging, fewer dependencies
Use `signal()` for internal state	Keeps element reactive without `NgZone`

Define custom `@Input({ signal: true })`	Reactive props from host
Emit DOM events via `@Output()`	Enables interop with parent apps
Keep bundle self-contained	Avoid dependencies outside component scope
Lazy load element module if possible	Smaller footprint

Summary

Topic	Supported?	Notes
Standalone Component → Custom Element	Yes	Use `@angular/elements`
`signal()` in custom element	Yes	Fully compatible with DOM updates
Use in React/Vue/static HTML	Yes	No Angular host needed
Input/Output bindings	Yes	via attributes and CustomEvents
Interop-safe, reactive UI	Yes	Works across frameworks

Lab: Implement SSR Hydration with Signal-Based Data in Angular

Objectives

By the end of this lab, you will:

- Set up Server-Side Rendering (SSR) with Angular Universal
- Use Angular Signals for data state
- Convert `Observable` to `Signal` with `toSignal()`
- Enable hydration for seamless client-side reactivity
- Safely handle `effect()` in server and browser contexts

Step 1: Create a Standalone Angular App

```
ng new signal-ssr-hydration --standalone --routing=false --style=css
cd signal-ssr-hydration
```

Step 2: Add Angular SSR

```
ng add @angular/ssr
```

This adds:

- `server.ts`
- `main.server.ts`
- `app.server.module.ts`

Step 3: Enable Hydration

Open `src/main.ts` and update:

```
import { provideClientHydration } from '@angular/platform-browser';

bootstrapApplication(AppComponent, {
  providers: [provideClientHydration()]
});
```

- This enables DOM hydration after SSR.

Step 4: Create a Service to Fetch Data

Create `src/app/user.service.ts`:

```
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class UserService {
  private http = inject(HttpClient);

  getUser(): Observable<any> {
    return this.http.get('https://jsonplaceholder.typicode.com/users/1');
  }
}
```

Step 5: Convert Observable to Signal

Update `src/app/app.component.ts`:

```
import { Component, inject, signal } from '@angular/core';
import { toSignal } from '@angular/core/rxjs-interop';
import { UserService } from './user.service';
import { CommonModule } from '@angular/common';
import { HttpClientModule } from '@angular/common/http';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, HttpClientModule],
  template: `
    <h1>Signal-based SSR Hydration</h1>
    <div *ngIf="user() as u">
```

```
    <p><strong>Name:</strong> {{ u.name }}</p>
    <p><strong>Email:</strong> {{ u.email }}</p>
  </div>
 `
})
export class AppComponent {
 private userService = inject(UserService);

 // SSR-safe conversion from Observable to Signal
 user = toSignal(this.userService.getUser(), { initialValue: null });
}
```

- This setup works on both server and client using `toSignal()`.

Step 6: Avoid Effects on Server (Optional but recommended)

```
import { effect, inject, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

constructor() {
 const platformId = inject(PLATFORM_ID);

 if (isPlatformBrowser(platformId)) {
  effect(() => {
   console.log('User loaded:', this.user());
  });
 }
}
```

Step 7: Build and Serve the SSR App

```
npm run build:ssr
npm run serve:ssr
```

Visit: http://localhost:4000

Test Hydration:

- **View Page Source** – confirms user data is server-rendered
- **Open Console** – confirms rehydration and `signal()` continues working without a flicker

Bonus Challenge

- Add a `PostService` and display a list of posts using `toSignal()`
- Use `computed()` to filter posts
- Add a timer with `signal()` + `effect()` and hydrate time-based state

Lab: Build a Mini NgRx Signals Store in Angular

Objectives

By the end of this lab, you will:

- Build a lightweight state management service using Angular's `signal()`, `computed()`, and component bindings
- Simulate a mini NgRx-like store using signals
- Replace selectors and actions with computed values and methods
- Structure the store for maintainability and reuse

Step 1: Create a New Angular App

```
ng new mini-signal-store --standalone --routing=false --style=css
cd mini-signal-store
```

Step 2: Create the Mini Store (`CounterStore`)

Create a file: `src/app/stores/counter.store.ts`

```
import { Injectable, signal, computed } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class CounterStore {
  private _count = signal(0);

  // Selectors (computed values)
  readonly count = this._count.asReadonly();
  readonly double = computed(() => this._count() * 2);
  readonly isEven = computed(() => this._count() % 2 === 0);

  // Actions (mutators)
  increment() {
    this._count.update(c => c + 1);
  }

  decrement() {
    this._count.update(c => c - 1);
```

```
  }

  reset() {
    this._count.set(0);
  }
}
```

Step 3: Create a UI Component to Bind the Store

```
ng generate component counter --standalone --skip-tests
```

Update `counter.component.ts`:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CounterStore } from '../stores/counter.store';

@Component({
  selector: 'app-counter',
  standalone: true,
  imports: [CommonModule],
  template: `
    <h2> Mini NgRx Signals Store</h2>
    <p>Count: {{ store.count() }}</p>
    <p>Double: {{ store.double() }}</p>
    <p>Even? {{ store.isEven() ? 'Yes' : 'No' }}</p>

    <button (click)="store.increment()">➕</button>
    <button (click)="store.decrement()">➖</button>
    <button (click)="store.reset()"> Reset</button>
  `
})
export class CounterComponent {
  constructor(public store: CounterStore) {}
}
```

Step 4: Wire Up the App Component

Replace `src/app/app.component.ts`:

```
import { Component } from '@angular/core';
import { CounterComponent } from './counter/counter.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CounterComponent],
  template: `<app-counter />`,
})
export class AppComponent {}
```

Step 5: Run the App

```
ng serve
```

Navigate to: http://localhost:4200

- You should see a working signal-based counter with reactive bindings.

Bonus: Add a Derived Signal Array

Extend `CounterStore` with a signal list of actions:

```
private _history = signal<string[]>([]);

readonly history = this._history.asReadonly();

increment() {
  this._count.update(c => c + 1);
  this._history.update(h => [...h, 'Increment']);
}
```

Then render `history()` in your template.

Lab: Export a Signal-Powered Angular Component as a Web Component

Objectives

By the end of this lab, you will:

- Create a standalone Angular component using `signal()`
- Export it as a native Web Component (custom element) using `@angular/elements`
- Embed and interact with it inside a plain HTML page (non-Angular environment)
- Handle `@Input()` and `@Output()` bindings in a web-native way

Step 1: Create a New Angular App

```
ng new signal-web-component --standalone --routing=false --style=css
cd signal-web-component
```

Step 2: Install Angular Elements & Zone.js

```
npm install @angular/elements zone.js
```

Step 3: Create a Signal-Powered Component

```
ng generate component signal-button --standalone
```

Update `signal-button.component.ts`:

```
import { Component, EventEmitter, Output, signal, Input } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'signal-button',
  standalone: true,
  imports: [CommonModule],
```

```
  template: `
    <div style="border:1px solid #ccc; padding:1rem; border-radius:8px;">
      <p>{{ label() }}</p>
      <p>Clicks: {{ count() }}</p>
      <button (click)="increment()">Click Me</button>
    </div>
  `
})
export class SignalButtonComponent {
  @Input({ alias: 'label', required: false }) label = signal('Click Counter');
  @Output() clicked = new EventEmitter<number>();

  count = signal(0);

  increment() {
    this.count.update(n => n + 1);
    this.clicked.emit(this.count());
  }
}
```

Step 4: Register the Component as a Web Component

Replace `src/main.ts` with:

```
import { createCustomElement } from '@angular/elements';
import { bootstrapApplication } from '@angular/platform-browser';
import { SignalButtonComponent } from './app/signal-button/signal-button.component';

bootstrapApplication(SignalButtonComponent).then(appRef => {
  const element = createCustomElement(SignalButtonComponent, {
    injector: appRef.injector,
  });
  customElements.define('signal-button', element);
});
```

- This registers `<signal-button>` as a native custom element.

Step 5: Add Static HTML Host File

Create a new file `src/host.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
  <title>Signal Web Component Demo</title>
</head>
<body>
  <h2>Signal-powered Angular Web Component</h2>

  <signal-button label="Signal Rocks!"></signal-button>

  <script>
    const el = document.querySelector('signal-button');
    el.addEventListener('clicked', e => {
      console.log('[Web Component] Clicked count:', e.detail);
    });
  </script>

  <script src="main.js"></script>
</body>
</html>
```

Step 6: Modify Angular Build Output

Edit `angular.json`:

```
"outputHashing": "none",
"index": "src/host.html"
```

- This ensures Angular builds `main.js` without a hash so you can include it in plain HTML.

Step 7: Build and Serve

```
ng build
npx http-server dist/signal-web-component/browser
```

Navigate to: http://localhost:8080/host.html

- You should see your web component rendered in plain HTML, and click events logged to the console.

Bonus Challenges

- Add a `@Input({ signal: true })` called `step`, and increment by `step` instead of `1`

- Add styles scoped inside the component
- Deploy the component to a CDN and use it in an external HTML file