

# Angular Directives

What are Directives ?

Structural Directives

Attribute Directives

Pipes

---

# Directives

---

# What are Directives ?

Directives are similar to HTML attributes but have special behavior that extends the functionality of the elements they are applied to.

**Types of Directives:**

**Structural Directives:** Change the structure of the page.

**Attribute Directives:** Change the appearance or behavior of elements.

# Structural Directives

Structural directives create or remove HTML elements based on some conditions or data.

They usually start with an asterisk \* then the directive name like `*ngIf="condition"`

## Common Structural Directives:

**\*ngIf**: Displays an element only if a condition is true.

**\*ngFor**: Repeats an element for each item in a list.

**ngSwitch**: Displays one of many possible elements based on a matching condition.

# ngIf Directive

**ngIf** is used to show or hide an HTML element based on a condition.

**Syntax:**

**\*ngIf="condition"**, where **condition** is any expression that evaluates to true or false.

**Behavior:** If the condition is true, the element is added to the page. If false, the element is removed from the page.

**Access to Component Properties:** Inside the condition part, you can access any property of the component that the current template belongs to.

# ngIf Directive

```
// The app.component.ts file
@Component({
  selector: 'app-root',
  templateUrl:
    './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  isTaskDone = false;
  taskTitle = 'Task 1';
}
```

```
// The app.component.html file.
The task title will only be displayed
if the isTaskDone property is true.
If the isTaskDone property is false,
the heading will not be rendered.
<h1 *ngIf="isTaskDone">{{taskTitle}}
</h1>
```

# ngIf Else Block

To add an **else block** to an **\*ngIf** directive, you can use the **<ng-template>** element with a template reference variable defined on it using the **#** symbol followed by any name.

Then after the condition place a semicolon followed by the **else** keyword and the template reference variable name.

Any content inside the **<ng-template>** element will be rendered only if the condition in the **\*ngIf** directive is false.

```
<h1 *ngIf="isTaskDone; else MyElseBlock">{{ taskTitle }}</h1>
<ng-template #MyElseBlock>
  <h1>The task is not done yet</h1>
</ng-template>
```

# Grouping Elements

`<ng-container>` is a special element in Angular that **groups** multiple elements together in a template without rendering a real element in the DOM.

This can be used to avoid rendering redundant HTML elements when grouping multiple elements inside a condition.

The `div` tag is rendered in the DOM.

```
<div *ngIf="isTaskDone">
  <h1>{{ taskTitle }}</h1>
  <p>{{ taskDescription }}</p>
</div>
```

The `ng-container` tag is not rendered in the DOM. If the condition is true then only the content inside the `ng-container` tag is rendered.

```
<ng-container *ngIf="isTaskDone">
  <h1>{{ taskTitle }}</h1>
  <p>{{ taskDescription }}</p>
</ng-container>
```

# @if Block

Starting with Angular v17, you can replace the `*ngIf` directive with the `@if` block

```
@if (a > b) {  
<p>{{ a }} > {{ b }}</p>  
}
```

```
@if (a > b) {  
<p>{{ a }} > {{ b }}</p>  
} @else if (a < b) {  
<p>{{ a }} < {{ b }}</p>  
} @else {  
<p>{{ a }} = {{ b }}</p>  
}
```

# ngSwitch Directive

The **ngSwitch** directive is used to display one element from multiple options based on a matching condition using the syntax: **[ngSwitch]="condition"**

The ngSwitch directive value can be any property inside the component class.

```
<ng-container [ngSwitch]="userType">
```

The value of the **\*ngSwitchCase** directive can be a hardcoded value like **'Regular'** or a property of the component class like **premiumUser**.

```
<p *ngSwitchCase="'Regular'">No Discount</p>
```

```
<p *ngSwitchCase="premiumUser">10% Discount</p>
```

```
<p *ngSwitchDefault>Invalid User</p> The default case
```

```
</ng-container>
```

# @switch Block

Starting with Angular v17, you can replace the `*ngSwitch` directive with the `@switch` block

```
@switch (userType) {  
  @case ('Regular') {  
    <p>No Discount</p>  
  }  
  @case (premiumUser) {  
    <p>10% Discount</p>  
  }  
  @default {  
    <p>Invalid User</p>  
  }  
}
```

# ngFor Directive

```
export class AppComponent {  
  items = ['Apple', 'Banana', 'Orange'];  
}
```

The **ngFor** directive is used to repeat an HTML element for each item in a given list or array using the syntax

**\*ngFor**="let item of items"

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



Rendered DOM

```
<ul>  
  <li>Apple</li>  
  <li>Banana</li>  
  <li>Orange</li>  
</ul>
```

# ngFor Directive Variables Usage

Variable	Description
<b>index</b>	Current index (0, 1, 2, ...)
<b>first</b>	true for the first item
<b>last</b>	true for the last item
<b>odd</b>	true for odd index numbers
<b>even</b>	true for even index numbers

```
export class AppComponent {  
  items = ['Apple', 'Banana', 'Orange'];  
}
```

```
<ul>  
  <li *ngFor="let item of items; let i = index; let isFirst = first; let isLast = last">  
    {{ i + 1 }}. {{ item }}  
    <span *ngIf="isFirst">First</span>  
    <span *ngIf="isLast">Last</span>  
  </li>  
</ul>
```



Rendered DOM

```
1. Apple  First  
2. Banana  
3. Orange  Last
```

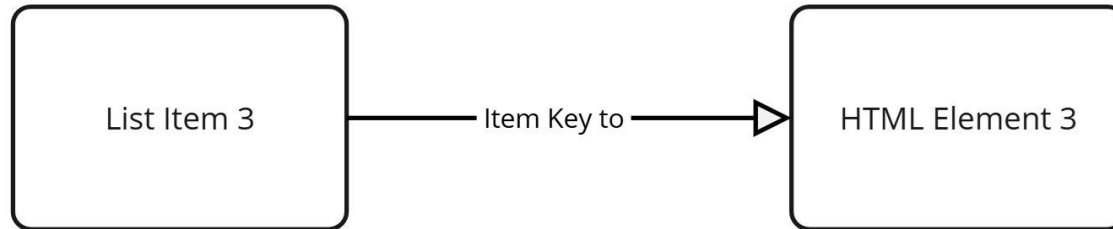
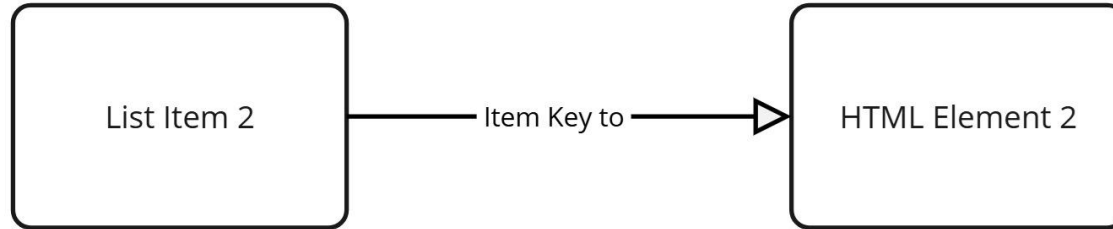
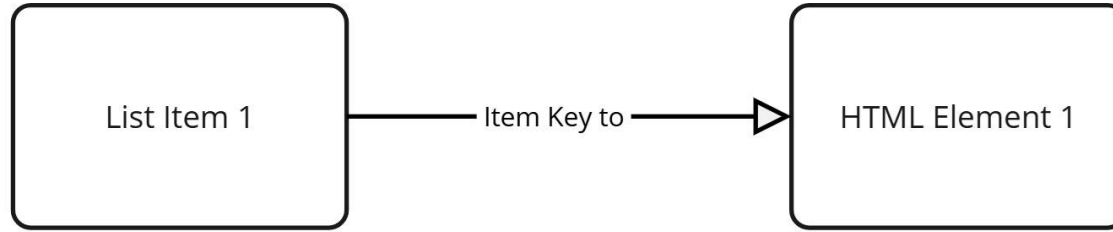
# List Item to Element Mapping – The Problem

When rendering a list using `*ngFor`, each item in the list is associated with an HTML element on the page.

**Default Mapping:** Angular uses the item's memory reference (object identity) as a key to map each item to its corresponding HTML element.

When using `*ngFor`, Angular recreates the entire list when:

- An item is added or removed.
- The array is replaced (e.g., from an API).
- This is **bad for performance** because: The entire list re-renders, even if only one item changes.
- Any state within the HTML elements (like user input or component state) is lost when elements are re-created.



# List Item to Element Mapping

When `updateUsers()` is called:

- Angular destroys all existing `<li>` elements.
- Then it creates new ones from scratch.
- Any user input inside `<li>` (like a text field) is lost.

```
export class AppComponent {  
  users = [  
    { id: 1, name: 'Alice', age: 25 },  
    { id: 2, name: 'Bob', age: 30 },  
  ];  
  
  updateUsers() {  
    this.users = [  
      { id: 1, name: 'Alice', age: 26 }, // Age updated  
      { id: 2, name: 'Bob', age: 30 },  
      { id: 3, name: 'Charlie', age: 22 }, // New user added  
    ];  
  }  
}
```

```
<ul>  
  <li *ngFor="let user of users">  
    {{ user.name }} ({{ user.age }})  
  </li>  
</ul>  
<button (click)="updateUsers()">Update List</button>
```



# ngFor **trackBy** Property for Performance Optimization

**Custom Tracking:** trackBy allows you to specify a **function that returns a custom identifier** (like an id property) for each item instead of using the memory reference as a key. For Example:

```
*ngFor="let note of notes; let index = index; trackBy: trackByFn"
```

```
trackByFn(index: number, note: Note) { return note.id; }
```

- Angular uses the provided identifier to map a list item to its corresponding HTML element.
- Even if the item's memory reference changes, as long as the identifier remains the same, Angular considers it the same item
- The item's new data is bound to the existing HTML element in the page.

# ngFor trackBy

## When updateUser() is called:

Normally, without trackBy, it would destroy and recreate all elements.

But with trackBy, it compares each user's ID and only updates necessary changes.

## Angular Calls the trackBy Function

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

Angular checks each item's id:

- Alice (ID: 1) → Exists → Updates age from 25 → 26
- Bob (ID: 2) → Exists → No changes, DOM remains the same
- Charlie (ID: 3) → New user, so Angular adds a new <li>

```
export class AppComponent {  
  users = [  
    { id: 1, name: 'Alice', age: 25 },  
    { id: 2, name: 'Bob', age: 30 },  
  ];  
  
  updateUser() {  
    this.users = [  
      { id: 1, name: 'Alice', age: 26 }, // Age updated  
      { id: 2, name: 'Bob', age: 30 },  
      { id: 3, name: 'Charlie', age: 22 }, // New user added  
    ];  
  }  
  
  trackById(index: number, user: any): number {  
    return user.id; // Unique identifier  
  }  
}
```

```
<ul>  
  <li *ngFor="let user of users; trackBy: trackById">  
    {{ user.name }} ({{ user.age }})  
  </li>  
</ul>  
<button (click)="updateUser()">Update List</button>
```

- Alice (25)
- Bob (30)

Update List

- Alice (26) ✓ (Updated, not recreated)
- Bob (30) (Not touched)
- Charlie (22) + (Newly Created)

Update List

# @for Block

Starting with Angular v17, you can replace the **\*ngFor** directive with the **@for** block

To specify the identifier used for tracking the list items, use the **track** keyword followed by the the variable you want to use, For example: **item.id** this is similar to a **trackBy** function that returns **item.id**

When accessing built in variables like the **index** add a **\$** sign before the variable name.

```
@for (item of items; track item.id; let idx = $index) {  
  <p>Item #{{ idx }}: {{ item.name }}</p>  
}
```

# Attribute Directives

**Attribute Directives** Change the appearance or behavior of elements.

The **ngClass** directive is used to dynamically add or remove CSS classes on an element.

It applies classes based on an expression that evaluates to a **string**, an **array**, or an **object**, these values can be defined directly in the template or reference properties within the component class

# ngClass Directive

**String:** When ngClass is set to a string, it adds one or more classes separated by a space.

Example: `<div [ngClass]='active'></div>` will add the active class.

Rendered as

```
<div class='active'>
</div>
```

**Array:** When ngClass is set to an array, it adds multiple classes listed in the array.

Example: `<div [ngClass]='[ 'active', 'highlight' ]'></div>` adds both active and highlight classes to the element.

Rendered as

```
<div class='active highlight'>
</div>
```

# ngClass Directive

The value of **ngClass** can be defined directly in the template or reference properties within the component class. The value of **classesArray** is defined in the component class and can be updated based on the component's logic or state.

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'],  
})  
export class AppComponent {  
  classesArray = ['active', 'highlight', 'bold'];  
}
```

```
<div  
  [ngClass]="classesArray"></div>
```

# ngClass Directive

When ngClass is set to an object, it conditionally adds or removes classes based on the truthiness of the properties. Example:

```
<div [ngClass]="{ 'active': isActive, 'hidden': isHidden }"></div>
```

If **isActive** is true, the active class is added.

If **isHidden** is true, the hidden class is added.

Property binding can also be used with the HTML class attribute, if **isExpanded** is true the expanded class is added.

```
<div [class.expanded]="isExpanded"></div>
```

# ngStyle Directive

The **ngStyle** directive is used to dynamically apply **inline CSS** to an element. It binds an object with key-value pairs, where the key is the CSS property name, and the value is the property's value.

The value of **ngStyle** can directly use properties from the component class. The **color** and **font-size** styles are applied based on **textColor** and **fontSize** values from the component.

```
<div [ngStyle]="{ 'color': textColor, 'font-size': fontSize + 'px' }"> Rendered as
```

```
<div style="color: blue; font-size: 20px;">
</div>
```

# ngStyle Directive

ngStyle also allows you to apply inline styles conditionally by using component properties and expressions.

**font-style:** Set to italic if **canSave** is true, otherwise normal.

**font-weight:** Set to bold if **isUnchanged** is false, otherwise normal.

**font-size:** Set to 24px if **isSpecial** is true, otherwise 12px.

```
<div
  [ngStyle]="{
    'font-style': canSave ? 'italic' : 'normal',
    'font-weight': !isUnchanged ? 'bold' : 'normal',
    'font-size': isSpecial ? '24px' : '12px'
  }"
></div>
```

# Pipes

---

# Pipes

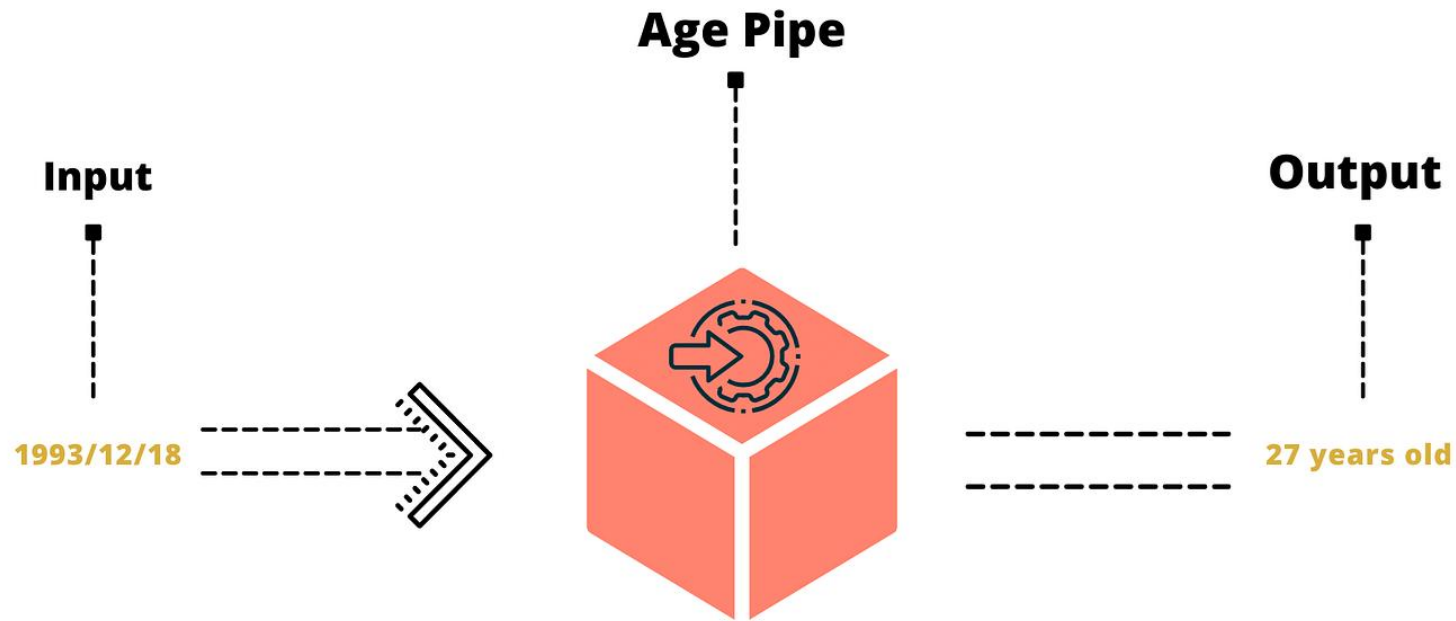
**Pipes** in Angular are similar to transformation functions that take **input** data and return **transformed output** directly in the **template**. They make it easy to format and display data without cluttering the component class with extra logic.

Pipes are used in templates with the **|** symbol to pass data through a transformation.

**Syntax:** `{{ value | pipeName }}` transforms value using pipeName.

There many **built in** pipes in Angular such as the **uppercase** pipe.

`<div>{{ text | uppercase }}</div>` This will convert text to uppercase letters.



# Pipes

Pipes can accept parameters using a colon : followed by a string.

Pipes can also be chained `{{ dateValue | date:'shortDate' | uppercase }}`

Mon Nov 11 2024 21:09:48 GMT+0200 (Eastern European Standard Time) Before using the date pipe  
11/11/24 After using the date pipe

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'],  
})  
export class AppComponent {  
  currentDate = new Date();  
}
```

```
<div>{{ currentDate | date:'shortDate' }}</div>
```



# Hands On Challenge : Todo List

**Time Limit:** 10 minutes

## Build a simple angular app Todo List

- 1) You can show/hide the task list (**\*ngIf**).
- 2) You can add tasks dynamically (**\*ngFor**).
- 3) The first task is highlighted in yellow (**ngClass**).
- 4) The last task is bold & blue (**ngStyle**).



### Task List

Hide List

Enter task

Add Task

- 1. Hands On
- 2. Lecture Assignment

# Hands On Solution

app.component.ts

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'my-app',
5   templateUrl: './app.component.html',
6   styleUrls: [ './app.component.css' ]
7 })
8 export class AppComponent {
9   showList = true;
10  newTask = '';
11  tasks: string[] = ['Hands On', 'Lecture Assignment'];
12
13  addTask() {
14    if (this.newTask.trim()) {
15      this.tasks.push(this.newTask.trim());
16      this.newTask = ''; // Clear input after adding
17    }
18  }
19 }
```

app.component.html

```
1 <div>
2   <h2>Task List</h2>
3
4   <button (click)="showList = !showList">
5     {{ showList ? 'Hide' : 'Show' }} List
6   </button>
7
8   <!-- Input & Add Button -->
9   <div>
10     <input [(ngModel)]="newTask" placeholder="Enter task" />
11     <button (click)="addTask()">Add Task</button>
12   </div>
13
14   <!-- Task List -->
15   <ul *ngIf="showList">
16     <li
17       *ngFor="let task of tasks; let i = index"
18       [ngClass]="{ highlight: i === 0, 'bold-blue': i === tasks.length - 1 }"
19       [ngStyle]="
20         i === tasks.length - 1 ? { color: 'blue', 'font-weight': 'bold' } : {}
21       "
22     >
23       {{ i + 1 }}. {{ task }}
24   </li>
```

# Assignment 1

Mock [this](#) site that showcases a product using Axure RP 9.

The Requirements are as follows:

1. Create a page for viewing a product's details identical to the one provided in the given link.
2. Create a carousel slider for displaying the product's images.
3. Keep showing the main navigation menu while scrolling the page down.
4. Make the Buy Now button transfer the user to a login page like [this](#) one.

Log in to [share.axure.com](https://share.axure.com) in your web browser and create a new workspace. Then, submit the preview link in the assignment [form](#) before 16 November 2025 at 11:55 pm.

You are only required to mock the visible part of the page when it loads without scrolling.

# Thank You ; )

## Any Questions ?

