Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

COMPONENTS, DIRECTIVES & DATA BINDING (WEEKS 3-4)

Understanding Components and Templates in Angular

CHAPTER 1: INTRODUCTION TO ANGULAR COMPONENTS

1.1 What Are Components in Angular?

In **Angular**, a **component** is the fundamental building block of an application. Each component:

✓ **Defines a UI section** – Represents a part of the webpage.
✓ **Handles logic and behavior** – Contains code that interacts with the UI.
✓ **Encapsulates styles** – Has its own CSS, making it reusable.

Each Angular application starts with a **root component (AppComponent)**, which serves as the entry point of the application.

1.2 Anatomy of an Angular Component

An Angular component consists of **three main files**:

1. **TypeScript File (.ts)** – Contains logic and data.
2. **HTML Template (.html)** – Defines the structure of the UI.
3. **CSS File (.css)** – Manages styles for the component.

Example: Basic Angular Component

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


@Component({
 selector: 'app-welcome',
 templateUrl: './welcome.component.html',
 styleUrls: ['./welcome.component.css']
})
export class WelcomeComponent {
 message: string = "Welcome to Angular Components!";
}
```

✓ selector: Defines the **custom HTML tag** for the component.
✓ templateUrl: Points to the **HTML file** for the component.
✓ styleUrls: Links to the **CSS file** for styling.

CHAPTER 2: CREATING AND USING COMPONENTS

2.1 Generating a Component Using Angular CLI

To create a new component, use the following command:

```
ng generate component welcome
```

✓ This creates a welcome component inside src/app/.

2.2 Using a Component in Another Component

To display the WelcomeComponent, add its selector <app-welcome> inside app.component.html:

<h1>My Angular Application</h1>

<app-welcome></app-welcome>

✓ The **selector (app-welcome)** allows embedding one component inside another.

CHAPTER 3: UNDERSTANDING TEMPLATES IN ANGULAR

3.1 What Is a Template in Angular?

A **template** defines the **HTML structure and dynamic content** of a component.

✓ Uses **Angular directives** to control behavior (*ngIf, *ngFor).
✓ Supports **data binding** ({{ }}) to display values dynamically.
✓ Can use **inline HTML** or an **external file**.

Example: Template with Dynamic Data

<h2>{{ message }}</h2>

<button (click)="changeMessage()">Click Me</button>

✓ **{{ message }}** binds the TypeScript variable to the UI.
✓ **(click)="changeMessage()"** triggers a function when the button is clicked.

3.2 Inline vs. External Templates

Angular allows defining **templates in two ways**:

1️⃣ External Template (Recommended for Large Apps)

```
@Component({
 selector: 'app-welcome',
 templateUrl: './welcome.component.html',
 styleUrls: ['./welcome.component.css']
})
```

✓ **Uses a separate HTML file** (welcome.component.html).

2️⃣ Inline Template (For Simple Components)

```
@Component({
 selector: 'app-inline',
 template: `<h2>Welcome to Angular</h2>`,
 styles: ['h2 { color: blue; }']
})
```

✓ **Uses a string inside template,** making it compact but harder to maintain.

CHAPTER 4: DATA BINDING IN TEMPLATES

4.1 Interpolation ({{ }}) – Displaying Data

Interpolation allows embedding TypeScript variables in HTML.

Example: Using Interpolation in a Template

```
export class WelcomeComponent {
```

```
 username: string = "Alice";
}
<h2>Welcome, {{ username }}!</h2>
```

✓ Displays **dynamic data** in the UI.

4.2 Property Binding ([property]) – Binding to Attributes

Property binding updates **HTML attributes dynamically**.

Example: Binding to an Image Source

```
export class WelcomeComponent {
 imageUrl: string = "assets/logo.png";
}
<img [src]="imageUrl" alt="Angular Logo">
```

✓ Updates the src attribute dynamically.

4.3 Event Binding ((event)) – Handling User Actions

Event binding listens for **user interactions** like clicks, keypresses, and form submissions.

Example: Button Click Event

```
export class WelcomeComponent {
 message: string = "Welcome to Angular!";

 changeMessage() {
  this.message = "You clicked the button!";
```

```
  }
}
<h2>{{ message }}</h2>
<button (click)="changeMessage()">Click Me</button>
```

✓ Updates message dynamically when the button is clicked.

CHAPTER 5: USING DIRECTIVES IN TEMPLATES

5.1 *ngIf – Conditional Rendering

*ngIf dynamically **hides or shows** elements.

Example: Show Content Based on a Condition

```
export class WelcomeComponent {
  isLoggedIn: boolean = false;
}
<p *ngIf="isLoggedIn">Welcome back!</p>
```

✓ The text **only appears if isLoggedIn is true**.

5.2 *ngFor – Looping Over Data

*ngFor repeats elements for **each item in an array**.

Example: Displaying a List of Users

```
export class WelcomeComponent {
  users = ["Alice", "Bob", "Charlie"];
}
```

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

✓ Loops through users and **displays each name in a <li> element**.

Case Study: How an E-Commerce Platform Used Angular Components for Product Listings

Background

An **e-commerce platform** needed **modular, reusable UI components** to display **products dynamically**.

Challenges

✓ **Hard-to-maintain HTML files** without reusable components.
✓ **Slow page load times** due to inefficient DOM updates.
✓ **Difficult to implement real-time filtering and sorting**.

Solution: Implementing Angular Components and Templates

✓ **Built a ProductComponent** to display each product separately.
✓ **Used *ngFor to dynamically list products** from an API.
✓ **Implemented event binding ((click)) to add products to the cart**.

```
export class ProductComponent {
  products = [
    { name: "Laptop", price: 1000 },
    { name: "Smartphone", price: 500 }
  ];
```

```
 addToCart(product: any) {
  console.log(product.name + " added to cart!");
 }
}
<div *ngFor="let product of products">
 <h3>{{ product.name }}</h3>
 <p>Price: ${{ product.price }}</p>
 <button (click)="addToCart(product)">Add to Cart</button>
</div>
```

Results

🚀 **40% faster page load times** with component-based rendering.
⚡ **Improved maintainability** by reusing UI components.
🔍 **Better user experience** with real-time cart updates.

By using **Angular Components and Templates**, the e-commerce platform improved **performance, code organization, and reusability**.

Exercise

1. Create a **new Angular component** called profile.
2. Add a name and email variable in profile.component.ts.
3. Use **interpolation ({{ }})** to display the name and email in profile.component.html.
4. Add a **button with (click) event** that changes the user's name dynamically.

5. Use ***ngIf** to hide/show an extra message based on a condition.

Conclusion

In this section, we explored:

✓ **How Angular components work and how to create them**.

✓ **How to define templates and use data binding ({{ }})**.

✓ **How directives like *ngIf and *ngFor improve UI flexibility**.

✓ **How Angular components improve code reusability and performance**.

COMPONENT LIFECYCLE HOOKS IN ANGULAR

CHAPTER 1: INTRODUCTION TO COMPONENT LIFECYCLE HOOKS

1.1 What are Component Lifecycle Hooks?

In **Angular**, components go through a **lifecycle of creation, update, and destruction**. **Lifecycle hooks** allow developers to execute **custom logic** at different stages of a component's lifecycle.

✓ **Monitors component initialization, rendering, and destruction**.
✓ **Executes custom logic at specific lifecycle stages**.
✓ **Improves performance by handling resources properly**.

1.2 Why Are Lifecycle Hooks Important?

✓ **Fetch data from APIs when a component loads**.
✓ **Run cleanup logic before a component is destroyed**.
✓ **Optimize performance by handling updates efficiently**.
✓ **Monitor changes in component inputs dynamically**.

Angular provides **several lifecycle hooks** that execute at different points in a component's life.

CHAPTER 2: OVERVIEW OF ANGULAR LIFECYCLE HOOKS

2.1 The Lifecycle Hooks in Angular

Lifecycle Hook	Description	When It Runs

ngOnChanges	Detects changes in @Input properties.	Before ngOnInit and whenever input changes.
ngOnInit	Initializes component logic.	Once after component is created.
ngDoCheck	Detects changes not caught by ngOnChanges.	Runs after ngOnInit and on every change detection cycle.
ngAfterContentInit	Runs after content projection (ng-content).	Once after first ngDoCheck.
ngAfterContentChecked	Runs after projected content updates.	Runs after every change detection cycle.
ngAfterViewInit	Runs after component's child views are initialized.	Once after first ngAfterContentChecked.
ngAfterViewChecked	Runs after the component's child views are updated.	Runs after every change detection cycle.

ngOnDestroy	Cleanup logic before the component is removed.	When a component is destroyed.

CHAPTER 3: IMPLEMENTING LIFECYCLE HOOKS IN ANGULAR

3.1 Using ngOnInit for Component Initialization

The ngOnInit hook is commonly used for:

✓ **Fetching data from APIs** when a component loads.
✓ **Initializing component properties**.
✓ **Setting up event listeners**.

Example: Using ngOnInit to Fetch Data

```
import { Component, OnInit } from '@angular/core';


@Component({
 selector: 'app-user',
 template: `<h1>Welcome, {{ user.name }}</h1>`
})
export class UserComponent implements OnInit {
 user = { name: '' };


 ngOnInit() {
  this.user.name = "John Doe"; // Simulating API call
 }
```

```
}
```

✓ The ngOnInit method **runs once** when the component is initialized.

3.2 Detecting Input Changes with ngOnChanges

The ngOnChanges hook runs **whenever an @Input property changes**.

Example: Watching Input Changes

```
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';


@Component({
 selector: 'app-profile',
 template: `<p>Username: {{ username }}</p>`
})
export class ProfileComponent implements OnChanges {
 @Input() username: string = '';

 ngOnChanges(changes: SimpleChanges) {
  console.log("Previous:", changes.username.previousValue);
  console.log("Current:", changes.username.currentValue);
 }
}
```

✓ Detects **whenever the username input changes**.

3.3 Detecting Custom Changes with ngDoCheck

If ngOnChanges **misses changes,** ngDoCheck can manually track them.

Example: Detecting Object Changes

```
import { Component, DoCheck } from '@angular/core';


@Component({
 selector: 'app-counter',
 template: `<p>Count: {{ count }}</p>`
})
export class CounterComponent implements DoCheck {
 count = 0;


 ngDoCheck() {
  console.log("Change detected!");
 }
}
```

✓ ngDoCheck runs **on every change detection cycle**.

3.4 Handling Projected Content with ngAfterContentInit & ngAfterContentChecked

If a component **receives projected content** (via <ng-content>), these hooks **monitor changes**.

Example: Using ngAfterContentInit

```
import { Component, AfterContentInit } from '@angular/core';


@Component({
 selector: 'app-message',
 template: `<ng-content></ng-content>`
})
export class MessageComponent implements AfterContentInit {
 ngAfterContentInit() {
  console.log("Projected content initialized!");
 }
}
```

✓ Runs **once after projected content is inserted**.

3.5 Handling Child Views with ngAfterViewInit & ngAfterViewChecked

These hooks **monitor and update child components**.

Example: Using ngAfterViewInit for DOM Manipulation

```
import { Component, ViewChild, AfterViewInit, ElementRef } from '@angular/core';
```

```
@Component({
 selector: 'app-greeting',
 template: `<h1 #greetingText>Welcome!</h1>`
})
export class GreetingComponent implements AfterViewInit {
 @ViewChild('greetingText') heading!: ElementRef;

 ngAfterViewInit() {
  this.heading.nativeElement.style.color = 'blue';
 }
}
```

✓ **Accesses and modifies DOM elements** after Angular initializes the view.

3.6 Cleaning Up with ngOnDestroy

The ngOnDestroy hook is **used to clean up resources** like:

✓ **Unsubscribing from Observables**.
✓ **Removing event listeners**.
✓ **Clearing intervals or timeouts**.

Example: Unsubscribing from an API Stream

```
import { Component, OnDestroy } from '@angular/core';
import { Subscription, interval } from 'rxjs';
```

```
@Component({
 selector: 'app-timer',
 template: `<p>Time: {{ time }}</p>`
})
export class TimerComponent implements OnDestroy {
 time = 0;
 private timerSubscription!: Subscription;

 constructor() {
  this.timerSubscription = interval(1000).subscribe(() =>
this.time++);
 }

 ngOnDestroy() {
  this.timerSubscription.unsubscribe();
  console.log("Component destroyed!");
 }
}
```

✓ Prevents **memory leaks** by cleaning up subscriptions.

Case Study: How a Social Media App Used Lifecycle Hooks to Improve Performance

Background

A **social media platform** needed:

✓ **Real-time updates for posts and comments**.
✓ **Efficient handling of dynamic UI changes**.
✓ **Prevention of memory leaks** when users navigate between pages.

Challenges

- **Performance slowdowns** due to frequent DOM updates.
- **Memory leaks** caused by unclosed subscriptions.
- **Inefficient handling of user interactions**.

Solution: Implementing Lifecycle Hooks

✓ Used ngOnInit for **fetching user posts**.
✓ Used ngAfterViewInit for **loading dynamic UI elements**.
✓ Used ngOnDestroy to **clean up WebSocket connections**.

Results

🚀 **50% faster UI updates**, improving user experience.
🔍 **Eliminated memory leaks**, reducing crashes.
⚡ **Optimized event handling**, leading to better performance.

By leveraging **lifecycle hooks**, the app handled **real-time data efficiently** while maintaining performance.

Exercise

1. Create a new Angular component and implement ngOnInit to log a message when initialized.
2. Use @Input and ngOnChanges to detect changes in a component's input property.

3. Implement ngOnDestroy to unsubscribe from an interval timer.
4. Modify a component's DOM using ngAfterViewInit.

Conclusion

In this section, we explored:

✓ **What lifecycle hooks are and how they control component behavior**.

✓ **How to use ngOnInit, ngOnChanges, and ngOnDestroy effectively**.

✓ **How lifecycle hooks help optimize Angular applications**.

WORKING WITH ANGULAR DIRECTIVES (STRUCTURAL & ATTRIBUTE)

CHAPTER 1: INTRODUCTION TO ANGULAR DIRECTIVES

1.1 What Are Directives in Angular?

Directives in Angular are **special instructions** used in templates to **modify the structure, behavior, or appearance** of elements in the DOM (Document Object Model).

✓ **Extends HTML functionalities** with Angular-specific behaviors.
✓ **Enhances dynamic rendering** without manipulating the DOM manually.
✓ **Reusable and maintainable**, improving application scalability.

Angular directives fall into three categories:

Type	Description	Example
Component Directives	Extend **HTML elements** with a component's logic.	<app-user></app-user>
Structural Directives	Modify the **layout** by adding, removing, or replacing elements.	*ngIf, *ngFor, *ngSwitch
Attribute Directives	Change the **appearance or behavior** of elements.	[ngClass], [ngStyle], customDirective

CHAPTER 2: STRUCTURAL DIRECTIVES

2.1 What Are Structural Directives?

Structural directives **modify the DOM** by **adding, removing, or manipulating** elements.

✓ Identified by the *** prefix** (e.g., *ngIf, *ngFor).
✓ Works by **modifying the DOM structure dynamically**.

2.2 *ngIf – Conditional Rendering

The *ngIf directive **conditionally renders elements** based on an expression.

Example: Displaying a Message If a Condition is True

```
<p *ngIf="isLoggedIn">Welcome, user!</p>
```

✓ If isLoggedIn is true, the paragraph **renders**; otherwise, it is **removed from the DOM**.

Example: Using else and then Blocks

```
<ng-container *ngIf="isLoggedIn; else notLoggedIn">
  <p>Welcome back!</p>
</ng-container>

<ng-template #notLoggedIn>
  <p>Please log in.</p>
</ng-template>
```

✓ Provides an **alternative UI** when the condition is false.

2.3 *ngFor – Looping Through Data

The *ngFor directive **iterates over arrays** and renders elements dynamically.

Example: Displaying a List of Users

```
<ul>
  <li *ngFor="let user of users">{{ user.name }}</li>
</ul>
```

✓ Iterates over users and displays each user.name.

Example: Accessing Index in *ngFor

```
<ul>
  <li *ngFor="let user of users; let i = index">
    {{ i + 1 }}. {{ user.name }}
  </li>
</ul>
```

✓ The **index variable (i)** keeps track of **iteration count**.

2.4 *ngSwitch – Multi-Condition Rendering

The *ngSwitch directive **displays one element based on a condition**.

Example: Displaying Different Messages Based on User Role

```
<div [ngSwitch]="userRole">
  <p *ngSwitchCase="'admin'">Admin Dashboard</p>
  <p *ngSwitchCase="'editor'">Editor Panel</p>
  <p *ngSwitchDefault>Guest View</p>
```

✓ Displays content based on userRole.
✓ ***ngSwitchDefault** handles unmatched cases.

CHAPTER 3: ATTRIBUTE DIRECTIVES

3.1 What Are Attribute Directives?

Attribute directives **modify the behavior or styling** of an element.

✓ Used as **element properties** ([directive]).
✓ **Does not change** the DOM structure.

3.2 ngClass – Applying CSS Classes Dynamically

The [ngClass] directive **binds CSS classes dynamically**.

Example: Applying Classes Based on a Condition

<p [ngClass]="{ 'highlight': isActive, 'disabled': !isActive }">Styled Text</p>

✓ If isActive is true, the highlight class applies; otherwise, disabled applies.

3.3 ngStyle – Dynamic Inline Styling

The [ngStyle] directive **binds styles dynamically**.

Example: Changing Text Color Based on a Condition

<p [ngStyle]="{ color: isError ? 'red' : 'green' }">Status Message</p>

✓ If isError is true, text appears **red**; otherwise, it appears **green**.

3.4 Creating a Custom Attribute Directive

Custom attribute directives allow developers to **create reusable styling or behavior changes**.

Step 1: Generate a Custom Directive

ng generate directive highlight

✓ This creates a highlight.directive.ts file.

Step 2: Implement the Directive Logic

Modify highlight.directive.ts to change text color:

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

@Directive({
 selector: '[appHighlight]'
})
export class HighlightDirective {
 constructor(private el: ElementRef, private renderer: Renderer2) {}

 @HostListener('mouseenter') onMouseEnter() {
  this.renderer.setStyle(this.el.nativeElement, 'color', 'blue');
 }

 @HostListener('mouseleave') onMouseLeave() {
```

```
  this.renderer.setStyle(this.el.nativeElement, 'color', 'black');
 }
}
```

✓ @HostListener('mouseenter') changes text color **on hover**.
✓ @HostListener('mouseleave') resets color when the mouse leaves.

Step 3: Use the Custom Directive in a Template

```
<p appHighlight>Hover over this text to change color.</p>
```

✓ The appHighlight directive applies **hover effects dynamically**.

Case Study: How a Retail Website Used Angular Directives for Dynamic UI

Background

A retail company needed a **dynamic, interactive UI** for its e-commerce site.

Challenges

✓ **Hardcoded elements** made the UI **difficult to maintain**.
✓ **No real-time updates** for stock availability.
✓ **Poor styling management**, requiring **manual CSS updates**.

Solution: Implementing Angular Directives

✓ **Used *ngIf for stock availability** → Products only showed "In Stock" if available.
✓ **Used *ngFor to dynamically list products** from an API.
✓ **Used ngClass for conditional styling** (e.g., discount labels).
✓ **Created a custom directive for hover effects** on product cards.

```
<div *ngIf="product.inStock; else outOfStock">
  <p [ngClass]="{ 'discount': product.onSale }">{{ product.name }}</p>
</div>

<ng-template #outOfStock>
  <p class="out-of-stock">Out of Stock</p>
</ng-template>
```

Results

🚀 **Faster updates** for dynamic product listings.
🎨 **Improved user experience** with interactive styling.
📈 **Scalable UI**, making future updates seamless.

By leveraging **Angular directives**, the company transformed its **UI into a dynamic, data-driven experience**.

Exercise

1. **Use *ngIf** to display a message if a user is logged in.
2. **Use *ngFor** to list names from an array.
3. **Use ngClass** to apply conditional styling to an element.
4. **Create a custom directive** that changes the background color on hover.

Conclusion

In this section, we explored:
✓ How structural directives (*ngIf, *ngFor, *ngSwitch) modify

the DOM.

✓ **How attribute directives (ngClass, ngStyle) enhance UI elements**.

✓ **How to create custom directives** for reusable UI behavior.

CREATING REUSABLE COMPONENTS IN ANGULAR

CHAPTER 1: INTRODUCTION TO REUSABLE COMPONENTS

1.1 What are Reusable Components?

Reusable components in Angular are **self-contained UI elements** that can be used multiple times across an application. They help developers **write cleaner, modular, and maintainable code** by reducing repetition and improving maintainability.

✓ **Encapsulation** – Each component has its own logic, styles, and templates.
✓ **Reusability** – Components can be used in multiple places without rewriting code.
✓ **Scalability** – Enhances project structure by making the application **modular**.

1.2 Why Use Reusable Components?

✓ **Reduces code duplication** – Write once, use everywhere.
✓ **Improves maintainability** – Makes debugging and updates easier.
✓ **Enhances consistency** – Ensures uniform design across the application.
✓ **Supports scalability** – Easily integrates into larger applications.

CHAPTER 2: SETTING UP A REUSABLE COMPONENT IN ANGULAR

2.1 Creating a Component Using Angular CLI

To generate a new component using Angular CLI, run:

ng generate component components/button

OR

ng g c components/button

✓ Creates a **button component** inside the components/ folder with:

- button.component.ts – Component logic.
- button.component.html – Component template.
- button.component.css – Component styling.
- button.component.spec.ts – Unit test file.

2.2 Basic Structure of a Reusable Component

Example: Creating a Reusable Button Component

```
import { Component, Input } from '@angular/core';


@Component({
 selector: 'app-button',
 template: `<button [ngClass]="color" (click)="handleClick()">{{ label }}</button>`,
 styleUrls: ['./button.component.css']
})
export class ButtonComponent {
 @Input() label: string = 'Click Me';
 @Input() color: string = 'primary';


 handleClick() {
```

```
  console.log(`${this.label} button clicked`);
 }
}
```

✓ Uses **@Input()** to accept dynamic values like label and color.
✓ The handleClick() method logs button clicks.

Using the Button Component in Another Component

```
<app-button label="Submit" color="success"></app-button>
<app-button label="Cancel" color="danger"></app-button>
```

✓ The button **adapts dynamically** based on the label and color properties.

CHAPTER 3: USING INPUT AND OUTPUT FOR COMPONENT REUSABILITY

3.1 Passing Data to a Component Using @Input()

@Input() allows **parent components to pass data** to child components.

Example: Input Property in a Card Component

```
import { Component, Input } from '@angular/core';

@Component({
 selector: 'app-card',
 template: `
  <div class="card">
```

```
    <h3>{{ title }}</h3>
    <p>{{ description }}</p>
  </div>
 `,
 styleUrls: ['./card.component.css']
})
export class CardComponent {
 @Input() title!: string;
 @Input() description!: string;
}
```

Using the Card Component

```
<app-card title="Angular Basics" description="Learn the fundamentals of Angular."></app-card>
<app-card title="Advanced Angular" description="Deep dive into Angular's advanced features."></app-card>
```

✓ Enables **dynamic card generation** without modifying the component.

3.2 Sending Events from Child to Parent Using @Output()

@Output() allows child components to **emit events** to the parent component.

Example: Creating an Event Emitter in a Toggle Switch Component

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

```
@Component({
 selector: 'app-toggle',
 template: `<button (click)="toggleState()">Toggle</button>`,
})
export class ToggleComponent {
 @Output() toggled = new EventEmitter<boolean>();
 isActive = false;

 toggleState() {
  this.isActive = !this.isActive;
  this.toggled.emit(this.isActive);
 }
}
```

Using the Toggle Component in a Parent Component

```
<app-toggle (toggled)="handleToggle($event)"></app-toggle>
handleToggle(event: boolean) {
 console.log("Toggle state:", event);
}
```

✓ Allows **two-way communication** between parent and child components.

CHAPTER 4: USING CONTENT PROJECTION (NG-CONTENT) FOR FLEXIBILITY

4.1 What is Content Projection?

Content projection allows developers to **pass custom content** inside a component using <ng-content>.

Example: Creating a Modal Component with Content Projection

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


@Component({
 selector: 'app-modal',
 template: `
  <div class="modal">
   <h3>Modal Header</h3>
   <ng-content></ng-content>
   <button (click)="close()">Close</button>
  </div>
 `,
 styleUrls: ['./modal.component.css']
})
export class ModalComponent {
 close() {
  console.log("Modal closed");
 }
```

}

✓ The <ng-content> tag allows inserting **custom content** inside the modal.

Using the Modal Component with Custom Content

<app-modal>

This is a dynamic message inside the modal.

</app-modal>

✓ **Enables customization** without modifying the component logic.

CHAPTER 5: OPTIMIZING REUSABLE COMPONENTS

5.1 Using Interfaces for Strong Typing

To maintain **type safety**, define an interface for data structures.

Example: Using an Interface in a User List Component

```
interface User {
  id: number;
  name: string;
  email: string;
}


@Component({
  selector: 'app-user-list',
  template: `
    <div *ngFor="let user of users">
```

```
    {{ user.name }} ({{ user.email }})
  </div>
 `
})
export class UserListComponent {
  @Input() users: User[] = [];
}
```

✓ Ensures that **only valid user objects** are passed to the component.

5.2 Lazy Loading Components for Performance

To improve **performance**, load components **only when needed** using lazy loading.

Example: Lazy Loading a Feature Module

Modify app-routing.module.ts:

```
const routes: Routes = [
  { path: 'dashboard', loadChildren: () =>
import('./dashboard/dashboard.module').then(m =>
m.DashboardModule) }
];
```

✓ Reduces **initial load time**, making applications faster.

Case Study: How a SaaS Company Scaled UI Development Using Reusable Components

Background

A **SaaS company** developing a project management tool faced:

✓ **Inconsistent UI elements across pages**.
✓ **Code duplication** due to repeated UI components.
✓ **High maintenance effort** for fixing UI bugs.

Challenges

- UI changes **required updates across multiple files**.
- **Redundant code** made debugging difficult.
- **Slow feature releases** due to manual UI implementation.

Solution: Implementing Reusable Angular Components

The team:

✓ Created **reusable components** (buttons, cards, modals).
✓ Used **@Input() and @Output()** for dynamic customization.
✓ Leveraged **lazy loading** for faster UI rendering.

Results

🚀 **Reduced development time by 50%**.
⚡ **Improved UI consistency and maintainability**.
🔧 **Faster feature rollout** with modular UI components.

By using **Angular reusable components,** the company **enhanced development efficiency** and **delivered a scalable UI**.

Exercise

1. Create a **reusable card component** with @Input() for title and content.

2. Build a **toggle switch component** that emits an event when clicked.
3. Implement a **modal component** using <ng-content>.
4. Use **interfaces** to enforce type safety in an @Input() component.

Conclusion

In this section, we explored:

✓ **How to create reusable components in Angular using CLI**.

✓ **How to use @Input(), @Output(), and ng-content for flexibility**.

✓ **How to optimize components with interfaces and lazy loading**.

ONE-WAY AND TWO-WAY DATA BINDING IN ANGULAR

CHAPTER 1: INTRODUCTION TO DATA BINDING IN ANGULAR

1.1 What is Data Binding?

Data Binding is the process of synchronizing data between the **TypeScript logic** and the **HTML template** in an Angular application. It allows users to **display dynamic data** and respond to user interactions efficiently.

✓ **Eliminates manual DOM manipulation**
✓ **Ensures a reactive UI**
✓ **Improves code maintainability**

1.2 Types of Data Binding in Angular

Angular provides **four types** of data binding:

Type	Description	Symbol
Interpolation	One-way binding from **TypeScript → HTML**	{{ }}
Property Binding	Binds component properties to HTML attributes	[ ]
Event Binding	Handles user actions like clicks and keypresses	( )
Two-way Binding	Syncs data between **TypeScript ↔ HTML**	[( )]

CHAPTER 2: ONE-WAY DATA BINDING IN ANGULAR

2.1 What is One-Way Data Binding?

One-way data binding updates the UI **from the component,** but changes in the UI do not affect the component.

✓ **Ensures better performance** by avoiding unnecessary updates
✓ **Prevents accidental modifications** of component data

There are **three types** of one-way data binding:

1. **Interpolation ({{ }}) – Binding Variables to the UI**
2. **Property Binding ([property]) – Binding Attributes**
3. **Event Binding ((event)) – Handling User Interactions**

2.2 Using Interpolation ({{ }})

Interpolation allows displaying **dynamic values** inside the HTML template.

Example: Using Interpolation

```
export class AppComponent {
  username: string = "Alice";
}
```

```
<h2>Welcome, {{ username }}!</h2>
```

✓ {{ username }} replaces **the variable value in the UI**.

2.3 Using Property Binding ([property])

Property binding allows binding **HTML attributes** to TypeScript properties dynamically.

Example: Binding an Image Source

```
export class AppComponent {
  imageUrl: string = "assets/logo.png";
}
```

```
<img [src]="imageUrl" alt="Logo">
```

✓ **The src attribute updates dynamically** based on imageUrl.

Other use cases:

```
<input [value]="username">
```

```
<button [disabled]="isDisabled">Click Me</button>
```

✓ **Ensures the UI reflects component values**.

2.4 Using Event Binding ((event))

Event binding listens for **user interactions** like clicks, keystrokes, or form submissions.

Example: Button Click Event

```
export class AppComponent {
  message: string = "Hello, User!";

  updateMessage() {
    this.message = "You clicked the button!";
  }
}
```

```
<h2>{{ message }}</h2>
```

```
<button (click)="updateMessage()">Click Me</button>
```

✓ Clicking the button **triggers the function,** updating the message dynamically.

CHAPTER 3: TWO-WAY DATA BINDING IN ANGULAR

3.1 What is Two-Way Data Binding?

Two-way data binding synchronizes data between the **component (.ts) and the UI (.html)**.

✓ Changes in the UI update the component
✓ Changes in the component update the UI
✓ Useful for forms and real-time user interactions

3.2 Using Two-Way Binding with [(ngModel)]

To use **two-way data binding,** we use **[(ngModel)],** which requires the FormsModule in app.module.ts.

Step 1: Import FormsModule in app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule], // Import FormsModule
  bootstrap: [AppComponent]
```

```
})
export class AppModule { }
```

Step 2: Implement Two-Way Binding in app.component.ts

```
export class AppComponent {
  userInput: string = "Type here...";
}
```

Step 3: Bind [(ngModel)] in app.component.html

```
<input [(ngModel)]="userInput">
<p>You typed: {{ userInput }}</p>
```

✓ Typing in the input field **updates userInput immediately**.

3.3 Two-Way Binding vs. One-Way Binding

Feature	One-Way Binding	Two-Way Binding
Data Flow	Component → UI	UI ↔ Component
Use Cases	Displaying data, event handling	Forms, user input fields
Performance	Faster	Slightly slower
Example	[value]="username"	[(ngModel)]="username"

CHAPTER 4: WHEN TO USE ONE-WAY VS. TWO-WAY DATA BINDING

4.1 When to Use One-Way Binding

✓ **Static data display** – When values do not change frequently.
✓ **Performance optimization** – Reduces unnecessary updates.
✓ **Reusable components** – Ensures controlled data flow.

✅ **Best for:**

- Displaying user profiles
- Showing product listings
- Rendering dynamic UI elements

4.2 When to Use Two-Way Binding

✓ **Real-time user interactions** – Forms, live search, editable text fields.
✓ **Instant feedback** – Updates model instantly when user inputs change.
✓ **Data-driven applications** – Forms with validation, chat apps, dashboards.

✅ **Best for:**

- Login forms
- Search filters
- User profile editors

Case Study: How an E-Commerce Platform Used Data Binding for Product Filters

Background

An **e-commerce company** needed a **real-time product filtering** system using Angular.

Challenges

✓ **Slow UI updates** due to manual DOM manipulation.
✓ **Delayed user feedback** when selecting product filters.
✓ **Complex form handling** for price range and category selection.

Solution: Implementing Two-Way Binding

✓ Used [(ngModel)] for **live search and filtering**.
✓ Used *ngFor and [value] to dynamically display products.
✓ Handled category selection dynamically using event binding.

Example: Implementing Live Product Search

```
export class ProductComponent {
  searchQuery: string = "";
  products = ["Laptop", "Smartphone", "Tablet", "Headphones"];

  getFilteredProducts() {
    return this.products.filter(p =>
p.toLowerCase().includes(this.searchQuery.toLowerCase()));
  }
}
<input [(ngModel)]="searchQuery" placeholder="Search products">
<ul>
  <li *ngFor="let product of getFilteredProducts()">{{ product }}</li>
</ul>
```

Results

🚀 **50% faster search results** with live filtering.
🔍 **Real-time UI updates,** improving user experience.
⚡ **Smooth navigation,** reducing backend queries.

By using **two-way data binding,** the e-commerce site improved **performance and usability**.

Exercise

1. Create a new **Angular component** called profile.
2. Add a username property and display it using **interpolation ({{ }})**.
3. Add an **input field** that updates username using [(ngModel)].
4. Add a **button** that resets the username when clicked.

Conclusion

In this section, we explored:
✓ **How One-Way ({{ }}, [property], (event)) and Two-Way ([(ngModel)]) Data Binding works in Angular**.
✓ **How to implement data binding using interpolation, property binding, and event binding**.
✓ **When to use one-way vs. two-way binding** for different scenarios.
✓ **How Angular improves real-time interactivity** with two-way binding.

HANDLING USER INPUT & EVENTS IN ANGULAR

CHAPTER 1: INTRODUCTION TO USER INPUT HANDLING IN ANGULAR

1.1 Why Handle User Input in Angular?

User interactions—such as clicking buttons, entering text, or selecting options—are **essential for dynamic web applications**. Angular provides **powerful event handling mechanisms** to:

✓ Capture and process **user input** efficiently.
✓ React dynamically to **clicks, keystrokes, form inputs, and gestures**.
✓ Improve user experience with **real-time updates**.

1.2 How Angular Handles Events

Angular provides **event binding** to listen for **user actions** and trigger functions in the component.

✓ Uses the **(event) syntax** to bind UI events to component methods.
✓ Supports **various events** like click, keyup, change, submit, etc.
✓ Works with **template-driven** and **reactive forms** for handling input.

CHAPTER 2: BINDING USER INPUT WITH EVENT LISTENERS

2.1 Using (event) Binding for User Actions

Example: Handling a Button Click Event

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

```
@Component({
 selector: 'app-click-demo',
 template: `<button (click)="showMessage()">Click Me</button>`
})
export class ClickDemoComponent {
 showMessage() {
  alert("Button clicked!");
 }
}
```

✓ **Binds** the click event to the showMessage() method.
✓ **Triggers an alert** when the button is clicked.

2.2 Using $event to Access Event Data

Angular allows **passing event data** to the component using $event.

Example: Logging Mouse Click Position

```
@Component({
 selector: 'app-click-position',
 template: `<button (click)="logClick($event)">Click to Log Position</button>`
})
export class ClickPositionComponent {
 logClick(event: MouseEvent) {
```

```
  console.log(`Clicked at: X=${event.clientX}, Y=${event.clientY}`);
 }
}
```

✓ Captures **mouse coordinates** when the button is clicked.

Chapter 3: Handling Keyboard and Input Events

3.1 Using keyup to Detect User Typing

Angular can capture **real-time input changes** with the keyup event.

Example: Displaying Live User Input

```
@Component({
 selector: 'app-keyup-demo',
 template: `<input (keyup)="updateMessage($event)" placeholder="Type something">
        <p>You typed: {{ message }}</p>`
})
export class KeyupDemoComponent {
 message: string = "";

 updateMessage(event: KeyboardEvent) {
  this.message = (event.target as HTMLInputElement).value;
 }
}
```

✓ **Dynamically updates text** as the user types.

3.2 Using keydown and keypress Events

Example: Detecting Specific Key Presses

```
@Component({
 selector: 'app-keypress-demo',
 template: `<input (keydown)="detectKey($event)" placeholder="Press Enter">`
})
export class KeypressDemoComponent {
 detectKey(event: KeyboardEvent) {
  if (event.key === "Enter") {
   alert("Enter key pressed!");
  }
 }
}
```

✓ **Executes logic when the Enter key is pressed**.

CHAPTER 4: TWO-WAY DATA BINDING FOR INPUT FIELDS

4.1 What is Two-Way Binding?

Two-way data binding **synchronizes input fields with component properties**.

✓ Uses the **[(ngModel)] directive**.
✓ Ensures real-time updates in both **view and component logic**.

4.2 Implementing Two-Way Data Binding

Example: Binding Input to a Component Property

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


@Component({
 selector: 'app-two-way-binding',
 template: `<input [(ngModel)]="name" placeholder="Enter your name">
        <p>Hello, {{ name }}!</p>`
})
export class TwoWayBindingComponent {
 name: string = "";
}
```

✓ Automatically **updates the UI when the user types**.

CHAPTER 5: HANDLING FORM SUBMISSIONS IN ANGULAR

5.1 Using (submit) to Process Forms

Angular allows form submission handling with the **(submit)** event.

Example: Handling a Basic Form Submission

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


@Component({
 selector: 'app-form-demo',
```

```
 template: `
   <form (submit)="submitForm()">
    <input [(ngModel)]="username" placeholder="Enter Username" required>
    <button type="submit">Submit</button>
   </form>
 `
})
export class FormDemoComponent {
 username: string = "";

 submitForm() {
   console.log("Form Submitted with username:", this.username);
 }
}
```

✓ Uses **(submit) event** to handle form submissions.
✓ Logs **user input to the console** on submission.

5.2 Preventing Default Form Behavior

By default, forms reload the page on submission. Use $event.preventDefault() to prevent this.

Example: Preventing Page Reload on Form Submission

```
submitForm(event: Event) {
```

```
 event.preventDefault(); // Prevents page refresh
 console.log("Form submitted successfully!");
}
```

✓ Ensures the form **submits without reloading the page**.

CHAPTER 6: ADVANCED EVENT HANDLING WITH @HOSTLISTENER

6.1 What is @HostListener?

The @HostListener decorator **listens for global events** (e.g., keyboard presses, window resizing).

Example: Detecting Window Resize

```
import { Component, HostListener } from '@angular/core';

@Component({
 selector: 'app-resize-listener',
 template: `<p>Window Width: {{ width }}px</p>`
})
export class ResizeListenerComponent {
 width: number = window.innerWidth;

 @HostListener('window:resize', ['$event'])
 onResize(event: Event) {
  this.width = (event.target as Window).innerWidth;
 }
```

}

✓ Dynamically **updates width** when the browser is resized.

Case Study: How an E-Commerce Website Used Angular for Real-Time Input Handling

Background

An **e-commerce platform** needed a **real-time search feature** for product filtering.

Challenges

✓ Slow response time in **search bar input**.
✓ Frequent API requests **slowing down performance**.
✓ Poor form validation leading to **incorrect submissions**.

Solution: Implementing Efficient Event Handling in Angular

✓ Used keyup to **update search results instantly**.
✓ Applied debounceTime() to **reduce API calls**.
✓ Used @HostListener to detect **window resize events** for mobile responsiveness.

Results

🚀 **Improved search speed by 60%,** enhancing user experience.
🔍 **Reduced API calls by 40%,** lowering server load.
⚡ **Optimized form validation,** preventing invalid submissions.

By using **Angular's event handling,** the platform achieved **faster performance and a better UX**.

Exercise

1. Create an input field that **logs text changes** using keyup.
2. Implement a **click event** that shows an alert when a button is clicked.
3. Prevent **form submission from refreshing the page**.
4. Use @HostListener to **detect window resizing** and display the new width.

Conclusion

In this section, we explored:

✓ **How to bind events (click, keyup, submit) in Angular.**

✓ **How to use $event to access event data.**

✓ **How to implement two-way data binding with [(ngModel)].**

✓ **How to optimize event handling using @HostListener.**

REACTIVE FORMS VS. TEMPLATE-DRIVEN FORMS IN ANGULAR

CHAPTER 1: INTRODUCTION TO FORMS IN ANGULAR

1.1 What Are Forms in Angular?

Forms are an essential part of web applications, allowing users to input and submit data. Angular provides two distinct ways to create and manage forms:

✓ **Template-Driven Forms** – Uses HTML and directives for simple form handling.
✓ **Reactive Forms** – Uses TypeScript-based form control for better flexibility and validation.

Both approaches help **bind user input to data models**, but they are used in different scenarios based on complexity and control requirements.

CHAPTER 2: UNDERSTANDING TEMPLATE-DRIVEN FORMS

2.1 What Are Template-Driven Forms?

Template-driven forms rely on **Angular directives** and are defined **within the HTML template**.

✓ **Easy to implement** – Uses standard HTML elements.
✓ **Best for simple forms** – Suitable for basic data entry.
✓ **Uses ngModel for two-way data binding**.
✓ **Automatic form validation using Angular directives**.

2.2 Setting Up a Template-Driven Form

Step 1: Import FormsModule in app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';  // Import FormsModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule],  // Add FormsModule
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 2: Create a Form in app.component.html

```
<form #userForm="ngForm" (ngSubmit)="submitForm(userForm)">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" [(ngModel)]="user.name" required>

  <label for="email">Email:</label>
```

```
 <input type="email" id="email" name="email"
[(ngModel)]="user.email" required>


 <button type="submit"
[disabled]="userForm.invalid">Submit</button>

</form>
```

✓ Uses **ngModel for two-way data binding**.
✓ Uses #userForm="ngForm" to track form state and validity.
✓ Disables submit button if the form is invalid ([disabled]="userForm.invalid").

Step 3: Handling Form Submission in app.component.ts

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


@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 user = { name: '', email: '' };


 submitForm(form: any) {
  console.log('Form Submitted!', form.value);
```

```
  }
}
```

✓ **Logs the form data** when submitted.
✓ Works **without complex TypeScript code**.

2.3 Advantages & Disadvantages of Template-Driven Forms

Advantages	Disadvantages
Easy to set up	**Difficult to scale** for large applications
Uses **two-way data binding**	**Limited control** over form inputs
Built-in validators	Harder to implement **dynamic form fields**

CHAPTER 3: UNDERSTANDING REACTIVE FORMS

3.1 What Are Reactive Forms?

Reactive Forms provide **full programmatic control** over form inputs using TypeScript.

✓ **Best for complex forms** – Handles dynamic form fields and nested structures.
✓ **Uses FormControl and FormGroup** – No need for ngModel.
✓ **More scalable and testable** – Works well in enterprise applications.
✓ **Explicit validation rules** – Defined in TypeScript.

3.2 Setting Up a Reactive Form

Step 1: Import ReactiveFormsModule in app.module.ts

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { ReactiveFormsModule } from '@angular/forms'; // Import ReactiveFormsModule

import { AppComponent } from './app.component';


@NgModule({

 declarations: [AppComponent],

 imports: [BrowserModule, ReactiveFormsModule], // Add ReactiveFormsModule

 providers: [],

 bootstrap: [AppComponent]

})

export class AppModule { }
```

Step 2: Define the Form in app.component.ts

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

import { FormGroup, FormControl, Validators } from '@angular/forms';


@Component({

 selector: 'app-root',
```

```
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 userForm = new FormGroup({
  name: new FormControl('', Validators.required),
  email: new FormControl('', [Validators.required, Validators.email])
 });

 submitForm() {
  console.log('Form Submitted!', this.userForm.value);
 }
}
```

✓ Uses **FormGroup** to manage multiple form controls.
✓ Uses **Validators** for validation rules.

Step 3: Create the Reactive Form in app.component.html

```
<form [formGroup]="userForm" (ngSubmit)="submitForm()">
 <label for="name">Name:</label>
 <input type="text" id="name" formControlName="name">
 <span *ngIf="userForm.get('name')?.invalid &&
userForm.get('name')?.touched">
  Name is required
```

```
  </span>


  <label for="email">Email:</label>

  <input type="email" id="email" formControlName="email">

  <span *ngIf="userForm.get('email')?.invalid &&
userForm.get('email')?.touched">

    Enter a valid email

  </span>


  <button type="submit"
[disabled]="userForm.invalid">Submit</button>

</form>
```

✓ Uses **formGroup** for reactive binding.
✓ Uses **formControlName** instead of ngModel.
✓ Displays **validation messages** dynamically.

3.3 Advantages & Disadvantages of Reactive Forms

Advantages	Disadvantages
Highly scalable	Requires **more boilerplate code**
Better for complex forms	Harder to set up for **small forms**
Explicit validation control	**No automatic two-way binding**
Easier unit testing	**More complex for beginners**

CHAPTER 4: KEY DIFFERENCES BETWEEN REACTIVE & TEMPLATE-DRIVEN FORMS

Feature	Template-Driven Forms	Reactive Forms
Form Creation	Defined in the **HTML template**	Defined in **TypeScript**
Form Binding	Uses ngModel for **two-way data binding**	Uses FormControl and FormGroup
Validation	Uses Angular **built-in directives** (required, minlength)	Uses **explicit validators** (Validators.required)
Best For	**Simple forms**	**Complex forms with dynamic fields**

Case Study: How a Healthcare App Used Reactive Forms for Patient Registration

Background

A **healthcare startup** needed a **secure and scalable** patient registration form.

Challenges

✓ **Handling complex form fields** (patient details, medical history).
✓ **Dynamic field validation** (date of birth, email, insurance details).
✓ **Tracking form changes in real time**.

Solution: Implementing Reactive Forms

✓ Used FormGroup for **nested form fields**.
✓ Applied Validators to enforce **required fields**.
✓ Implemented statusChanges to **track real-time input changes**.

Results

🚀 **50% faster form validation,** reducing user errors.
📈 **Scalable form structure,** allowing easy modifications.
🔍 **Better data integrity,** ensuring all patient records were valid.

By **switching to Reactive Forms,** the startup improved **user experience and form security**.

Exercise

1. Create a **Template-Driven Form** with fields: name, email, age.
2. Convert it into a **Reactive Form** using FormGroup.
3. Add **custom validation** to ensure age is greater than 18.
4. Display **error messages dynamically** for validation failures.

Conclusion

In this section, we explored:
✓ **How to implement Template-Driven and Reactive Forms**.
✓ **Key differences between both approaches**.
✓ **Best practices for choosing the right form type**.

FORM VALIDATION AND CUSTOM VALIDATORS IN ANGULAR

CHAPTER 1: INTRODUCTION TO FORM VALIDATION IN ANGULAR

1.1 Understanding Form Validation

Form validation ensures that user input meets required **data integrity and formatting** rules before submission. In **Angular,** form validation helps:

✓ **Prevent incorrect data entry** – Ensures users enter valid information.
✓ **Improve user experience** – Provides real-time feedback.
✓ **Enhance security** – Avoids SQL injection and malformed inputs.

1.2 Types of Forms in Angular

Angular provides **two types of forms** for handling user input:

Form Type	Description	Use Case
Template-driven Forms	Uses directives in the HTML template (ngModel)	Simple forms with minimal validation
Reactive Forms	Uses a TypeScript model (FormControl, FormGroup)	Complex, dynamic forms with advanced validation

✓ **Template-driven forms** are best for small applications.
✓ **Reactive forms** offer **better scalability, flexibility, and unit testing**.

CHAPTER 2: SETTING UP FORM VALIDATION IN ANGULAR

2.1 Creating a Simple Form with Template-Driven Validation

Example: Creating a Basic Login Form

```
<form #loginForm="ngForm" (ngSubmit)="onSubmit()">
 <label>Email:</label>
 <input type="email" name="email" [(ngModel)]="email" required
#emailInput="ngModel">
 <div *ngIf="emailInput.invalid && emailInput.touched">
  <p>Email is required and must be valid</p>
 </div>

 <label>Password:</label>
 <input type="password" name="password"
[(ngModel)]="password" required minlength="6"
#passwordInput="ngModel">
 <div *ngIf="passwordInput.invalid && passwordInput.touched">
  <p>Password must be at least 6 characters</p>
 </div>

 <button type="submit"
[disabled]="loginForm.invalid">Login</button>
</form>
```

✓ Uses **required**, **minlength**, and **email** validators.
✓ Disables the submit button when **form is invalid**.
✓ Shows **error messages dynamically** when fields are invalid.

2.2 Creating a Reactive Form with Validation

Step 1: Import ReactiveFormsModule

In app.module.ts:

```
import { ReactiveFormsModule } from '@angular/forms';


@NgModule({
 imports: [ReactiveFormsModule],
})
export class AppModule {}
```

Step 2: Define a Reactive Form in Component

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, Validators } from
'@angular/forms';


@Component({
 selector: 'app-register',
 templateUrl: './register.component.html',
})
export class RegisterComponent {
```

```
 registerForm = new FormGroup({

  username: new FormControl('', [Validators.required, Validators.minLength(4)]),

  email: new FormControl('', [Validators.required, Validators.email]),

  password: new FormControl('', [Validators.required, Validators.minLength(6)])

 });


 onSubmit() {

  console.log(this.registerForm.value);

 }

}
```

✓ Uses **Validators.required**, **Validators.email**, and **Validators.minLength()**.
✓ **Reactive form approach** allows greater flexibility and control.

2.3 Displaying Error Messages in Reactive Forms

Example: Displaying Validation Messages

```
<form [formGroup]="registerForm" (ngSubmit)="onSubmit()">

 <label>Username:</label>

 <input type="text" formControlName="username">

 <div *ngIf="registerForm.controls.username.invalid && registerForm.controls.username.touched">
```

```
  <p>Username must be at least 4 characters</p>
 </div>

 <label>Email:</label>
 <input type="email" formControlName="email">
 <div *ngIf="registerForm.controls.email.invalid &&
registerForm.controls.email.touched">
  <p>Email is required and must be valid</p>
 </div>

 <label>Password:</label>
 <input type="password" formControlName="password">
 <div *ngIf="registerForm.controls.password.invalid &&
registerForm.controls.password.touched">
  <p>Password must be at least 6 characters</p>
 </div>

 <button type="submit"
[disabled]="registerForm.invalid">Register</button>
</form>
```

✓ **Error messages appear dynamically** when fields are invalid.
✓ **Form submission is disabled** until inputs are valid.

CHAPTER 3: CREATING CUSTOM VALIDATORS IN ANGULAR

3.1 Why Use Custom Validators?

Built-in validators handle basic validation, but sometimes **custom logic** is required, such as:

✓ Checking if a **username is already taken**.
✓ Enforcing a **password complexity rule**.
✓ Validating **phone numbers** with specific formats.

3.2 Creating a Custom Validator Function

Example: Password Strength Validator

```
import { AbstractControl, ValidationErrors } from '@angular/forms';


export function passwordStrengthValidator(control:
AbstractControl): ValidationErrors | null {
  const value = control.value || '';
  if (!/[A-Z]/.test(value) || !/[0-9]/.test(value)) {
    return { passwordStrength: 'Password must contain at least one
uppercase letter and one number' };
  }
  return null;
}
```

✓ Checks if the password **contains at least one uppercase letter and one number**.
✓ Returns **an error message** if the condition fails.

3.3 Using a Custom Validator in a Reactive Form

Step 1: Apply the Custom Validator to the FormControl

```
import { passwordStrengthValidator } from
'./validators/password.validator';


this.registerForm = new FormGroup({

  password: new FormControl('', [Validators.required,
passwordStrengthValidator])

});
```

Step 2: Display Validation Message

```
<div
*ngIf="registerForm.controls.password.errors?.passwordStrength">

  <p>{{ registerForm.controls.password.errors.passwordStrength
}}</p>

</div>
```

✓ Shows **custom validation error messages dynamically**.

CHAPTER 4: ASYNCHRONOUS VALIDATORS FOR API VALIDATION

4.1 Why Use Asynchronous Validators?

Asynchronous validators are useful when **validating data from a server**, such as:

✓ Checking if an **email is already registered**.
✓ Validating a **username's availability**.

4.2 Creating an Async Validator

Example: Checking if Email is Already Taken

```
import { AbstractControl, ValidationErrors } from '@angular/forms';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { UserService } from '../services/user.service';

export function emailExistsValidator(userService: UserService) {
  return (control: AbstractControl): Observable<ValidationErrors | null> => {
    return userService.checkEmail(control.value).pipe(
      map((exists) => (exists ? { emailTaken: true } : null))
    );
  };
}
```

✓ Calls the backend to check if the email **already exists**.

4.3 Using an Async Validator in a Form

```
this.registerForm = new FormGroup({
  email: new FormControl('', [Validators.required, Validators.email], [emailExistsValidator(this.userService)])
});
```

✓ Ensures **real-time validation** for user registration.

Case Study: How an E-Learning Platform Improved User Registration with Form Validation

Background

An **e-learning platform** required a **secure and user-friendly** registration system.

✓ **Users entered weak passwords,** causing security risks.
✓ **Duplicate email registrations** led to database inconsistencies.
✓ **Unstructured forms resulted in poor user experience**.

Challenges

- Users **forgot required fields,** leading to **incomplete registrations**.
- Passwords were **too simple,** increasing security vulnerabilities.
- Manually verifying duplicate emails **slowed down registration**.

Solution: Implementing Angular Form Validation

The development team:

✓ Used **Reactive Forms** for structured input handling.
✓ Implemented **custom password validators** for security.
✓ Added **async email validation** to prevent duplicate accounts.

Results

🚀 **User registration completion increased by 40%**.
🔒 **Stronger password enforcement,** reducing account breaches.
⚡ **Faster onboarding process,** improving user satisfaction.

By leveraging **Angular form validation,** the platform ensured **data integrity and a smooth user experience**.

Exercise

1. Create a **login form** with email and password validation.
2. Implement a **custom validator** to ensure passwords contain a **special character**.
3. Use an **async validator** to check if a username already exists in the database.
4. Display **real-time validation messages** based on user input.

Conclusion

In this section, we explored:

✓ **How to implement template-driven and reactive form validation in Angular**.

✓ **How to create custom validators for specific business logic**.

✓ **How to use async validators for API-based checks**.

ASSIGNMENT:

DEVELOP A FORM-BASED ANGULAR APPLICATION WITH DYNAMIC INPUT VALIDATION.

SOLUTION GUIDE: DEVELOPING A FORM-BASED ANGULAR APPLICATION WITH DYNAMIC INPUT VALIDATION

Step 1: Set Up the Angular Project

First, create a new Angular project using **Angular CLI**:

```
ng new form-validation-app
cd form-validation-app
```

Then, install **Bootstrap** (optional, for styling):

```
npm install bootstrap
```

Modify angular.json to include Bootstrap:

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
]
```

Step 2: Import FormsModule and ReactiveFormsModule

Navigate to app.module.ts and **import the necessary modules**:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
```

```
import { AppComponent } from './app.component';


@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, ReactiveFormsModule],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

✓ FormsModule – Required for **Template-Driven Forms**.
✓ ReactiveFormsModule – Required for **Reactive Forms**.

Step 3: Create a User Form Using Reactive Forms

Modify app.component.ts to create the form and define validation rules.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```
export class AppComponent {
  userForm: FormGroup;
  submitted = false;

  constructor(private fb: FormBuilder) {
    this.userForm = this.fb.group({
      name: ['', [Validators.required, Validators.minLength(3)]],
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(6)]],
      age: ['', [Validators.required, Validators.min(18),
Validators.max(65)]]
    });
  }

  // Getter methods for form controls
  get f() { return this.userForm.controls; }

  onSubmit() {
    this.submitted = true;
    if (this.userForm.valid) {
      console.log("Form Submitted Successfully!",
this.userForm.value);
    }
```

```
 }


 resetForm() {
  this.userForm.reset();
  this.submitted = false;
 }
}
```

✓ **Form validation rules**:

- name: Required, at least **3 characters**.
- email: Required, must be a **valid email format**.
- password: Required, minimum **6 characters**.
- age: Required, must be between **18 and 65**.

Step 4: Create the Form UI in HTML

Modify app.component.html to include the form fields and dynamic validation messages.

```
<div class="container mt-5">
 <h2 class="text-center">User Registration Form</h2>


 <form [formGroup]="userForm" (ngSubmit)="onSubmit()">


  <!-- Name Field -->
  <div class="form-group">
```

```
    <label>Name</label>

    <input type="text" class="form-control"
formControlName="name">

    <small class="text-danger" *ngIf="submitted &&
f.name.errors?.required">Name is required</small>

    <small class="text-danger" *ngIf="submitted &&
f.name.errors?.minlength">Minimum 3 characters required</small>

   </div>


   <!-- Email Field -->

   <div class="form-group">

    <label>Email</label>

    <input type="email" class="form-control"
formControlName="email">

    <small class="text-danger" *ngIf="submitted &&
f.email.errors?.required">Email is required</small>

    <small class="text-danger" *ngIf="submitted &&
f.email.errors?.email">Invalid email format</small>

   </div>


   <!-- Password Field -->

   <div class="form-group">

    <label>Password</label>

    <input type="password" class="form-control"
formControlName="password">
```

```
    <small class="text-danger" *ngIf="submitted &&
f.password.errors?.required">Password is required</small>

    <small class="text-danger" *ngIf="submitted &&
f.password.errors?.minlength">Minimum 6 characters
required</small>

  </div>


  <!-- Age Field -->

  <div class="form-group">

    <label>Age</label>

    <input type="number" class="form-control"
formControlName="age">

    <small class="text-danger" *ngIf="submitted &&
f.age.errors?.required">Age is required</small>

    <small class="text-danger" *ngIf="submitted &&
f.age.errors?.min">Minimum age is 18</small>

    <small class="text-danger" *ngIf="submitted &&
f.age.errors?.max">Maximum age is 65</small>

  </div>


  <!-- Submit and Reset Buttons -->

  <button type="submit" class="btn btn-primary mt-
3">Submit</button>

  <button type="button" class="btn btn-secondary mt-3 ml-2"
(click)="resetForm()">Reset</button>
```

```
</form>

<!-- Success Message -->
<div *ngIf="submitted && userForm.valid" class="alert alert-success mt-3">
  Form submitted successfully!
</div>
</div>
```

✓ Uses **Bootstrap classes** for styling.
✓ Displays **error messages dynamically** when input is invalid.
✓ Shows a **success message** when the form is submitted.

Step 5: Run the Angular Application

Start the development server:

```
ng serve
```

✓ Open http://localhost:4200/ in the browser.
✓ Test the form by entering **valid and invalid data**.

Step 6: Enhancing Validation with Custom Error Messages

Modify app.component.ts to add **custom validation messages** for each field.

```
getErrorMessage(field: string) {
  if (this.f[field].hasError('required')) {
```

```
    return `${field} is required`;
  } else if (this.f[field].hasError('minlength')) {
    return `${field} must be at least ${this.f[field].errors?.minlength.requiredLength} characters`;
  } else if (this.f[field].hasError('email')) {
    return `Invalid email format`;
  } else if (this.f[field].hasError('min')) {
    return `Minimum allowed value is ${this.f[field].errors?.min.min}`;
  } else if (this.f[field].hasError('max')) {
    return `Maximum allowed value is ${this.f[field].errors?.max.max}`;
  }
  return '';
}
```

Modify app.component.html to use the error message dynamically:

```
<small class="text-danger" *ngIf="submitted && f.name.errors">{{ getErrorMessage('name') }}</small>
```

✓ **Enhances readability** by centralizing error messages.

Case Study: How a Startup Improved Form Validation Using Angular

Background

A fintech startup needed a **secure and user-friendly registration form** for their banking app.

Challenges

✓ Users entered **incorrect email formats,** causing login failures.
✓ Many **passwords were weak,** increasing security risks.
✓ Manual data validation **slowed down registration**.

Solution: Implementing Dynamic Validation in Angular

✓ Used **Reactive Forms with real-time validation**.
✓ Implemented **custom error messages** for a better user experience.
✓ Integrated **password strength checks** to ensure security.

Results

🚀 **Form submission errors reduced by 80%**.
🔍 **Improved user experience** with instant feedback.
⚡ **Faster registration process,** increasing conversions.

By using **Angular Reactive Forms,** the company ensured **high-quality user data and security**.

Exercise

1. Add a **phone number field** with a pattern validation.
2. Add a **checkbox** for terms & conditions with validation.
3. Display a **message when the user enters an invalid email**.
4. Ensure **passwords must contain at least one number**.

Conclusion

In this assignment, we:
✓ Created a **dynamic form using Reactive Forms**.

✓ Implemented **real-time validation** for better user feedback.

✓ Used **Bootstrap for UI styling**.

✓ Improved **form usability and security**.