**What is Angular?**

* A **front-end JavaScript framework**
* Helps build **advanced, feature-rich, interactive web UIs**

**Why use Angular?**

* Simplifies UI development compared to using plain JavaScript
* Provides **structure, rules, and reusable packages**

**Beyond the Framework:**

* Angular includes a **tooling ecosystem**:
  + **CLI** (Command Line Interface) for project setup and dev server
  + **Debugging tools and IDE plugins**
  + Enhances **developer productivity and experience**

**Use Cases:**

* Suitable for projects ranging from **simple websites to enterprise-level apps**

**Why Use Angular Over Plain JavaScript?**

**1. Simplifies Complex UI Development**

* Angular is ideal for **large-scale, interactive apps**
* Offers structure, rules, and powerful built-in features

**2. Declarative Programming**

* You define **what the UI should look like**, not how to achieve it
* Angular handles the **DOM updates** behind the scenes

**3. Component-Based Architecture**

* Break UIs into **independent, reusable components**
* Encourages **separation of concerns**
* Makes **team collaboration** easier on large apps

**4. Embraces OOP Concepts**

* Uses **classes and dependency injection**
* Supports **scalable and maintainable** codebases
* Makes your app type safe

**5. Uses TypeScript (Not Vanilla JS)**

* Superset of JavaScript with **strong typing**
* Helps **catch errors early** in development
* Improves **code quality and maintainability**
* TypeScript knowledge will be covered **as part of the course**

**Angular Evolution & Versioning**

**Angular's Stability & Innovation**

* Angular (from **version 2**, released in **2016**) is a **complete rewrite** of AngularJS (Angular 1)
* Angular follows a **6-month major release cycle** (twice a year)
* Despite frequent updates, it's **highly stable** and **backward-compatible**

**Version Compatibility in Practice**

* Older code from Angular 2 still works in newer versions
* **New features are optional**, not mandatory
  + E.g., **Standalone Components (v14)** and **Signals (v16)**

**Real-World Relevance**

* Many teams still use **older Angular versions**
* Course content is designed to **support both old and new versions**
* Instructor highlights version-specific features clearly

**Course Approach**

* Learn Angular in a **modern and future-proof** way
* No need to relearn Angular with each release
* The course evolves along with Angular's updates

**Getting Started with Angular Projects**

**Why You Can’t Just Use HTML & JS Files**

* Angular uses **non-standard HTML syntax** and **TypeScript**
* Browsers can’t interpret this directly — needs **compilation and optimization**

**Angular CLI: Your Main Tool**

* CLI (Command Line Interface) is used to:
  + **Create Angular projects**
  + **Compile TypeScript to JavaScript**
  + **Run development servers**
  + **Optimize builds for production**

**Pre-requisite: Install Node.js**

* Visit [nodejs.org](https://nodejs.org) and install the **LTS version**
* Node.js includes npm (Node Package Manager)
* Required to install Angular CLI

**Install Angular CLI**

npm install -g @angular/cli

* Use sudo on Mac/Linux if needed
* After install, ng command becomes available

**Create New Angular Project**

ng new first-angular-app

* Choose options like:
  + **CSS or SCSS**
  + **Server-side rendering** (answer "No" for now)
* Project name: use **lowercase with dashes**, no spaces
* Follow CLI prompts to complete project setup

**Project Structure & Running the Angular App**

**1. Use the Provided Starter Project**

* Created using **Angular CLI**
* Ensures **consistent starting point** for all learners
* Folder structure may vary slightly with CLI versions (e.g., favicon location)

**2. Project Structure Overview**

* **Root-Level Files**:
  + **tsconfig.\*.json: TypeScript config (controls how TS compiles to JS)**
  + **package.json: Lists dependencies (Angular libraries, etc.)**
  + **angular.json: CLI and build config**
  + .editorconfig: Code formatting rules
  + .gitignore: Git-related exclusions
* **SRC Folder** *(main working area)*:
  + **index.html: Main HTML file loaded in the browser**
  + **main.ts: First TypeScript file executed (entry point)**
  + styles.css: Global styles for the app
  + assets/: Store static assets like images
  + app/: Core Angular code and components will be built here

**3. First Steps Before Running the App**

* Run this command to install dependencies:

npm install

(Only required **once** after downloading the project)

* To start the development server:

npm start

(Runs ng serve behind the scenes)

* Visit the **local URL** shown in terminal to view the app in the browser

**How Angular Loads & Displays Content**

**1. index.html Loads First**

* Contains only one tag: <app-root> (a **custom HTML tag**)
* **Not** standard HTML — the browser doesn't understand it directly

**2. Angular CLI Injects Scripts Automatically**

* No script tags in index.html initially
* When you run npm start (ng serve), Angular CLI:
  + Compiles TypeScript to JavaScript
  + Injects required scripts into the page

**3. main.ts Is the Entry Point**

* Executes **first** when app loads
* Calls bootstrapApplication(AppComponent)  
  → Bootstraps the root Angular component

**4. The Role of AppComponent**

* Defined in app.component.ts
* Decorated with @Component → tells Angular it's a component
  + **Selector**: 'app-root' → matches the tag in index.html
  + **templateUrl**: path to HTML markup for the component
  + **styleUrls**: scoped styles only for this component

**5. Angular Replaces <app-root>**

* Angular finds <app-root> in index.html
* Replaces it with the HTML from app.component.html  
  → That’s what you see on the screen (title, subtitle, image, etc.)

# ****Starting the Demo App – Creating a Header Component****

**1. Angular App = Multiple Reusable Components**

* App is made of **UI building blocks** (e.g., header, sidebar, dialog)
* Each **block = separate Angular Component**

**2. Component File Naming Convention**

* Format: component-name.component.ts
  + Example: header.component.ts
  + Follows the convention:
    - header: descriptive name
    - .component: indicates it’s a component
    - .ts: TypeScript file

**3. Component Structure**

* A component typically includes:
  + .ts: TypeScript logic class with @Component decorator
  + .html: Template file with UI markup
  + .css/.scss: Component-scoped styles

**4. Creating a Component Manually**

* Step-by-step to create HeaderComponent:
  + Create header.component.ts file
  + Add and **export** a class HeaderComponent
  + Decorate it using @Component({ ... })
  + Import Component decorator from @angular/core
  + Keep the class body empty for now

**Example Skeleton Code:**

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

@Component({

selector: 'app-header',

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

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

})

export class HeaderComponent { }

You'll also need to create header.component.html and header.component.css in the same folder.

# ****Organizing and Generating Angular Components****

**1. Organize Components with Subfolders**

* Place each component’s files into its own **feature-named subfolder** inside src/app
  + Example: src/app/header/ → stores header.component.ts, .html, .css
* Keeps the project **clean and maintainable** as the app grows
* **Update import paths** if files are moved (IDE usually handles this)

**2. Generate Components Using Angular CLI**

* Use CLI to avoid manual file creation
* Command:

ng generate component component-name

or shorthand:

ng g c component-name

Example:

ng g c user

**3. What CLI Generates:**

* **user.component.ts → Component class with selector & decorator**
* **user.component.html → Template**
* user.component.css → Styles (empty by default)
* user.component.spec.ts → Unit test file (can delete for now)

**4. CLI-Generated Features**

* **Follows naming conventions**
* standalone: true is added by default (recommended modern approach)
* Includes imports: [] array for importing other components or modules

# ****Using & Styling the User Component****

**1. Add Markup to UserComponent Template**

* Basic structure inside user.component.html:

<div>

<button>

<img src="" alt="User Image" />

<span>User Name</span>

</button>

</div>

* Will later bind dynamic data (image + name)

**2. Add Styles for Better Appearance**

* Paste predefined styles into user.component.css
* Ensures the layout looks polished

**3. Display UserComponent in AppComponent Template**

* Add custom tag below header:

<app-user />

**4. Fix Unknown Component Issue**

* Angular needs to **know about** the UserComponent
* Import it in app.component.ts and add to imports array
  + VS Code offers auto-import via lightbulb/quick fix

**5. Clean Up Template**

* Use **self-closing syntax** for components without inner content:

<app-user />

**6. Improve Layout**

* Wrap app-user in semantic HTML & styled container:

<main>

<ul id="users">

<li><app-user /></li>

</ul>

</main>

* Copy provided styles into app.component.css for better layout

**Next Step:** Add dynamic user image and name in the c

# ****String Interpolation in Angular****

**1. Purpose of Interpolation**

* Bind **dynamic values** from the component class to the **template**
* Used for **text content**, such as names, labels, etc.

**2. Syntax**

{{ selectedUser.name }}

* Double curly braces: {{ ... }}
* Can access any **public property** from the component class

**3. Access Object Properties**

* If selectedUser is an object, use dot notation:
  + selectedUser.name
  + selectedUser.image

**4. Public vs. Private Properties**

* Only **public properties** are accessible in the template
* Omitting private makes a property public by default

**5. IDE Support**

* Visual Studio Code and others provide **auto-completion**
* Helps with correct property usage and error reduction

**6. Result**

* Every page reload shows a **random user’s name**, thanks to dynamic binding

# ****String Interpolation in Angular****

**1. Purpose of Interpolation**

* Bind **dynamic values** from the component class to the **template**
* Used for **text content**, such as names, labels, etc.

**2. Syntax**

{{ selectedUser.name }}

* Double curly braces: {{ ... }}
* Can access any **public property** from the component class

**3. Access Object Properties**

* If selectedUser is an object, use dot notation:
  + selectedUser.name
  + selectedUser.image

**4. Public vs. Private Properties**

* Only **public properties** are accessible in the template
* Omitting private makes a property public by default

**5. IDE Support**

* Visual Studio Code and others provide **auto-completion**
* Helps with correct property usage and error reduction

**6. Result**

* Every page reload shows a **random user’s name**, thanks to dynamic binding

# ****String Interpolation vs. Property Binding****

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

* Used for **dynamic text content**
* Syntax:

<span>{{ selectedUser.name }}</span>

* Only works for **text between HTML tags**

#### ****2. Property Binding (****[property]="value"****)****

* Used to **set DOM properties or attributes** (e.g., src, alt, disabled)
* Syntax:

<img [src]="'assets/users/' + selectedUser.avatar" [alt]="selectedUser.name" />

* Wrap the **property name in square brackets**
* Value inside quotes is **JavaScript expression**, no curly braces

#### ****3. When to Use What****

| **Use Case** | **Recommended Binding** |
| --- | --- |
| Text between tags | {{ interpolation }} |
| Setting attributes | [property]="expression" |

#### ****4. Dynamic Path Example****

* Construct dynamic file paths in binding expressions:

'assets/users/' + selectedUser.avatar

#### ****5. Key Takeaway****

* **Angular is about dynamic, data-driven UI**
* Mastering **interpolation** and **property binding** is essential for building interactive apps

# ****Using Getters for Cleaner Template Code****

**1. Why Use Getters in Angular Components?**

* Keeps your **template clean** and readable
* Offloads **complex expressions or computed values** to the component class
* Recommended practice for **maintainability**

**2. Example Use Case: Image Path Construction**

* Instead of writing this in the template:

[src]="'assets/users/' + selectedUser.avatar"

* Move it into the component class using a **getter**:

get imagePath() {

return 'assets/users/' + this.selectedUser.avatar;

}

**3. Getter Syntax in TypeScript**

* Defined using get keyword
* Can be used like a property (no parentheses)
* Access other class properties with this

this.selectedUser.avatar

**4. Using the Getter in the Template**

* Simple property binding with the getter:

<img [src]="imagePath" />

**5. Summary**

* Use **getters** to compute values cleanly in the class
* Use **property binding** in the template to refer to those getters
* Improves **code readability**, **reusability**, and **testability**

# ****Handling User Input with Event Binding****

**1. Listening to User Events in Angular**

* Use **event binding** syntax to react to DOM events
* Common use case: click event on a button

**2. Syntax for Event Binding**

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

* **Wrap event name in parentheses**: (click)
* Call the method defined in the component class

**3. Define the Event Handler in the Component Class**

onSelectUser() {

console.log('Clicked');

}

* Method names commonly start with on (e.g., onClick, onSelectUser)
* Can execute any logic inside the handler (e.g., update data, show alert)

**4. Supported Events**

* You can bind to **any standard DOM event**
* Use **Ctrl + Space** in the template to see suggestions in VS Code

**5. Key Takeaway**

* Angular makes it easy to **react to user input** declaratively
* Use (event)="handler()" syntax and define logic in the component class

# ****Updating the UI Dynamically on Events****

**1. Combining Event Binding + Data Binding**

* React to events **and** update the data (state)
* Angular automatically re-renders the UI when data changes

**2. Use Case: Change User on Click**

* Instead of changing the user **only on reload**, change it on **button click**

**3. Managing Component State**

* selectedUser is the **state**
* When selectedUser changes, UI reflects the new state automatically

**4. Updated onSelectUser() Method**

onSelectUser() {

const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

this.selectedUser = DUMMY\_USERS[randomIndex];

}

* Recalculates a **new random index** every time the button is clicked
* Updates selectedUser with a new value
* Angular automatically updates the view (name & image)

**5. Key Takeaway**

* No need for special state management tools
* Simply **update the property** bound in the template → Angular handles UI updates

# ****How Angular Automatically Updates the UI (Change Detection)****

**1. State-Driven UI**

* Any **data stored in component properties** (state)  
  → When updated, **automatically updates the UI**

**2. Angular's Change Detection Mechanism**

* Angular uses **change detection** to track data changes
* When a property value changes, Angular:
  + Checks the **template for changes**
  + Compares the new DOM snapshot
  + **Updates the view** if necessary

**3. Powered by zone.js (Under the Hood)**

* zone.js is a **library** used internally by Angular
* It tracks **asynchronous events**:
  + User actions (clicks, inputs)
  + Timers, HTTP calls, etc.
* After each tracked event, Angular **runs change detection**

**4. Developer Benefit**

* No need for manual DOM updates
* Just update component state → Angular **handles the rest**

**5. Key Takeaway**

* Angular’s UI is **reactive by default**
* **Automatic, event-driven UI updates** make it easier to build dynamic apps

# ****Introducing Signals in Angular (Angular 16+)****

#### ****1. What Are Signals?****

* A **new way to manage state** in Angular
* Acts like a **reactive container** that holds a value
* Automatically **notifies Angular** when the value changes
* UI updates only when **specific Signal values** change

#### ****2. Creating a Signal****

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

selectedUser = signal(DUMMY\_USERS[randomIndex]);

* Initialized with a value (e.g., random user)
* Replaces traditional component properties for state

#### ****3. Reading a Signal****

* Must **call it as a function**:

{{ selectedUser().name }}

* Ensures Angular **tracks the usage** and re-renders only when needed

#### ****4. Updating a Signal****

selectedUser.set(DUMMY\_USERS[newRandomIndex]);

* Triggers automatic UI update **only where it's used**

#### ****5. Replacing Getters with**** computed()

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

imagePath = computed(() => {

return 'assets/users/' + selectedUser().avatar;

});

* Creates **derived/computed values** from Signals
* Re-evaluates **only when dependent Signals** change
* Also needs to be **called like a Signal** in template:

<img [src]="imagePath()" />

#### ****6. Signals vs. Traditional State (Zone.js)****

|  |  |
| --- | --- |
| **Traditional Angular (Zone.js)** | **Signals (New Reactive Model)** |
| UI updated via **event tracking** | UI updated via **explicit tracking** |
| **All components checked** on events | **Only affected parts updated** |
| Uses Zone.js for change detection | No need for Zone.js |

#### ****7. Compatibility Note****

* **Signals introduced in Angular 16**, stable in **Angular 17**
* May not be usable in older projects
* Course uses Signals in this component only  
  → Full Signals section later in course

# ****Making Components Reusable with Input Binding****

#### ****1. Reusability = Power of Angular Components****

* You can use the **same component multiple times** with **different data**
* Example: Multiple <app-user /> entries showing different users

#### ****2. Current Limitation****

* UserComponent currently holds **its own data logic** (random user)
* All instances show the **same user initially**
* Defeats the purpose of **reusable + configurable components**

#### ****3. Goal: Make**** UserComponent ****Configurable****

<app-user [user]="userData" />

* That allows each instance to **display a different user**

#### ****4. @Input()****

* Introduce Angular’s @Input() decorator
* Use @Input() to accept external values into a component
* This enables **custom attributes** like [user] to pass data down

#### ****5. Purpose of**** @Input()

* Allows a parent component to **pass data** to a child component
* Makes components **reusable and dynamic**

#### ****6. How to Use**** @Input()

* In the child component (UserComponent):

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

@Input() avatar!: string;

@Input() name!: string;

* Use !: to tell TypeScript the property **will be assigned** externally
* Set the expected type (e.g., string)

#### ****7. Pass Data from Parent Template****

* In AppComponent:

<app-user [avatar]="users[0].avatar" [name]="users[0].name"></app-user>

<app-user [avatar]="users[1].avatar" [name]="users[1].name"></app-user>

<!-- and so on -->

#### ****8. Using Inputs in Template****

* Update template to reflect new bound properties:

<img [src]="imagePath" [alt]="name" />

<span>{{ name }}</span>

* Getter in UserComponent for image path:

get imagePath() {

return 'assets/users/' + this.avatar;

}

#### ****9. Why This Matters****

* Avoids hardcoded logic inside components
* Makes components **stateless, testable, and flexible**
* Encourages **component reusability** with different data

#### ****10. TypeScript Notes****

* @Input() props must be **typed**
* Use !: to avoid initialization errors for externally set values

# ****Enforcing Required Inputs in Angular with**** @Input({ required: true })

#### ****1. Problem****

* Without required: true, **missing inputs** (e.g., [name], [avatar]) won’t cause errors during development
* Using !: only tells **TypeScript** it will be initialized — not Angular

#### ****2. Risk****

* If you forget to bind a required input:
  + UI breaks (e.g., missing name/image)
  + No TypeScript warning unless explicitly handled

#### ****3. Solution: Add**** required: true ****to**** @Input()

@Input({ required: true }) name!: string;

@Input({ required: true }) avatar!: string;

* Tells **Angular** that this input **must be set**
* IDEs like VS Code will show errors **at usage sites** if the input is missing

#### ****4. Benefits****

* **Early error detection** during development
* Improves **developer experience and reliability**
* Helps align with !: assertion in TypeScript

#### ****5. Example Error Message****

"The required input 'name' from component 'UserComponent' must be specified."

# Two Ways to Accept Inputs

1. **Classic approach**  
   Using the @Input() decorator (with uppercase I)
2. **Modern approach**  
   Using the input() function (with lowercase i) that returns a **read-only signal**

### Using the Signal-based input() Function

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

avatar = input<string>(); // with optional default value

name = input.required<string>(); // marks as required input

* These are **signals**, so they must be **called as functions** in templates:

<img [src]="imagePath()" [alt]="name()" />

<span>{{ name() }}</span>

### Computed Inputs

Use computed() to derive values from signal inputs:

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

imagePath = computed(() => 'assets/users/' + this.avatar());

### Signal Inputs Are Read-Only

* Cannot update a signal input inside the component (.set() is not allowed)
* Signal inputs only update when the parent component passes new values

### Advantages of Signal Inputs

* More efficient change detection through fine-grained tracking
* No need for !: or default value workarounds
* Automatically tracks dependencies and re-renders affected template parts

### When Not to Use Signal Inputs

* Signals are a newer feature (Angular 16+)
* Most existing codebases, especially in enterprises, still use the classic @Input() approach
* Signal support may not be relevant in projects with older Angular versions or simpler data flow

### Key Takeaway

Learn and understand **both approaches**:

* Use @Input() for compatibility with existing codebases
* Use input() for modern, reactive, and fine-grained state management when applicable

# Emitting Custom Events from Components in Angular

#### Why Emit Events?

* **Inputs** pass data into a component.
* **Outputs** allow a component to emit events outward to the parent component.
* Helps parent components react to child component actions (e.g., button click).

### Setup: Emitting Events from a Component

#### Step 1: Import and Use @Output and EventEmitter

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

@Output()

select = new EventEmitter<string>(); // Event emitting a string (user ID)

#### Step 2: Emit Event in Response to Action

onSelectUser() {

this.select.emit(this.id); // Emit user ID

}

#### Step 3: Pass Required Inputs (like id, avatar, name)

@Input() id!: string;

### Using the Custom Event in the Parent

#### Step 4: Handle the Event in App Component

<app-user

[id]="users[0].id"

(select)="onSelectUser($event)">

</app-user>

* (select) binds to the custom event.
* $event holds the emitted value (user ID).

onSelectUser(id: string) {

console.log('Selected user with ID:', id);

}

### TypeScript Tip

If you receive an error like:

"Parameter 'id' implicitly has an 'any' type"

onSelectUser(id: string)

### Key Takeaways

* Use @Output() with EventEmitter to emit custom events.
* Use $event in the parent to access emitted data.
* This pattern allows **reusable components** to trigger **external behavior**.

# Emitting Custom Events in Angular

#### 1. Traditional Approach (Most Common)

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

@Output()

select = new EventEmitter<string>();

onSelectUser() {

this.select.emit(this.id);

}

* Uses @Output decorator.
* Explicitly creates an EventEmitter.
* Widely used in most Angular projects.

#### 2. Modern Approach (New in Angular 17+)

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

select = output<string>();

onSelectUser() {

this.select.emit(this.id);

}

* Uses output() function instead of a decorator.
* Shorter syntax, no need to create EventEmitter manually.
* Requires generic type (<string>).

#### Template Usage (Same for Both)

<app-user (select)="onSelectUser($event)"></app-user>

* $event holds the emitted value.

#### Notes

* output() simplifies syntax but behaves the same.
* Most projects still use @Output.
* Use output() if you're already using input() to avoid mixing decorators and functions.

# Rendering Lists Dynamically in Angular with @for

Manually repeating <app-user> components for each user is inefficient, especially when:

* The number of users is large or changes dynamically.
* You want maintainable, scalable code.

### Solution: Use Angular’s @for Syntax

Angular provides a built-in syntax for rendering lists: @for.

@for (user of users; track user.id) {

<li><app-user [name]="user.name" [avatar]="user.avatar" [id]="user.id" (select)="onSelectUser($event)"></app-user></li>

}

* user of users: Loops through the array users, exposing each user as user.
* track user.id: Tells Angular how to uniquely identify each item in the list for efficient DOM updates.

### Why track is Required:

Angular needs to efficiently manage the DOM when items are added, removed, or changed.

* By specifying track user.id, Angular uses the unique id to track each rendered item.
* This avoids unnecessary re-rendering and improves performance.

### Result:

This approach dynamically renders the list based on the number of users and makes your component reusable and maintainable.

# Conditional Rendering in Angular with @if and @else

Sometimes, you don’t want to render content unless certain conditions are met — for example, only showing the TasksComponent if a user is selected.

### Solution: Use Angular's @if Syntax

Just like @for is used for loops, @if is used for conditionally rendering template blocks.

### Syntax:

@if (selectedUser) {

<h2>{{ selectedUser.name }}'s Tasks</h2>

<app-tasks [user]="selectedUser"></app-tasks>

} @else {

<p id="fallback">Select a user to see their tasks.</p>

}

### How it works:

* @if conditionally renders content only if the expression evaluates to true.
* @else provides a fallback UI when the condition is not met.
* Inside the @if block, Angular knows selectedUser is defined, so no extra checks or non-null assertions (!) are needed.

### Why this is useful:

* It prevents rendering empty or invalid elements.
* It keeps your UI clean and meaningful.
* It allows you to handle both states (data available vs. not) in a declarative, readable way.

# Angular Structural Directives: \*ngFor and \*ngIf (Legacy Syntax)

If you're working with Angular **versions before 17**, you **won't have access to** @for and @if. Instead, you'll use the traditional **structural directives**: \*ngFor and \*ngIf.

### Replacing @for with \*ngFor

**Old syntax:**

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

<app-user [name]="user.name" [avatar]="user.avatar"></app-user>

</li>

**Don’t forget to:**

1. **Import NgFor** from @angular/common
2. **Add it to the imports array** in your component:

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

@Component({

// ...

imports: [NgFor]

})

### Replacing @if and @else with \*ngIf

<h2 \*ngIf="selectedUser">{{ selectedUser!.name }}</h2>

<ng-template #fallback>

<p>Select a user to see their tasks.</p>

</ng-template>

**And reference the else block like this:**

<h2 \*ngIf="selectedUser; else fallback">

{{ selectedUser!.name }}

</h2>

**Import Required:**

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

@Component({

// ...

imports: [NgIf]

})

### Best Practice (Angular 17+):

Prefer the **@for** and **@if** syntax for newer projects. It’s cleaner and eliminates the need for NgFor/NgIf imports.

### Why still learn the old way?

* **Legacy projects** still use \*ngIf and \*ngFor
* Many tutorials, blogs, and StackOverflow answers still reference them
* You’ll likely encounter both styles in real-world Angular development

# ****Two-Way Binding with ngModel for Form Inputs****

To capture user input in real-time and keep it in sync with component state, Angular offers **two-way data binding** using the [(ngModel)] syntax.

### 1. ****Setting Up Component Properties****

Inside your component class (new-task.component.ts), define properties to store the input values:

enteredTitle = '';

enteredSummary = '';

enteredDate = '';

Angular will bind the values entered by the user into these fields in the template.

### 2. ****Using ngModel in the Template****

In your component template (new-task.component.html), add the [(ngModel)] directive to input fields to set up two-way binding:

<input type="text" [(ngModel)]="enteredTitle" />

<textarea [(ngModel)]="enteredSummary"></textarea>

<input type="date" [(ngModel)]="enteredDate" />

This setup ensures that:

* The input values are updated in the component properties on each keystroke.
* Any programmatic updates to the properties also reflect in the input fields.

### 3. ****Enabling ngModel****

By default, Angular doesn't recognize [(ngModel)] unless the FormsModule is imported.

In the component's file:

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

Then add it to the imports array of the component:

@Component({

...

imports: [FormsModule]

})

### 4. ****Demonstrating Two-Way Binding****

To test the two-way binding, you can add multiple inputs bound to the same property. When typing into one, the others reflect the same value instantly.

This confirms that:

* Component state updates on user input.
* Changes in the component automatically update the DOM.

### 5. ****Note on Input Types****

Even if you use an <input type="date">, the value will still be a string, not a Date object. Keep this in mind when handling the input programmatically.

### Summary

Two-way binding with ngModel is a powerful Angular feature that:

* Simplifies form handling.
* Keeps component state and the template in sync.
* Is ideal for small-to-medium complexity forms.

For more advanced form needs, Angular offers **Reactive Forms**, which you'll explore later in the course.

# ****Using Signals with Two-Way Binding in Angular****

Angular supports the use of **signals** alongside **two-way data binding** via [(ngModel)]. This offers an alternative to standard class properties, enabling more fine-grained reactivity.

### 1. ****Replacing Class Properties with Signals****

Instead of using regular properties like:

enteredTitle = '';

You can define signals:

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

enteredTitle = signal('');

enteredSummary = signal('');

enteredDate = signal('');

This wraps your initial values in reactive signal containers.

### 2. ****Using**** [(ngModel)] ****with Signals****

No change is required in your template. You can continue using:

<input type="text" [(ngModel)]="enteredTitle" />

Angular's ngModel directive will:

* Automatically detect the property is a signal.
* Internally call the signal to read its value.
* Update the signal when the input value changes.

This means you don’t need to manually read (enteredTitle()) or update (enteredTitle.set()) the signal yourself for input binding.

### 3. ****Important Notes****

* **Do not use signal reading syntax (enteredTitle()) in [(ngModel)]**. Angular handles this automatically.
* If you try to use enteredTitle() in two-way binding, Angular will throw an error:  
  "Unsupported expression in a two-way binding."

### 4. ****Conclusion****

Using signals with [(ngModel)] is seamless and requires no changes in the template. However, in many cases, regular properties are sufficient. Signals are powerful for more advanced reactivity and efficiency, but it's essential to understand when and how to use them.

For this example, we’ll continue with regular properties, but now you know how to work with signals in two-way binding contexts when needed.

# ****Handling Form Submission in Angular with**** ngSubmit

After setting up two-way binding for capturing input values, the next step is to handle **form submission** so we can use the data and update our application state.

### 1. ****Understanding Browser Default Form Behavior****

By default, when a <form> is submitted:

* The browser **automatically sends a request** to the server that served the page.
* In development (e.g., using Angular’s dev server), this is problematic because the server is not configured to handle such POST requests.

### 2. ****Preventing Default Behavior with Angular****

Good news — Angular automatically prevents the default form submission when using the FormsModule.

How?

* FormsModule includes an internal Angular form component that:
  + Recognizes <form> elements.
  + Listens to the native submit event.
  + Prevents the browser’s default behavior automatically.

So you don’t need to manually call event.preventDefault() — Angular handles that for you.

### 3. ****Reacting to Form Submissions with**** ngSubmit

While the default action is prevented, we **still want to run our own code** when the form is submitted. Angular provides a special event for this:

* Use (ngSubmit)="onSubmit()" on the <form> element.
* This event is emitted **only after the default submission is canceled**.

#### Example:

<form (ngSubmit)="onSubmit()">

<!-- form controls -->

<button type="submit">Add Task</button>

</form>

In the component:

onSubmit() {

// Access enteredTitle, enteredSummary, enteredDate

// Add the new task to the task list

// Close the dialog

}

### 4. ****Summary****

* Angular’s FormsModule automatically blocks the default browser form submission.
* Use (ngSubmit) to handle the event and trigger your custom logic.
* You do not need to handle event objects or call preventDefault() manually.
* This approach keeps your form reactive and fully managed within Angular.

# ****Creating a Reusable Wrapper with**** ng-content ****and a Shared Card Component****

To follow best practices and keep styling consistent across your application, you can extract commonly used UI patterns into reusable components. One such pattern is a card-like container with rounded corners and drop shadows.

### ****1. Why Use a Reusable Component?****

Instead of repeating CSS styles across components (e.g., for drop shadows and border radii), we can encapsulate the styles inside a dedicated **CardComponent** that can be reused as a wrapper wherever needed.

### ****2. Creating the Shared**** CardComponent

Generate the component using Angular CLI:

ng generate component shared/card --skip-tests

This creates the CardComponent inside a shared folder, indicating it’s a reusable UI element.

### ****3. Styling the Card****

Copy the desired styles (e.g., border radius, box-shadow, overflow) into card.component.css.

.card {

border-radius: 6px;

box-shadow: 0 2px 8px rgba(0, 0, 0, 0.2);

overflow: hidden;

}

Then wrap the contents of the component template in a <div> with that class.

### ****4. Rendering Wrapped Content with**** ng-content

In the card.component.html, use the special Angular <ng-content> tag:

<div class="card">

<ng-content></ng-content>

</div>

This placeholder allows any content wrapped by <app-card> to be injected dynamically. Without this tag, the component would overwrite the content it's meant to wrap.

### ****5. Using the CardComponent in Other Components****

Wrap elements in the UserComponent and TaskComponent using <app-card> instead of duplicating style logic.

Example in user.component.html:

<app-card>

<button (click)="onSelectUser()">...</button>

</app-card>

Example in task.component.html:

<app-card>

<article>...</article>

</app-card>

Don’t forget to import CardComponent in each component where it’s used via the imports array of the @Component decorator.

### ****6. Outcome****

Now, all components wrapped with <app-card> will consistently have the same styling, and markup stays clean and reusable.

### ****Key Angular Concept:**** ng-content

* ng-content allows projecting custom content into a component.
* This enables you to build flexible wrapper or layout components.
* It’s essential for building reusable UI building blocks in Angular.

# ****Using Pipes to Format Output in Angular****

Pipes in Angular are **output transformers** used to format or transform displayed data directly in templates. A common use case is formatting dates, numbers, or currencies to be more human-readable.

### ****1. Why Use Pipes?****

When rendering data in the UI, raw values like ISO date strings can be less readable. Pipes help display such values in a cleaner format without manipulating the data in your component logic.

### ****2. Applying the Built-in**** date ****Pipe****

In task.component.html, suppose you're displaying a due date like this:

<p>{{ task.dueDate }}</p>

You can transform it using the date pipe by adding the pipe symbol (|) and the pipe name:

<p>{{ task.dueDate | date }}</p>

This tells Angular to format the dueDate using the built-in date pipe.

### ****3. Importing the Pipe (if Required)****

If Angular doesn't recognize the date pipe, you need to **import** it from @angular/common:

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

Then, add it to the component’s imports array:

imports: [DatePipe, CardComponent]

### ****4. Customizing Pipe Output with Parameters****

You can customize the output format by passing a parameter after a colon (:):

<p>{{ task.dueDate | date: 'short' }}</p>

This produces a short version of the date.

Some other format options:

* 'shortDate'
* 'mediumDate'
* 'fullDate' (e.g., "Monday, July 22, 2024")
* 'shortTime', 'mediumTime', 'fullTime'

For example:

<p>{{ task.dueDate | date: 'fullDate' }}</p>

This renders the date in a complete, readable format.

### ****5. Summary****

* Pipes are Angular's way to transform data right in templates.
* The date pipe formats date strings into readable formats.
* You can configure pipes using optional parameters.
* Other built-in pipes include uppercase, lowercase, currency, decimal, and percent.