# Chapter 4: Directives

Think of directives as special instructions or “magic helpers” in your templates. They **change how your HTML behaves or looks** based on your app’s logic.

There are two main types:

* **Structural Directives**: They change the layout by adding or removing parts of the DOM.
* **Attribute Directives**: They change the appearance or behavior of existing elements.

## 1. Structural Directives: Changing the DOM Layout

### *Import CommonModule*

imports: [CommonModule],

### A. \*ngIf — Show or hide elements

#### What it does

It decides if an element should be in the DOM at all, based on a condition.

#### When to use

* Show a loading spinner only when data is loading
* Display a welcome message only if the user is logged in

#### Snippet:

<div \*ngIf="isLoggedIn">Welcome back!</div>

#### Real-life example:

Show a message only if there are notifications.

<div \*ngIf="notifications.length > 0">

You have {{ notifications.length }} new notifications.

</div>

#### Complete example you can try:

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

@Component({

standalone: true,

imports: [CommonModule],

selector: 'app-ngif-example',

template: `

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

<div \*ngIf="isLoggedIn; else loggedOut">

<h2>Welcome back, user!</h2>

</div>

<ng-template #loggedOut>

<h2>Please log in to continue.</h2>

</ng-template>

`

})

export class NgIfExampleComponent {

isLoggedIn = false;

toggle() {

this.isLoggedIn = !this.isLoggedIn;

}

}

### B. \*ngFor — Loop through data lists

#### What it does

It repeats a chunk of template for each item in a list.

#### When to use

* Display a list of users or messages
* Render a table of products

#### Snippet:

<ul>

<li \*ngFor="let item of items">{{ item.name }}</li>

</ul>

#### Real-life example:

List user names fetched from an API.

<ul>

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

</ul>

#### Complete example you can try:

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

@Component({

standalone: true,

imports: [CommonModule],

selector: 'app-ngfor-example',

template: `

<h3>User List</h3>

<ul>

<li \*ngFor="let user of users; let i = index">

{{ i + 1 }}. {{ user.name }}

</li>

</ul>

`

})

export class NgForExampleComponent {

users = [

{ name: 'Alice' },

{ name: 'Bob' },

{ name: 'Charlie' }

];

}

### C. \*ngSwitch — Choose between multiple templates

#### What it does

It displays one template among many based on a switch expression.

#### When to use

* Show different UIs for user roles (admin, guest, member)
* Display different icons based on status

#### Snippet:

<div [ngSwitch]="userRole">

<p \*ngSwitchCase="'admin'">Admin Access</p>

<p \*ngSwitchCase="'guest'">Guest Access</p>

<p \*ngSwitchDefault>Member Access</p>

</div>

#### Complete example you can try:

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

@Component({

standalone: true,

imports: [CommonModule],

selector: 'app-ngswitch-example',

template: `

<label>

Select Role:

<select [(ngModel)]="userRole">

<option value="admin">Admin</option>

<option value="guest">Guest</option>

<option value="member">Member</option>

</select>

</label>

<div [ngSwitch]="userRole">

<p \*ngSwitchCase="'admin'">You have full admin access.</p>

<p \*ngSwitchCase="'guest'">You have guest access.</p>

<p \*ngSwitchDefault>You have member access.</p>

</div>

`,

imports: []

})

export class NgSwitchExampleComponent {

userRole = 'guest';

}

## 2. Attribute Directives: Changing Appearance & Behavior

### A. ngClass — Dynamic CSS classes

Adds or removes CSS classes dynamically based on conditions or expressions.

### B. ngStyle — Dynamic inline styles

Sets inline CSS styles dynamically.

## 3. Creating Custom Attribute Directives

Sometimes built-in directives don’t cut it, so you create your own!

### A. What’s a Custom Directive?

A directive that you build to change appearance or behavior of elements, reusing logic across your app.

### B. How to make one?

* Create a folder for custom directives
* ng g d <directive-name>

ng g d primary

* Use @Directive() decorator
* Inject ElementRef and/or Renderer2 to modify the host element

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

@Directive({

  selector: '[appPrimary]'

})

export class PrimaryDirective {

  constructor(e:ElementRef) {

    e.nativeElement.style.color = 'white';

    e.nativeElement.style.backgroundColor = 'blue';

  }

}

* Create a component
* Import the directive and use it as follows:

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

import { PrimaryDirective } from '../directives/primary.directive';

@Component({

  selector: 'app-p9-custom',

  imports: [PrimaryDirective],

  template: `

    <p appPrimary > p9-custom works! </p>

  `,

  styles: ``

})

export class P9CustomComponent {

}

### D. Another Example (with Renderer2):

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

@Directive({

standalone: true,

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private el: ElementRef, private renderer: Renderer2) {}

@HostListener('mouseenter') onMouseEnter() {

this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.renderer.removeStyle(this.el.nativeElement, 'backgroundColor');

}

}

### E. Usage example:

<p appHighlight>Hover over me to highlight!</p>

### F. Complete example you can try:

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

import { HighlightDirective } from './highlight.directive';

@Component({

standalone: true,

selector: 'app-directive-demo',

template: `

<p appHighlight>This paragraph highlights on hover.</p>

`,

imports: [HighlightDirective]

})

export class DirectiveDemoComponent {}

## 

## New Control Flow Syntax (Angular 17+)

### What’s new?

Angular v17 introduced a fresh, cleaner way to write control flow in templates using **@if**, **@for**, and **@switch** directives.

They are designed to **replace** the older structural directives (\*ngIf, \*ngFor, \*ngSwitch) with **improved syntax, better scoping, and performance**.

### Why care about this new syntax?

* **Cleaner, more intuitive templates**
* Variables declared inside blocks have limited scope (no accidental variable leaks)
* Easier to read and maintain complex UI logic
* Improved performance under the hood

### How to use?

You replace the old style:

<div \*ngIf="isLoggedIn; else loginPrompt">

Welcome back!

</div>

<ng-template #loginPrompt>

Please log in.

</ng-template>

with the new style:

@if (isLoggedIn) {

<div>Welcome back!</div>

} else {

<div>Please log in.</div>

}

### Detailed examples for each:

@if

@if (items.length > 0) {

<p>You have {{ items.length }} items.</p>

} else {

<p>No items found.</p>

}

@for

<ul>

@for (let item of items) {

<li>{{ item.name }}</li>

}

</ul>

@switch

@switch (status) {

@case ('loading') <p>Loading...</p>

@case ('error') <p>Error occurred.</p>

@default <p>Ready.</p>

}

### Scope & variable handling

Variables declared inside these blocks are **local** to the block — avoiding unintended access outside their scope. This reduces bugs and improves template clarity.

### Try this complete example:

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

@Component({

standalone: true,

selector: 'app-new-control-flow',

template: `

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

@if (isLoggedIn) {

<p>Welcome, user!</p>

} else {

<p>Please log in.</p>

}

<h3>Tasks:</h3>

<ul>

@for (let task of tasks) {

<li>{{ task }}</li>

}

</ul>

@switch (role) {

@case ('admin') <p>You have admin rights.</p>

@case ('guest') <p>Guest user.</p>

@default <p>Regular user.</p>

}

`

})

export class NewControlFlowComponent {

isLoggedIn = false;

tasks = ['Task 1', 'Task 2', 'Task 3'];

role = 'guest';

toggle() {

this.isLoggedIn = !this.isLoggedIn;

}

}