Fundamentals of Angular (v17) for Single Page Applications

After completing this unit, the learner will be able to **Create and configure** a scalable Angular application using **Angular CLI** with **Standalone Components**, following modern Angular architecture practices.

1.1 Angular Recap & Project Setup

1.1.1 Brief Recap of Angular 17 core concepts: Components, Services, Routing

Angular is a **TypeScript-based front-end framework** used to build **Single Page Applications (SPA)**. In Angular 17, the architecture is **component-driven**, **service-oriented**, and **router-based**.

Components in Angular 17

A **Component** is the **basic building block** of an Angular application.
It controls a **part of the user interface (UI)** and handles **presentation logic**. It is like a brain of the whole Angular application.

An Angular component consists of:

1. **HTML Template** – View (UI)
2. **TypeScript Class** – Logic
3. **CSS/SCSS** – Styling
4. **Decorator (`@Component`)** – Metadata

**In Angular, decorators are functions that use the @ symbol to attach metadata to classes, properties, methods, and parameters. This metadata tells Angular how to process and use these parts of the application, effectively turning plain TypeScript classes into Angular components, services, or modules.*

Example -

```
@Component({
  selector: 'app-header',
  standalone: true,
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent {
  title = 'Angular 17 App';
}
```

Angular 17 promotes **Standalone Components.** No need for `NgModule` for every component. Components are **reusable and modular**

Data binding - Data binding connects your component's state to the template(HTML).

- Interpolation `{{ }}`
- Property binding `[ ]`
- Event binding `( )`
- Two-way binding `[( )]`

Interpolation

In Angular, interpolation is a one-way data-binding technique that allows you to display dynamic data from your component's TypeScript logic within your HTML template. It uses double curly braces {{ }} as delimiters for expressions.

Description	Component (.ts)	Template (.html)
Displaying a property	`title = "Welcome to my app";`	`<h1>{{ title }}</h1>`
Mathematical operation	`num1 = 10; num2 = 5;`	`<p>The sum is: {{ num1 + num2 }}</p>`
Invoking a method	`getName() { return "Maria"; }`	`<p>User: {{ getName() }}</p>`

Property Binding

Property binding in Angular is a one-way data binding mechanism that sets a Document Object Model (DOM) element's property to a component's property value. It allows you to dynamically control HTML elements, component properties, and directives using data from your TypeScript code.

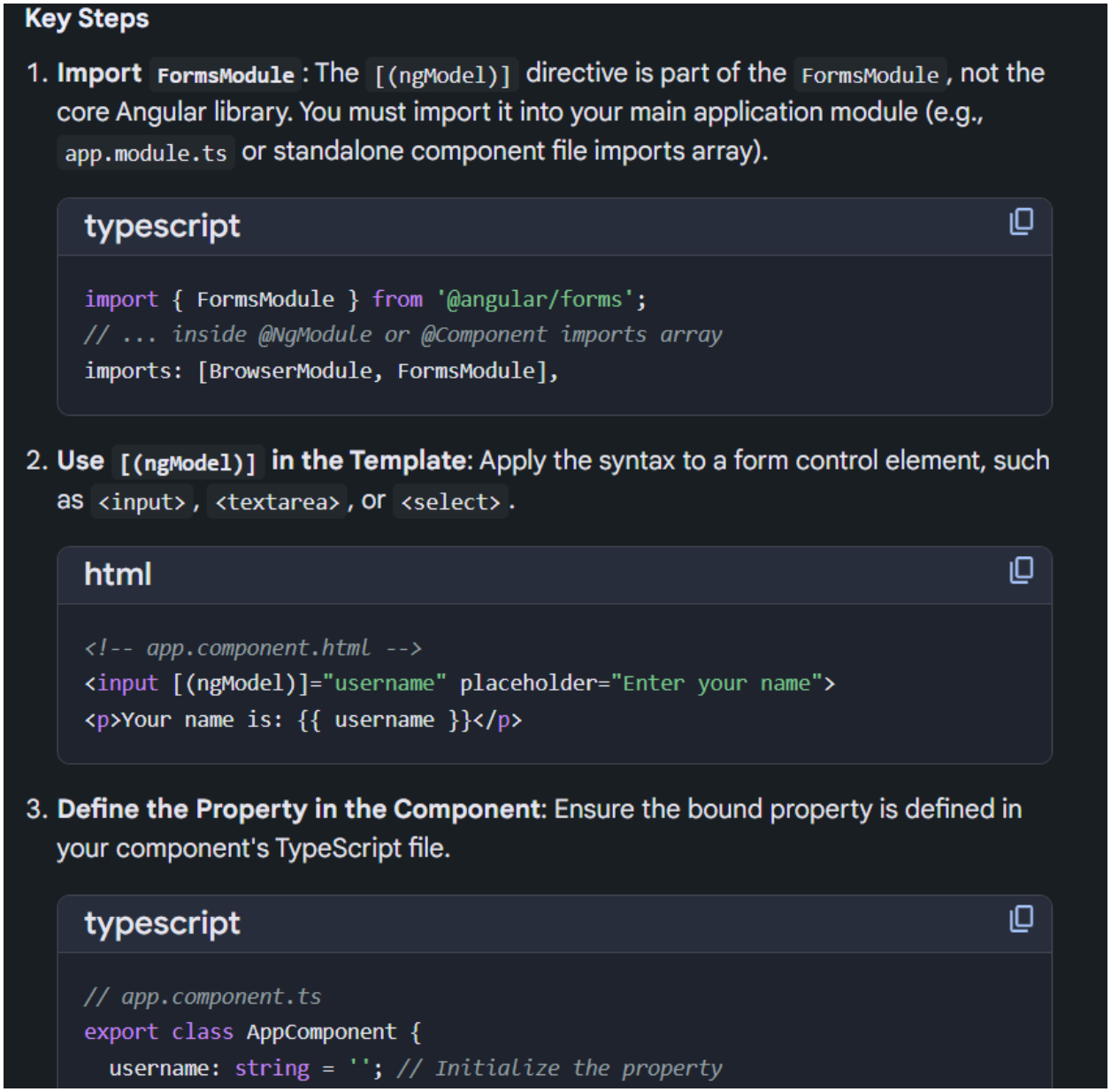
Event Binding

Event binding in Angular is a mechanism that allows you to listen for and respond to user actions, such as keystrokes, mouse clicks, and touches, in the view (template) and execute corresponding logic in the component's TypeScript code. It's a key part of one-way data flow from the view to the component.

Two way data binding

Two-way binding in Angular is a mechanism that provides automatic, bidirectional synchronization of data between the component's model (TypeScript class) and the view (HTML template).

This means: When data in the component changes, the view automatically reflects that change. When the user interacts with the view (e.g., types into an input field), the component's underlying data is updated immediately.



Key Steps

1. **Import `FormsModule`**: The `[(ngModel)]` directive is part of the `FormsModule`, not the core Angular library. You must import it into your main application module (e.g., `app.module.ts` or standalone component file imports array).

typescript

```typescript
import { FormsModule } from '@angular/forms';
// ... inside @NgModule or @Component imports array
imports: [BrowserModule, FormsModule],
```

2. **Use `[(ngModel)]` in the Template**: Apply the syntax to a form control element, such as `<input>`, `<textarea>`, or `<select>`.

html

```html
<!-- app.component.html -->
<input [(ngModel)]="username" placeholder="Enter your name">
<p>Your name is: {{ username }}</p>
```

3. **Define the Property in the Component**: Ensure the bound property is defined in your component's TypeScript file.

typescript

```typescript
// app.component.ts
export class AppComponent {
  username: string = ''; // Initialize the property
```

Services in Angular 17

A **Service** is a class that contains **business logic**, **data handling**, or **API communication**, separate from the UI.

Why Services are Needed :

- Separation of concerns
- Code reusability
- Cleaner components
- Centralized logic

Dependency Injection (DI)

Angular uses **Dependency Injection** to provide services to components.

`constructor(private userService: UserService) {}` // this is how we inject a dependency or service

Common Uses of Services

- HTTP API calls
- Authentication
- State management
- Utility functions

Benefits

- Singleton by default - *a service for which only one instance exists and is shared across the entire application*
- Easy testing
- Loose coupling

Routing in Angular 17

Routing allows navigation between different views/components **without reloading the page**, enabling SPA behavior.

Routing Example - in route.ts file

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent }
];
```

Router Features

- Lazy loading
- Route guards
- Route resolvers
- Parameterized routes

Benefits of Routing

- Faster navigation
- Better user experience
- Bookmarkable URLs
- Secure navigation

1.1.2 Setting up a scalable Angular project using Angular CLI with Standalone Components

Angular 17 introduces a **modern, simplified, and scalable project setup** using **Angular CLI** and **Standalone Components**, reducing boilerplate code and improving maintainability.

Angular CLI (Command Line Interface)

Angular CLI is an official command-line tool used to:

- Create Angular projects
- Generate components, services, routes
- Build, test, and deploy applications

Creating an Angular Project (Angular 17)

bash Copy code

```
ng new my-angular-app
```

During setup:

- Choose **Standalone Components (Yes)**
- Routing: **Yes**
- Styles: CSS / SCSS

What CLI Does Automatically

- Creates folder structure
- Configures TypeScript
- Sets up routing
- Adds build & environment configs

Standalone Components (Core Concept)

A **Standalone Component** is a component that **does not require an NgModule** and can be used independently.

Standalone Component Example

ts

```ts
@Component({
  selector: 'app-home',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './home.component.html'
})
export class HomeComponent {}
```

Why Standalone Components are Better

- Less boilerplate code
- Faster learning curve
- Modular architecture
- Better tree-shaking
- Improved performance

Scalable Project Folder Structure

Routing with Standalone Components(Angular 17)

```
export const routes: Routes = [

  { path: '', loadComponent: () =>
import('./features/home/home.component').then(c =>
c.HomeComponent) }

];
```

Advantages

- Easy lazy loading
- Faster initial load
- Feature isolation

Generating Components & Services using CLI

Components can be generated with the help of following command -

```
ng generate component <component-name>
```

Services can be generated with the help of following command -

```
ng generate service <service-name>
```

Advantages of Using Angular CLI + Standalone Components

Feature	Benefit
Angular CLI	Automation & consistency
Standalone Components	Modular & reusable
Feature-based folders	Scalability
Lazy loading	Performance
Clean bootstrap	Maintainability

1.1.3 Folder structure and module organization for large projects

As Angular applications grow in size, **proper folder structure and module organization** become critical for **scalability, maintainability, performance, and team collaboration**.

Angular 17 encourages a **feature-based and modular architecture**, especially when using **Standalone Components**.

Core Folder

Contains singleton services such as authentication, guards, interceptors, and global models. These services are loaded only once

Feature Folders

Each feature like authentication or dashboard has its own components and routing. Features can be lazy loaded for performance.

Shared Folder

Contains reusable UI components, pipes, directives, and utilities. Should not contain stateful services.

Routing Organization

Routing is organized centrally in app.routes.ts and feature-wise in individual route files to support lazy loading.

Benefits - Scalability, maintainability, performance optimization, team collaboration, and reusability.

1.2 Advanced Routing & State Handling

1.2.1 Implementing Lazy Loading with Feature Modules

What is Lazy Loading?

Lazy Loading is a technique in Angular where feature modules are loaded only when they are required, instead of loading the entire application at once.

Why Lazy Loading is Needed

In large applications:

- Many features (Admin, Dashboard, Reports, etc.)
- Large bundle size
- Slow initial load time

Problems Without Lazy Loading -

- Long loading time
- Poor performance
- Bad user experience

Benefits of Lazy Loading

Benefit	Explanation
Faster initial load	Only core module loads
Better performance	Smaller JS bundle
Scalability	Easy to add features
Optimized memory usage	Load when needed

Eager Loading vs Lazy Loading

Eager Loading	Lazy Loading
All modules load at start	Modules load on demand
Slow startup	Fast startup
Not scalable	Highly scalable

Feature Modules in Angular 17

A Feature Module represents a specific business feature of the application.

Examples:

- AuthModule
- AdminModule
- DashboardModule

Each feature module contains:

- Components
- Services
- Routing

Steps to Implement Lazy Loading

Step 1: Create Feature Module

```
ng generate module features/admin --route admin
--module app
```

Step 2: Feature Module Code

```
@NgModule({
  declarations: [AdminComponent],
  imports: [
    CommonModule,
    RouterModule.forChild(adminRoutes)
  ]
})
export class AdminModule {}
```

Step 3: Feature Routing

```
const adminRoutes: Routes = [
  { path: '', component: AdminComponent }
];
```

Step 4: Lazy Load in App Routing

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

When to Use Lazy Loading?

✔ Large applications
✔ Feature-based architecture
✔ Multiple routes
✔ Performance optimization

1.2.2 Route Guards: CanActivate, CanDeactivate for securing routes

What are Route Guards?

Route Guards are Angular services that control whether a user is allowed to navigate to or away from a route.

Guards help in **securing routes** and **controlling navigation** based on conditions like authentication, authorization, or unsaved data. They are like security guards for your application.

Why Route Guards are Needed

Without guards:

- Any user can access protected pages
- Sensitive routes remain unsecured
- Data loss may occur due to accidental navigation

Use Cases

- Protect dashboard pages
- Restrict admin access
- Warn users about unsaved form data
- Role-based access control

Types of Route Guards

Angular provides multiple guards, but **syllabus focus** is on:

- **CanActivate**
- **CanDeactivate**

CanActivate Guard

CanActivate decides whether a user **can access a route or not**.

When to Use

- Login required pages
- Admin-only sections
- Subscription-based content

Working

- Executes **before route is activated**
- Returns:
    - `true` → allow navigation
    - `false` → block navigation
    - `UrlTree` → redirect

CanActivate Example

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {

  canActivate(): boolean {
    return localStorage.getItem('token') !== null;
  }
}
```

Applying CanActivate in Routes

```
{
  path: 'dashboard',
  component: DashboardComponent,
  canActivate: [AuthGuard]
}
```

CanDeactivate Guard

CanDeactivate decides whether a user **can leave a route or not**.

When to Use

- Forms with unsaved data
- Edit pages
- Multi-step forms

This prevents **accidental navigation** and **data loss**.

CanDeactivate Example

```
export interface CanComponentDeactivate {
  canDeactivate: () => boolean;
}

@Injectable({ providedIn: 'root' })
export class UnsavedGuard
  implements CanDeactivate<CanComponentDeactivate> {

  canDeactivate(component: CanComponentDeactivate):
boolean {
    return component.canDeactivate();
  }
}
```

Applying CanDeactivate in Routes

```
{
  path: 'edit-profile',
  component: EditProfileComponent,
  canDeactivate: [UnsavedGuard]
}
```

Real-Life Analogy

- **CanActivate** → Security guard at building entry
- **CanDeactivate** → Warning before leaving exam hall without submitting paper

1.2.3 Route Resolvers for preloading data

What is a Route Resolver?

A Route Resolver is an Angular service that fetches required data before a route is activated.

Why Are Route Resolvers Needed?

Problem Without Resolvers

- Component loads first
- Data is fetched later
- User sees empty UI or loading spinner

Advantages of Using Resolvers

Advantage	Description
Better User Experience	No blank screen
Clean Components	No loading logic
Data Consistency	Data always available
Controlled Navigation	Route waits for data

How Route Resolvers Work

Working Flow

1. User navigates to a route
2. Resolver is triggered
3. Data is fetched from service/API
4. Route gets activated
5. Component loads with data

Creating a Route Resolver (Step-by-Step)

Step 1: Create Resolver Service

```
@Injectable({ providedIn: 'root' })
export class UserResolver implements Resolve<any> {

  constructor(private userService: UserService) {}

  resolve() {
    return this.userService.getUsers();
  }
}
```

Step 2: Attach Resolver to Route

```
{
  path: 'users',
  component: UserListComponent,
  resolve: { users: UserResolver }
}
```

Step 3: Access Resolved Data in Component

```
constructor(private route: ActivatedRoute) {}
ngOnInit() {
  this.users = this.route.snapshot.data['users'];
}
```

6. Route Resolver vs API Call in Component

Route Resolver	API Call in Component
Data loaded before route	Data loaded after view
Cleaner UI	Needs loaders
Better UX	UI flicker
Controlled navigation	No control

Route Guard	Route Resolver
Controls access	Preloads data
Returns boolean	Returns data
Used for security	Used for data

Real-Life Example

Route Resolver is like **preparing documents before a meeting**, so the meeting can start without delay.

1.2.4 Introduction to advanced state handling using RxJS Subjects and Behavior Subjects

What is State in an Angular Application?

State refers to the data that represents the current condition of an application and is shared across multiple components.

Examples:

- Logged-in user information
- Theme (dark/light)
- Cart items

Why State Management is Required?

Without proper state handling:

- Data duplication occurs
- Components become tightly coupled
- Inconsistent UI state
- Difficult debugging

Benefits of State Management

- Centralized data
- Consistent UI
- Better scalability
- Easy component communication

What is RxJS? note:read from assignn

Definition

RxJS (Reactive Extensions for JavaScript) is a library used in Angular to work with **asynchronous data streams**.

Angular uses RxJS extensively for:

- HTTP requests
- Event handling
- State management

What is a Subject in RxJS?

Definition

A **Subject** is a special type of Observable that:

- Can **emit values**
- Can be **subscribed to**
- Allows **multiple subscribers**

Characteristics of Subject

- Does not store previous value
- New subscribers get values only after subscription
- Useful for event-based communication

What is a BehaviorSubject?

A BehaviorSubject is a type of Subject that:

- Requires an initial value
- Always stores the latest emitted value
- Emits the current value to new subscribers

Why BehaviorSubject is Preferred?

- Maintains application state
- Ideal for shared data
- Ensures data availability

BehaviorSubject Example

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  isLoggedIn$ = new BehaviorSubject<boolean>(false);

  login() {
    this.isLoggedIn$.next(true);
  }

  logout() {
    this.isLoggedIn$.next(false);
  }
}
```

1.3 Reactive Forms in Real Applications

Reactive Forms are a model-driven approach to handling form inputs in Angular, where the form structure and validation logic are defined in the component class using TypeScript.

Reactive forms provide **better control, scalability, testability, and predictability** compared to template-driven forms.

Why Use Reactive Forms in Real Applications?

In real-world applications, forms are:

- Dynamic
- Complex
- Highly validated
- Integrated with APIs

Advantages

Feature	Benefit
Model-driven	Better control
Synchronous access	Instant validation
Scalable	Large forms
Testable	Unit testing easy
Dynamic	Runtime changes

Building Blocks of Reactive Forms

3.1 FormControl

Represents a single form input.

```
new FormControl('')
```

3.2 FormGroup

Represents a group of form controls.

```
new FormGroup({
  name: new FormControl(''),
  email: new FormControl('')
});
```

3.3 FormArray

Represents a **dynamic collection of controls** (important for real apps).

Dynamic Form Generation using FormArray

What is FormArray?

FormArray is used when:

- Number of inputs is not fixed
- Inputs need to be added/removed dynamically

Examples:

- Phone numbers
- Skills
- Addresses
- Dynamic questionnaires

FormArray Example

```
this.userForm = new FormGroup({
  name: new FormControl(''),
  phones: new FormArray([])
});
```

Adding Controls Dynamically

```
addPhone() {
  (this.userForm.get('phones') as FormArray)
    .push(new FormControl(''));
}
```

Why FormArray is Important

- Handles dynamic UI
- Clean structure
- Real-time form updates

1.3.2 Custom Validators and Asynchronous Validation

Custom Validators

Custom validators are **user-defined validation functions** used when built-in validators are not sufficient.

Custom Validator Example

```
export function noSpecialChars(control: AbstractControl) {
  return /[^a-zA-Z0-9]/.test(control.value)
    ? { specialChar: true }
    : null;
}
```

Asynchronous Validators

Asynchronous validators validate data **by calling a server/API**, such as:

- Email already exists
- Username availability

Async Validator Example

```
emailExists(control: AbstractControl) {
  return this.http.get(`/check-email/${control.value}`);
}
```

Difference

Custom Validator	Async Validator
Synchronous	Asynchronous
Immediate result	API dependent
Local logic	Server-side check

1.3.3 Centralized Error Handling and Validation Messages

Need for Centralized Error Handling

Without centralization:

- Repeated error code
- Messy templates
- Hard to maintain

Centralized Error Function

```
getErrorMessage(controlName: string): string {
  const control = this.form.get(controlName);
  if (control?.hasError('required')) return 'Field is required';
  if (control?.hasError('email')) return 'Invalid email';
  return '';
}
```

Benefits

- Clean templates
- Reusable logic
- Easy maintenance

1.3.4 Submitting Forms to APIs and Form State Management

Form Submission Flow

```
User Input
   ↓
Reactive Form
   ↓
Service (HttpClient)
   ↓
Backend API
   ↓
Response Handling
```

Submitting Form Data

```
onSubmit() {
  if (this.form.valid) {
    this.apiService.save(this.form.value).subscribe();
  }
}
```

Form State Properties

Property	Meaning
valid	Form passes validation
invalid	Validation fails
touched	User interacted
dirty	Value changed
pristine	No change

Why Form State Management Matters

- Disable submit button
- Show validation messages
- Improve UX

Reactive Forms vs Template-Driven Forms

Reactive Forms	Template-Driven Forms
Model-driven	Template-driven
Better scalability	Suitable for small forms
Easy testing	Hard testing
Complex validation	Limited validation

1.4 Reactive Forms in Real Applications

Modern Angular applications focus on **reusability, modularity, and clean architecture**.
Reusable UI components and proper design patterns help in building **scalable and maintainable applications**.

What are Reusable UI Components?

A **reusable UI component** is a generic component that can be **used multiple times across the application** with different data and behavior.

Examples:

- Cards
- Modals
- Alerts
- Buttons
- Tables

Why Reusability is Important

- Reduces code duplication
- Improves consistency in UI
- Easier maintenance
- Faster development

1.4.1 Creating Reusable Card, Modal, and Alert Components

A. Reusable Card Component

Purpose

Used to display content in a structured layout.

Key Concepts Used

- `@Input()` for data passing
- Generic template

```
@Component({
  selector: 'app-card',
  standalone: true,
  template: `
    <div class="card">
      <h3>{{ title }}</h3>
      <ng-content></ng-content>
    </div>
  `
})
export class CardComponent {
  @Input() title!: string;
}
```

B. Reusable Modal Component

Purpose

Used for popups, confirmations, forms, etc.

Key features:

- Content projection
- Event emission

```
@Output() close = new EventEmitter<void>();
```

C. Reusable Alert Component

Purpose

Used to show success, error, or warning messages.

```
@Input() type: 'success' | 'error' | 'warning';
@Input() message: string;
```

Benefits of Reusable Components

- One component → many use cases
- UI consistency
- Easy customization

1.4.2 Component Interaction with RxJS and Shared Services

Problem with Direct Component Communication

- Tight coupling
- Hard to scale
- Not suitable for distant components

Solution: Shared Services with RxJS

How It Works

- Shared service holds data
- RxJS `Subject` / `BehaviorSubject` used
- Multiple components subscribe to changes

1.4.3 Use of ng-template, ng-container, ng-content

These Angular directives provide **structural flexibility** and **dynamic UI control**.

A. ng-template

Purpose

Defines a **template that is not rendered immediately**.

Used for:

- Conditional rendering
- Dynamic templates

```
<ng-template #loading>
  <p>Loading...</p>
</ng-template>
```

B. ng-container

Purpose

Acts as a **logical container** without adding extra DOM elements.

```
<ng-container *ngIf="isLoggedIn">
  <p>Welcome User</p>
</ng-container>
```

C. ng-content

Purpose

Used for **content projection** (passing HTML from parent to child).

```
<ng-content></ng-content>
```

1.4.4 Smart vs Dumb Components

A. Smart Components (Container Components)

Definition

Smart components:

- Handle business logic
- Call APIs
- Manage state
- Interact with services

Characteristics

- Less reusable
- More complex
- Knows *how* things work

B. Dumb Components (Presentational Components)

Definition

Dumb components:

- Focus only on UI
- Receive data via `@Input()`
- Emit events via `@Output()`

Characteristics

- Highly reusable
- Easy to test
- No service dependency

1.5 Application Deployment & Performance Optimization

After development, an Angular application must be **built, optimized, and deployed** for real users.
Angular 17 provides a powerful **build system**, **environment management**, and **performance optimization techniques** to deliver fast and scalable applications.

1.5.1 Angular Build Process, Environments, and Optimization Flags

A. Angular Build Process

The **Angular build process** converts TypeScript and Angular code into **optimized JavaScript files** that browsers can execute.

Build Command

```
ng build
```

What Happens During Build

- TypeScript → JavaScript compilation
- Template compilation
- Bundling of files
- Minification
- Tree-shaking
- Optimization for browser

Production Build

```
ng build --configuration=production
```

B. Environment Configuration

Angular supports **multiple environments** to manage different settings.

Common environment files:

```
environment.ts        (development)

environment.prod.ts   (production)
```

Usage

```
import { environment } from '../environments/environment';

console.log(environment.apiUrl);
```

Purpose of Environments

- Separate API URLs
- Hide sensitive configuration
- Enable/disable debug features

C. Optimization Flags (Exam ⭐)

Flag	Purpose
optimization	Minifies and optimizes code
aot	Ahead-of-Time compilation
buildOptimizer	Removes Angular metadata
sourceMap	Debugging (disabled in prod)
vendorChunk	Separate vendor files

1.5.2 Deploying Angular Applications using Firebase Hosting

What is Firebase Hosting?

Firebase Hosting is a **fast, secure, and scalable hosting service** for web applications.

Why Firebase Hosting?

- Free SSL
- Fast global CDN
- Easy setup
- SPA routing support

Steps to Deploy Angular App

Step 1: Install Firebase CLI

bash Copy code

```
npm install -g firebase-tools
```

Step 2: Login to Firebase

bash Copy code

```
firebase login
```

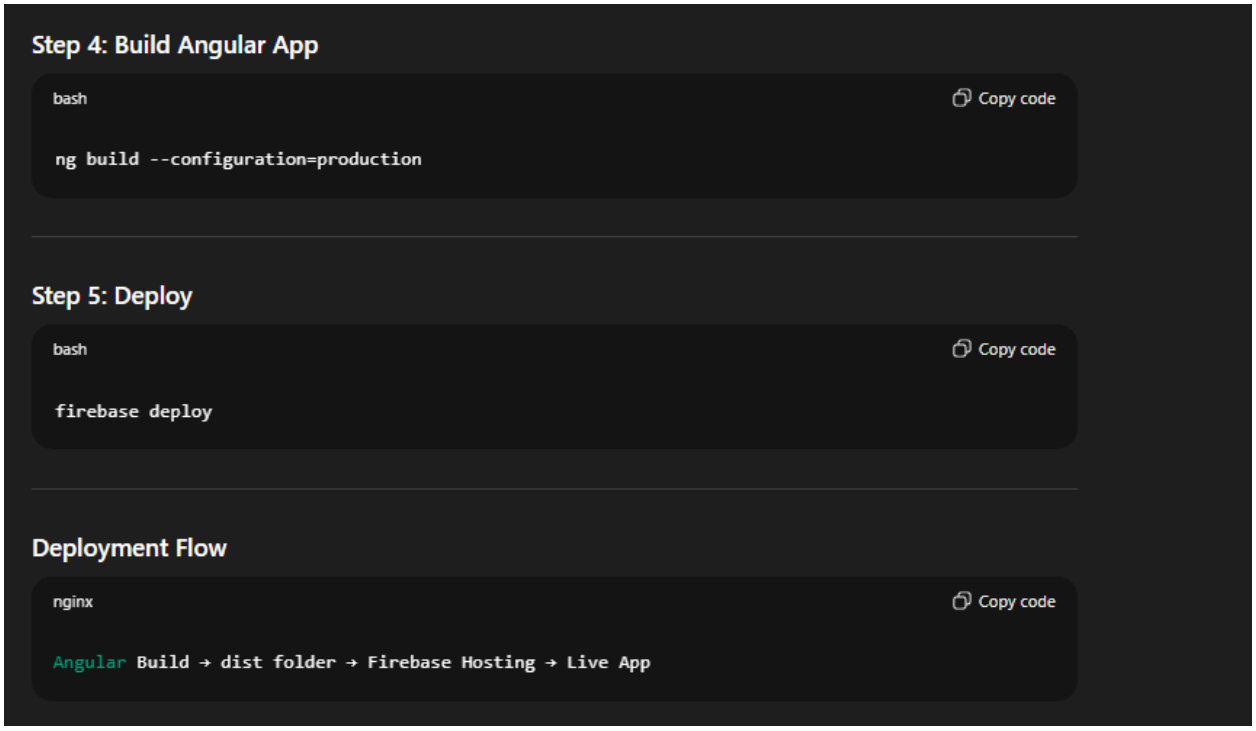
Step 3: Initialize Firebase

bash Copy code

```
firebase init
```

Select:

- Hosting
- Use `dist/` folder
- Configure as SPA → Yes



Benefits of Firebase Hosting

- Automatic HTTPS
- Fast load times
- Easy rollbacks
- Supports Angular routing

1.5.3 Performance Tuning Techniques

Performance optimization ensures **fast loading, smooth UI, and efficient resource usage**.

A. trackBy in ngFor

Problem

Angular re-renders the entire list when data changes.

Solution

Use `trackBy` to track items by unique ID.

```
<li *ngFor="let item of items; trackBy: trackById">
  {{ item.name }}
</li>


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

Benefits

- Fewer DOM updates
- Faster rendering
- Improved performance

B. OnPush Change Detection

Default Behavior

Angular checks all components on any change.

OnPush Strategy

Angular checks component **only when input reference changes**.

```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
})
```

C. Lazy Loading Routes and Components

Lazy Loading Routes

```
{
  path: 'admin',
  loadChildren: () =>
    import('./admin/admin.module')
      .then(m => m.AdminModule)
}
```

Lazy Loading Standalone Components

```
{
  path: 'dashboard',
  loadComponent: () =>
    import('./dashboard.component')
      .then(c => c.DashboardComponent)
}
```

Benefits of Lazy Loading

- Smaller initial bundle
- Faster startup
- Better scalability