# Chapter 3: Template Syntax

### 1. Interpolation ({{ ... }})

#### What you’re doing here

You’re showing some variable or text inside your HTML. Think of it as a placeholder that Angular fills with your component’s data.

#### Quick example snippet:

<h1>Hello, {{ userName }}!</h1>

#### Why this matters

Whenever you want your page to display some data — like a name, a message, or a number — interpolation is the easiest way.

#### Real-life uses

* Greeting a user by name on a dashboard
* Showing how many items are in a cart

#### Complete example you can try:

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

@Component({

standalone: true,

selector: 'app-greeting',

template: `<h1>Welcome, {{ userName }}!</h1>`

})

export class GreetingComponent {

userName = 'Alice';

}

### Full Example:

### interpolation.component.ts

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

import { CommonModule } from '@angular/common';

@Component({

selector: 'app-interpolation',

standalone: true,

imports: [CommonModule],

templateUrl: './interpolation.component.html',

styleUrls: ['./interpolation.component.css']

})

export class InterpolationComponent {

title = 'Angular Interpolation Demo';

userName = 'John Doe';

age = 25;

skills = ['Angular', 'TypeScript', 'HTML'];

user = {

firstName: 'John',

lastName: 'Doe',

email: 'john@example.com'

};

getFullName(): string {

return `${this.user.firstName} ${this.user.lastName}`;

}

getCurrentYear(): number {

return new Date().getFullYear();

}

rawHtml = '<b>This is bold text</b>';

}

### interpolation.component.html

<h1>{{ title }}</h1>

<!-- Interpolating a simple variable -->

<p>Hello, {{ userName }}!</p>

<!-- Interpolating a number -->

<p>Your age: {{ age }}</p>

<!-- Interpolating an expression -->

<p>Next year you will be: {{ age + 1 }}</p>

<!-- Interpolating an array value -->

<p>Your first skill is: {{ skills[0] }}</p>

<!-- Interpolating object property -->

<p>Email: {{ user.email }}</p>

<!-- Interpolating a method result -->

<p>Full Name: {{ getFullName() }}</p>

<p>Current Year: {{ getCurrentYear() }}</p>

<!-- Escaping HTML content -->

<p>This will be escaped: {{ rawHtml }}</p>

<!-- Rendering HTML safely -->

<p>This will render HTML: <span [innerHTML]="rawHtml"></span></p>

### 2. Property Binding ([property])

#### What you’re doing here

You’re connecting a property from your component to an HTML element’s property. It’s like telling the element, “Hey, here’s the value you should use!”

#### Quick example snippet:

<img [src]="logoUrl" />

<button [disabled]="isButtonDisabled">Click Me</button>

#### Why this matters

When you want the UI to **react to your component's state** — like showing a different image or enabling/disabling buttons — this is your tool.

#### Real-life uses

* Disabling a submit button until a form is ready
* Showing profile pictures loaded from a user account

#### Complete example you can try:

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

@Component({

standalone: true,

selector: 'app-button-disable',

template: `

<img [src]="logoUrl" alt="Logo" />

<button [disabled]="isButtonDisabled">Submit</button>

`

})

export class ButtonDisableComponent {

logoUrl = 'https://angular.io/assets/images/logos/angular/angular.svg';

isButtonDisabled = true;

}

### 3. Event Binding ((event))

#### What you’re doing here

You’re **listening** for something happening on the page — like a click or typing — and telling Angular what to do when it happens.

#### Quick example snippet:

<button (click)="increaseCounter()">Click me</button>

#### Why this matters

Your app can respond to user actions — click buttons, type in inputs, hover on things — making it interactive.

#### Real-life uses

* Clicking a “like” button to increase a count
* Logging user input when they type

#### Complete example you can try:

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

@Component({

standalone: true,

selector: 'app-clicker',

template: `

<button (click)="increaseCounter()">Clicked {{ count }} times</button>

`

})

export class ClickerComponent {

count = 0;

increaseCounter() {

this.count++;

}

}

#### List of events:

­­­­­

|  |  |
| --- | --- |
| (focus)="myMethod()"  (blur)="myMethod()"  (submit)="myMethod()"  (cut)="myMethod()"  (copy)="myMethod()"  (paste)="myMethod()"  (keydown)="myMethod()" | (keypress)="myMethod()"  (keyup)="myMethod()"  (mouseenter)="myMethod()"  (mousedown)="myMethod()"  (mouseup)="myMethod()"  (click)="myMethod()"  (dblclick)="myMethod()" |

### 4. Two-Way Binding ([(ngModel)])

#### What you’re doing here

You’re making a two-way connection between your component and an input field. When the user types, your component updates. When your component changes, the input updates.

#### Quick example snippet:

<input [(ngModel)]="userName" />

#### Why this matters

Makes forms super easy — no need to write extra code to keep things in sync.

#### Real-life uses

* Typing your name and seeing it appear live somewhere else on the page
* Binding checkboxes or dropdowns to your data model

#### Complete example you can try:

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

import { FormsModule } from '@angular/forms';

@Component({

standalone: true,

selector: 'app-name-input',

imports: [FormsModule],

template: `

<input [(ngModel)]="userName" placeholder="Enter your name" />

<p>Hello, {{ userName }}!</p>

`

})

export class NameInputComponent {

userName = '';

}

### 5. Local Template Variables (#variable)

#### What you’re doing here

You’re giving a nickname to an element inside your template so you can easily grab its value or call its methods right there in your template.

#### Quick example snippet:

<input #nameInput />

<button (click)="logName(nameInput.value)">Log Name</button>

#### Why this matters

You can access elements without extra TypeScript code — just right in the template.

#### Real-life uses

* Logging what a user typed when a button is clicked
* Getting a reference to an input box to focus it or clear it

#### 🧪 Complete example you can try:

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

@Component({

standalone: true,

selector: 'app-log-input',

template: `

<input #nameInput placeholder="Type your name" />

<button (click)="logName(nameInput.value)">Log Name</button>

`

})

export class LogInputComponent {

logName(value: string) {

console.log('User typed:', value);

}

}

### 6. Class Binding ([class.class-name] and [ngClass])

#### What you’re doing here

You’re adding or removing CSS classes dynamically based on your component’s data.

#### Quick example snippets:

* Single class toggle:

<button [class.active]="isActive">Click me</button>

* Multiple classes with an object:

<div [ngClass]="{ 'highlight': isHighlighted, 'error': hasError }"></div>

* We need CommonModule for ngclass, ngIf, ngFor…

import { CommonModule } from '@angular/common';

#### Why this matters

Control styles like highlighting, error states, or active states **without manipulating classes manually**.

#### Real-life uses

* Highlight the selected menu item
* Show red borders on invalid form inputs

#### Complete example you can try:

import { CommonModule } from '@angular/common';

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

@Component({

standalone: true,

imports: [CommonModule],

selector: 'app-class-binding',

template: `

<button [class.active]="isActive" (click)="toggleActive()">

Toggle Active ({{ isActive }})

</button>

<p [ngClass]="{ highlight: isActive, error: hasError }">

This paragraph changes class based on states.

</p>

<button (click)="toggleError()">Toggle Error</button>

`,

styles: [`

.active { background-color: lightgreen; }

.highlight { font-weight: bold; }

.error { color: red; }

`]

})

export class ClassBindingComponent {

isActive = false;

hasError = false;

toggleActive() {

this.isActive = !this.isActive;

}

toggleError() {

this.hasError = !this.hasError;

}

}

### 7. Style Binding ([style.property] and [ngStyle])

#### What you’re doing here

You’re changing CSS styles directly on elements dynamically.

#### Quick example snippets:

* Single style:

<div [style.color]="isError ? 'red' : 'black'">Text</div>

* Multiple styles:

<div [ngStyle]="{ 'font-size.px': fontSize, 'background-color': bgColor }"></div>

#### Why this matters

Control styles like colors, sizes, margins on the fly without CSS classes.

#### Real-life uses

* Change text color based on validation
* Adjust font size dynamically per user settings

#### Complete example you can try:

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

@Component({

standalone: true,

selector: 'app-style-binding',

template: `

<p [style.color]="isError ? 'red' : 'green'">

This text changes color dynamically.

</p>

<button (click)="toggleError()">Toggle Error</button>

`

})

export class StyleBindingComponent {

isError = false;

toggleError() {

this.isError = !this.isError;

}

}

### 8. Handling Optional or Null Values Safely (? safe navigation operator)

#### What you’re doing here

You’re protecting your templates from crashing if a value or object isn’t ready yet (i.e., null or undefined).

#### Quick example snippet:

<p>{{ user?.name }}</p>

#### Why this matters

Sometimes data loads asynchronously — this stops errors when data isn’t ready yet.

#### Real-life uses

* Showing user info after fetching from server
* Binding properties that might be missing or delayed

#### Complete example you can try:

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

@Component({

standalone: true,

selector: 'app-safe-navigation',

template: `

<p>User Name: {{ user?.name || 'Loading...' }}</p>

<button (click)="loadUser()">Load User</button>

`

})

export class SafeNavigationComponent {

user: { name: string } | null = null;

loadUser() {

setTimeout(() => {

this.user = { name: 'Charlie' };

}, 2000);

}

}

### 9. Combining It All Together

#### What you’re doing here

Mix different bindings — interpolation, property, event, class, style — all in one component to build real interactive UI.

#### Complete example you can try:

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

import { FormsModule } from '@angular/forms';

@Component({

standalone: true,

selector: 'app-binding-combo',

imports: [FormsModule],

template: `

<h2>{{ title }}</h2>

<input [(ngModel)]="userName" placeholder="Enter your name" #inputRef />

<button

(click)="greetUser(inputRef.value)"

[disabled]="!userName"

[class.active]="userName.length > 3"

[style.backgroundColor]="userName.length > 3 ? 'lightblue' : 'lightgray'">

Greet Me

</button>

<p \*ngIf="greeting" [ngClass]="{ highlight: greeting }">{{ greeting }}</p>

`,

styles: [`

.active { border: 2px solid blue; }

.highlight { font-weight: bold; color: green; }

`]

})

export class BindingComboComponent {

title = 'Welcome to Angular Bindings!';

userName = '';

greeting = '';

greetUser(name: string) {

this.greeting = `Hello, ${name}!`;

}

}