**Chapter 5: Services & Hooks**

Think of services as **reusable pieces of logic or data** that you want to share across components or parts of your app. They handle things like:

* Fetching data from APIs
* Managing state
* Business logic separate from UI

Services keep your components lean and focused only on presenting data.

**Dependency Injection (DI) — How Angular wires it all up**

Angular uses **Dependency Injection** to provide instances of services to components or other services automatically. Instead of manually creating service instances, Angular **injects** them where needed.

This makes your code:

* Easier to maintain
* Testable
* Loosely coupled

**Creating a Service**

**How to create?**

Use the Angular CLI command:

ng generate service services/user

This creates:

* user.service.ts (the service class)
* user.service.spec.ts (unit test file)

**Basic Service Example**

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

@Injectable({

providedIn: 'root' // Service available app-wide

})

export class UserService {

private users = ['Alice', 'Bob', 'Charlie'];

getUsers() {

return this.users;

}

}

**Using inject() instead of constructor injection (Angular 19+)**

Angular 19 introduced a handy way to inject dependencies **without a constructor** using inject().

Example:

@Component({

standalone: true,

selector: 'app-user-list',

template: `

<h3>Users:</h3>

<ul>

<li \*ngFor="let user of users">{{ user }}</li>

</ul>

`

})

export class UserListComponent {

private userService = inject(UserService);

users = this.userService.getUsers();

}

**When to provide a service?**

* Use providedIn: 'root' for **singleton** services shared across the app.
* Use providedIn: null and provide the service **in component providers array** if you want a **component-scoped** instance (a new instance per component).

**Example: Component-scoped service**

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

@Injectable()

export class CounterService {

count = 0;

increment() {

this.count++;

}

}

@Component({

standalone: true,

selector: 'app-counter',

template: `

<p>Count: {{ counterService.count }}</p>

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

`,

providers: [CounterService] // New instance per component

})

export class CounterComponent {

constructor(public counterService: CounterService) {}

}

**Real-world scenario:**

Suppose you want to share user authentication status and details throughout your app:

@Injectable({

providedIn: 'root'

})

export class AuthService {

private loggedIn = false;

private userName = '';

login(name: string) {

this.loggedIn = true;

this.userName = name;

}

logout() {

this.loggedIn = false;

this.userName = '';

}

isLoggedIn() {

return this.loggedIn;

}

getUserName() {

return this.userName;

}

}

And a component can inject and use it:

@Component({

standalone: true,

selector: 'app-login-status',

template: `

<div \*ngIf="auth.isLoggedIn(); else loggedOut">

Welcome, {{ auth.getUserName() }}!

<button (click)="auth.logout()">Logout</button>

</div>

<ng-template #loggedOut>

<button (click)="auth.login('Jane')">Login as Jane</button>

</ng-template>

`

})

export class LoginStatusComponent {

constructor(public auth: AuthService) {}

}

# Angular Lifecycle Hooks

Angular components go through a **series of lifecycle stages**: from being created, displayed, updated, and finally destroyed. **Lifecycle hooks** let you tap into these stages and run code at the right time.

Think of them like "events" that fire during your component’s life — similar to how a person is born, grows, does work, and eventually retires.

## Why Lifecycle Hooks Matter

They allow you to:

* Initialize data when the component loads
* Perform cleanup when the component is removed
* React to changes in inputs
* Access the DOM or child components
* Set up subscriptions or timers and clean them properly

## Common Lifecycle Hooks and Their Purpose

|  |  |  |
| --- | --- | --- |
| Hook | Called When... | Use For |
| ngOnInit() | Component is initialized (once) | Fetching data, setup logic |
| ngOnChanges() | @Input() properties change | Responding to input changes |
| ngAfterViewInit() | View (and child views) are rendered | Access DOM or @ViewChild() |
| ngOnDestroy() | Component is about to be destroyed | Unsubscribe, cleanup, timer kill |
| ngAfterContentInit() | Content projected with <ng-content> is ready | Access content children |
| ngAfterViewChecked() | After view or its children are checked (frequently) | DOM updates (rare use) |

## The Core Lifecycle Hooks in Practice

### A. ngOnInit()

**Called once** after the component is created.

Use this for:

* Fetching data from a service
* Starting timers
* Initial setup of state

#### Code snippet:

ngOnInit() {

this.fetchUser();

}

### B. ngOnChanges(changes: SimpleChanges)

Called **whenever @Input() properties** change (even on first load).

ngOnChanges(changes: SimpleChanges) {

console.log('Input changed:', changes['user']);

}

### C. ngAfterViewInit()

Fired after the component’s view and all child views have been initialized.

Use when you need:

* To access template elements via @ViewChild
* Run animation or layout logic after rendering

### D. ngOnDestroy()

Runs **before the component is destroyed**.

Use to:

* Unsubscribe from Observables
* Clear intervals or event listeners
* Clean up resources

ngOnDestroy() {

this.subscription.unsubscribe();

}

## Example: Lifecycle Hooks in Action

import {

Component, Input, OnInit, OnChanges,

SimpleChanges, OnDestroy, AfterViewInit

} from '@angular/core';

@Component({

standalone: true,

selector: 'app-lifecycle-demo',

template: `

<h3>Lifecycle Demo</h3>

<p>Name: {{ name }}</p>

`

})

export class LifecycleDemoComponent implements OnInit, OnChanges, AfterViewInit, OnDestroy {

@Input() name: string = '';

private intervalId: any;

constructor() {

console.log('Constructor called');

}

ngOnChanges(changes: SimpleChanges): void {

console.log('ngOnChanges:', changes);

}

ngOnInit(): void {

console.log('ngOnInit');

this.intervalId = setInterval(() => {

console.log('Interval running...');

}, 2000);

}

ngAfterViewInit(): void {

console.log('ngAfterViewInit');

}

ngOnDestroy(): void {

console.log('ngOnDestroy');

clearInterval(this.intervalId);

}

}

### To Test This:

Create a parent component and toggle the lifecycle component to observe console logs.

@Component({

standalone: true,

selector: 'app-root',

template: `

<button (click)="toggle()">Toggle Component</button>

<app-lifecycle-demo \*ngIf="show" [name]="userName"></app-lifecycle-demo>

`,

imports: [LifecycleDemoComponent]

})

export class AppComponent {

show = true;

userName = 'Alice';

toggle() {

this.show = !this.show;

}

}

## When to Use Each Hook

|  |  |
| --- | --- |
| Hook | When to Use |
| ngOnInit | Set up data when component loads |
| ngOnChanges | React to external input changes |
| ngAfterViewInit | DOM logic that depends on view initialization |
| ngOnDestroy | Clean up (timers, subscriptions) |

## Best Practices

* ✅ Use ngOnInit() for fetching data — **not** the constructor.
* ✅ Always **unsubscribe** in ngOnDestroy() to avoid memory leaks.
* ❌ Avoid using ngAfterViewChecked() unless absolutely needed (it runs very frequently).