Unit 1: Fundamentals of Angular (v17) for Single Page Applications

1.1 Angular Recap & Project Setup
- 1.1.1 Brief Recap of Angular 17 core concepts: Components, Services, Routing
- 1.1.2 Setting up a scalable Angular project using Angular CLI with Standalone Components
- 1.1.3 Folder structure and module organization for large projects

1.2 Advanced Routing & State Handling
- 1.2.1 Implementing Lazy Loading with Feature Modules
- 1.2.2 Route Guards: CanActivate, CanDeactivate for securing routes
- 1.2.3 Route Resolvers for preloading data
- 1.2.4 Introduction to advanced state handling using RxJS Subjects and Behavior Subjects

1.3 Reactive Forms in Real Applications
- 1.3.1 Dynamic form generation using FormArray
- 1.3.2 Custom Validators and Asynchronous Validation
- 1.3.3 Centralized error handling and displaying validation messages
- 1.3.4 Submitting forms to APIs and form state management

1.4 Buil+++++ding Reusable UI Components & Design Patterns
- 1.4.1 Creating reusable card, modal, and alert components
- 1.4.2 Component interaction with RxJS and Shared Services
- 1.4.3 Use of ng-template, ng-container, ng-content for structural flexibility
- 1.4.4 Smart vs Dumb Components: Best practices

1.5 Application Deployment & Performance Optimization
- 1.5.1 Angular build process, environments, and optimization flags
- 1.5.2 Deploying Angular applications using Firebase Hosting
- 1.5.3 Performance tuning: trackBy, OnPush change detection, lazy loading routes/components

1.1 Angular Recap & Project Setup

❖ **Angular Introduction**

- Angular is an **open-source JavaScript framewor**k.
- It is client-side JavaScript framework.
- It is built by **Google** for creating dynamic web applications.
- It can be freely used, changed and shared by anyone.
- It is a framework for building **single page applications.**

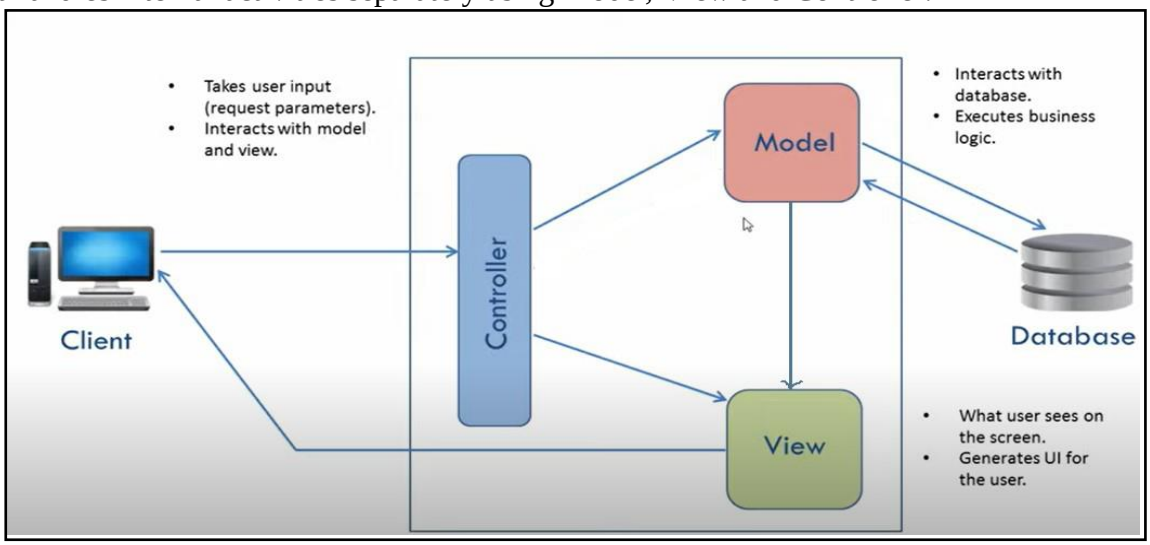
Aspect	Angular (Framework)	React (Library)
Developed By	Google	Facebook
Type	Full-fledged Framework. It comes with everything needed (routing, state management, form handling, HTTP requests, etc.).	UI Library. It focuses mainly on building **UI components**. For routing, state management, etc., React needs extra libraries.
Language	TypeScript (default)	JavaScript (ES6+), TypeScript (optional)
Architecture	MVC	Component-based
DOM	Real DOM + Change Detection	Virtual DOM
Data Binding	Two-way	One-way (unidirectional)
Flexibility	Less flexible (predefined structure)	Highly flexible (choose your own libraries)
Performance	Slower for very large apps	Faster updates with Virtual DOM
Best Suited For	Large-scale enterprise apps (ERP, CRM, dashboards)	Dynamic SPAs, interactive UIs
Examples	Gmail, Google Cloud, Upwork	Facebook, Instagram, Netflix, WhatsApp Web

❖ **Characteristics/Features of Angular/Advantages of Angular**

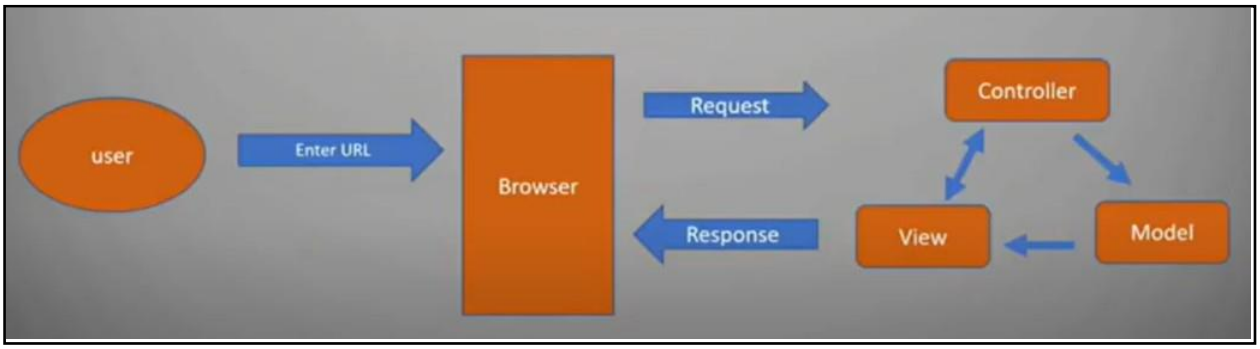
1. **Two-Way Data Binding**
   - Automatically synchronizes data between the **model** and the **view**.
   - If you update the model (JavaScript), the view (HTML) updates automatically.
   - If you change the input in the view (e.g., via a form), the model updates too.
2. **MVC Architecture**
   - AngularJS follows the MVC design pattern, which separates the application logic into three interconnected components:
   - **Model**: Manages the data (business logic).
   - **View**: Handles the UI (what the user sees).
   - **Controller**: Connects the model and view, processes input, and updates them.
3. **Directive**s
   - Directives are special markers (attributes or elements) in HTML that tell AngularJS to do something.
   - Built-in directives: ng-model, ng-repeat, ng-if, ng-show, etc.
   - It tells how to combine data into the HTML template.
   - With the use of directive, we can provide extra functionality to our angular application.
   - Angular provides a way to create custom directives too.
4. **Not Browser Specific**
   - Angular applications are not browser specific means there is no browser constraint on an angular application.
   - It can run on all major browsers except internet explorer 8.0 and smartphones including Android and IOS based phones/tablets.
5. **Code Less**
   - A programmer can write less and perform more functionality with the same code.
   - **Filters** in angular provide the functionality of write less do more.
   - The various filters in angular are uppercase, lowercase, currency etc.
   - You can use the filter and easily format the data.
6. **Speed And Performance**
   - Speed and performance of angular are faster because of three things:
   - When you are writing code using angular, it converts your template into a highly optimized code that gives you an advantage of handwritten code with the productivity of framework.
   - The first view of your application on .net, PHP, node.js and other servers that is till now dependent on HTML CSS for their front end which can serve using angular.
   - Its new component router loads angular app quickly. It provides the ability of automatic code splitting too. Therefore, only that code is loaded which is requested to render the view.
7. **Dependency Injection**
   - Dependency Injection specifies a design pattern in which components are given their dependencies instead of hard coding them within the component.
   - Whenever, angular JS detect that you need a **service** then it immediately provides an instance for that.
   - AngularJS has a built-in **dependency injection system**. It automatically provides required services or components (like $http, $scope, etc.) when needed.
8. **Routing**
   - Routing allows the **switching** between **views**.
   - Being a single page application ngRoute directive provided by angular, helps a user to navigate from one view to another, but the application will remain single page.
   - It means without reloading the angular application you can switch to different pages in your application.

❖ **Angular Architecture**

- MVC stands for Model View Controller.
- It is a software design pattern for developing web applications.
- It handles internal activities separately using Model, View and Controller.



1. Client sends request to the Controller for getting all students' detail which is available in database.
2. Controller passes the request to the Model and model communicate with database and get back response.
3. Model passes the row data to the View.
4. View does proper formatting to the data as per the instruction provided by Controller so it becomes visible.
5. Client can see the formatted data provided by the View.



The MVC pattern is made up of the following three parts:

1. It is responsible for **managing application data(Model)**. It responds to the requests from view and to the instructions from controller to update itself. (Using JSON Data object which is loaded inside view.). It can be represented as body of code which is used to represent the data.
2. It is responsible for **displaying all data or only a portion of data to the users(View)**. It also specifies the data in a particular format triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology. (Using HTML Tags.). It is a body of code which is used to represent UI.
3. It is responsible to **control the relation between models and views(Controller).** It responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.
4. 1.1.2 Setting up a scalable Angular project using Angular CLI with Standalone Components

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

❖ **Components**

- A **component** is a building block of an Angular app that controls a part of the user interface (UI). Each component includes:

- A **TypeScript file** (.ts) — logic
- An **HTML file** (.html) — template
- A **CSS file** (.css) — styles
- A **spec file** (.spec.ts) — for testing (optional)

Method 1: Creating a Component Using Angular CLI

ng generate component component-name

OR

ng g c component-name

Example: Create a "hello" component

ng g c hello

Auto-Generated Files:

src/app/hello/

```
├── hello.ts      → component logic
├── hello.html    → component template (UI)
├── hello.css     → component styles
└── hello.spec.ts → unit test file
```

Example:

hello.ts

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

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {
  message = 'Hello from Angular Component!';
}
```

hello.html

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

hello.css

```
h2 {
  color: blue;
}
```

app.html:

```
<h1>Welcome to My Angular App</h1>
<app-hello></app-hello>
```

Method 2: Creating a Standalone Component (Angular 14+)

- Angular 14 introduced **standalone components**, so you don't need to add them to an NgModule.
- A **standalone component** is a concept introduced in **Angular 14** (and improved in later versions) that allows you to create components that are **not part of any Angular module (NgModule)**.
- In other words — **a standalone component can exist and work on its own**, without needing to be declared inside an NgModule.

Traditional way (before standalone components)

- In older versions of Angular, every component had to be declared inside an **NgModule**, like this:
  Example: ng g c home

app.module.ts

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

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

- **Here, the HomeComponent must be declared inside the AppModule.**

With Standalone Components (Angular 14+)

- Now you can create a **standalone component** that doesn't need to be declared in any module.
  ng generate component home –standalone OR ng g c home --standalone

home.ts

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

@Component({
  selector: 'app-home',
  standalone: true,        //  Marks it as a standalone component
  imports: [CommonModule], // Import needed modules directly here
  template: `<h2>Welcome to the Home Component!</h2>`,
})
export class HomeComponent {}
```

- You can now **use it directly** in your application **without adding it to a module**:

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { HomeComponent } from './home.component';

bootstrapApplication(HomeComponent);
```

- **Key Benefits of Angular(14+)/Angular17**
  - No app.module.ts — everything is **standalone**.
  - No need to write **standalone keyword at a time creating component in Angular17.**
  - Each component imports its own dependencies.
  - No need for NgModules – simplifies project structure.
  - Faster setup – bootstrap directly with a component.
  - Easier code reuse – components are more self-contained.
  - Simplified lazy loading – can load standalone components easily via the router.

❖ **Services in Angular**

- **Definition:** A service is a class that contains business logic or data logic (not directly related to the view).
- **Purpose:**
  - Share data between components.
  - Interact with backend APIs.
  - Perform calculations or logic.

Creating Service in Angular using CLI:

Syntax: ng generate service service-name

Example:
ng generate service my-service

This will create:
src/app/my-service.ts
src/app/my-service.spec.ts

my-service.ts:

```
import { Injectable } from '@angular/core';
@Injectable({
providedIn: 'root' // Makes the service a singleton and available throughout the application
  })
export class MyService {
        constructor() { }
        getData(): string {
                return 'Data from MyService';
        }}
```

Call Service within component(app.ts):

```
import { Component } from '@angular/core';
import { MyService } from './my-service'; // Import the service
@Component({
selector: 'app-root',
templateUrl: './app.html',
styleUrls: ['./app.css']
  })
export class App {
message: string;
  constructor(private m1: MyService) { // Inject the service
          this.message = this.m1.getData(); // Use the service
    } }
```

app.html

```
<h1>Angular Service Example</h1>
{{message}}
```

❖ ROUTING

- Routing allows you to navigate between different views (components) using **URLs**.
- Routing enables navigation between different views (components) within a Single-Page Application (SPA) without full page reloads.
- **Routing** allows an Angular application to **navigate between different pages or views** without reloading the browser.
- Each route is linked to a **component**.
- In Angular 17, you can use **standalone components** directly with routes.
- Supports **lazy loading** to improve performance.

Example:

Step 1: Create Standalone Components

- For example, HomeComponent and AboutComponent.

```
ng new FWD –routing
ng g c home
ng g c about
```

HomeComponent(home.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-home',
  imports: [],
  templateUrl: './home.html',
  styleUrl: './home.css'
})
export class Home {}
```

home.html

```
<h1>Home Page</h1>
<hr>
```

AboutComponent(about.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-about',
  imports: [],
  templateUrl: './about.html',
  styleUrl: './about.css'
})
export class About {
}
```

about.html

```
<h1>About Us Page</h1>
<hr>
```

Step 2: Define Routes

src/app/app.routes.ts

```
import { Routes } from '@angular/router';
import { About } from './about/about';
import { Home } from './home/home';
export const routes: Routes = [
   {  path: '',component:Home },
   {  path:'about',component:About },
];
```

Step 3: Add RouterOutlet in Root Component
src/app/app.ts

```
import { Component} from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-root',
  imports: [CommonModule, RouterOutlet,RouterLink],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
   }
```

src/app/app.html

```
<h1>Angular 17 Routing Example</h1>
   <nav>
     <a routerLink="/">Home</a> |
     <a routerLink="/about">About</a>
   </nav>
   <hr />

   <router-outlet></router-outlet>
```

- <router-outlet> is a placeholder for the routed components.
- routerLink is used to navigate between routes.

Step 4: Bootstrap the Application with Router
src/main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { App } from './app/app';
import { provideRouter } from '@angular/router';
import { routes } from './app/app.routes';

bootstrapApplication(App, {
  providers: [provideRouter(routes)]
}).catch(err => console.error(err));
```

- No **NgModule** needed — Angular 17 uses bootstrapApplication() for standalone apps.

app.css

```
nav a {
  text-decoration: none;
  margin-right: 10px;
  color: blue;
}
nav a:hover {
  text-decoration: underline;
}
```

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

- In traditional Angular, every component had to be declared inside an NgModule. But with **standalone components**, introduced in Angular v14+, you can create components that work independently — no need to declare them in a module.
- Setting up a scalable Angular project using Angular CLI with Standalone Components involves leveraging the latest features for a more modular and efficient architecture.

Step:1. Install Angular CLI (if not already installed or update to the latest version):

```
npm install -g @angular/cli
```

Step:2. Create a new Angular project with Standalone Components enabled:

- When creating a new project, you can specify that the project should use **standalone** components from the start. But in angular 17 by default all the components must be standalone.

```
ng new my-standalone-app --standalone
```

- This command creates a new Angular project named **my-standalone-app** with the App component and main.ts file configured for standalone components.

Step:3. Generate Standalone Components:

- When generating new components, directives, or pipes, use the --standalone flag.

ng generate component **magic-component** --standalone

- This command creates **magic-component** as a standalone component, meaning it does not need to be declared within an **NgModule**.

Step:4. Import Dependencies in Standalone Components:

- Standalone components import their own dependencies directly within the **@Component decorator's imports array**.

magic-component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common'; // Example: importing a common module
@Component({
  selector: 'app-magic-component',
  standalone: true,
  imports: [CommonModule], // Import necessary modules or other standalone components
  templateUrl: './magic-component.html',
  styleUrls: ['./magic-component.css']
})
export class MagicComponent {
  // Component logic        }
```

Step:5. Bootstrap the Application with a Standalone Component:

- In a standalone application, you bootstrap directly with a standalone component, typically AppComponent, in **main.ts**.

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
bootstrapApplication(AppComponent)
  .catch(err => console.error(err));
```

Step:6. Implement Lazy Loading with Standalone Components:

- Lazy loading routes in a standalone application can be achieved using the **loadComponent** property in the router configuration.
- **Lazy Loading** means loading parts of your app (like a feature or route) **only when needed**—for example, when the user navigates to that route.
- It reduces the initial bundle size and improves performance.
- **Eager loading:** Loads everything at startup.
  Lazy loading: Loads only what's necessary initially, and loads other parts later on demand.

```
app.routes.ts (or your routing module)
import { Routes } from '@angular/router';
export const routes: Routes = [
  {
    path: 'admin',
    loadComponent: () => import('./admin/admin.component').then(m => m.Admin)
    // path: 'admin', component:Admin (No Lazy Loading)
  }];
```

- Using **loadComponent()** enables **lazy loading**.
- Components are loaded **only when the route is visited**. (You can see in inspect→debugger->src...)

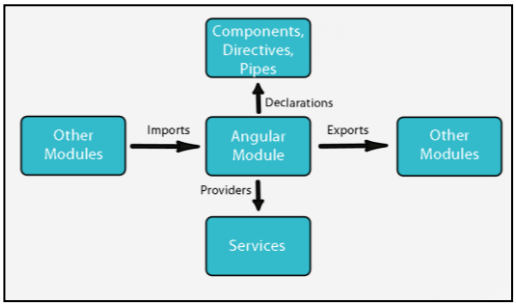
- **Eager Loading vs Lazy Loading**

- By default, Angular uses **eager loading**, where all modules are bundled and loaded upfront when the application starts. While this works well for small applications, it can cause performance issues as your app grows.
- In contrast, **lazy loading** loads feature modules dynamically as users navigate through the application. This keeps the initial bundle smaller and speeds up the bootstrapping process.

Loading Strategy	Description	Pros	Cons
Eager Loading	Loads all modules at startup	Simple to implement	Slower initial load time
Lazy Loading	Loads modules on demand	Faster startup, better performance	Requires careful routing setup

❖ **What is an Angular Module?(Extra Topic)**

- The module is the main Building Block of Angular which is the group of components, directives, pipes, and services that are the parts of the Angular application.
- We use ngModule metadata object for defining the module that instructs Angular how to compile and run the application.




```
import { NgModule } from '@angular/core';
 @NgModule({
  declarations: [here will be your Components],
  imports: [Import your module],
  providers: [Here will be your services],
  bootstrap: [Here will be your default component that will load]
})
export class AppModule { }
```

- **What is NgModule in Angular**
  - NgModule is a class that is created by the @NgModule decorator. It takes a metadata object that's described how to compile the component's template and create an injector at runtime. It defines the module's own components, directives, pipes, and services and makes some of them public, allowing external components to use them via the exports property.
  - **NgModule uses below metadata properties:**

1. **Declarations:** We call classes of components, directives, and pipes here. If you add the same class multiple times, it will give errors.
2. **Imports:** Here, we load additional modules such as FormsModule, RouterModule, CommonModule, or any other custom module.
3. **Providers:** We use injectable services here. It fetches data from the API.
4. **Exports:** A set of declarable components, directives, pipes, and modules that an importing module can use within any component template.
5. **Bootstrap**: Here we add the component that will be the root component or main application view, which is the host for all other application views.

- **How to create a custom module in Angular**
  - We can also create a custom module and organize your code in that module. Below are the CLI commands to create a module.
  - **The CLI command for generating a Module**

```
ng generate module module-name
```

  - **The CLI command for generating a Module with Routing**

```
ng generate module module-name --routing
```

  - **Sorter CLI command for generating a module.**

```
ng g m module-name
```

  - Angular 17 introduces a major shift: **standalone components**. These components can be used without being declared in a module. This simplifies the architecture and reduces boilerplate.
  - The only case I see where NgModules might be beneficial, is when you have a big feature consisting of several components/directives/pipes. Here a module can help to collect them, instead of needing to import all separately.

1.1.3 Folder structure and module organization for large projects

- For **large projects**, your structure should:
- Support **scalability** (easy to add new features)
- Enable **lazy loading**
- Maintain **separation of concerns**
- Support **team collaboration**
- Promote **code reusability** (shared utilities, components, etc.)
- **For large Angular projects, use a modular folder structure with Core, Shared, and Feature modules to ensure scalability, maintainability, and performance.**

```
src/
├── app/
│   ├── core/          # Singleton services, interceptors, guards
│   ├── shared/        # Reusable components, directives, pipes
│   ├── features/      # Domain-specific modules (e.g., user, admin)
│   ├── layouts/       # App-wide layout components (e.g., header, footer)
│   ├── app-routing.module.ts
│   └── app.module.ts
├── assets/            # Static files (images, fonts, etc.)
├── environments/      # Environment-specific configs
└── index.html
```

  - Here's a breakdown of the recommended folder structure and module organization for large-scale Angular applications:

Module Organization Strategy:

1. **Core Module**
   - Singleton services (e.g., AuthService, LoggingService)
   - Guards, interceptors
   - Should be imported only in AppModule

2. **Shared Module**
   - Reusable UI components (buttons, modals)
   - Pipes, directives
   - Can be imported in any feature module
3. **Feature Modules**
   - Organized by domain (e.g., UserModule, AdminModule)
   - Each has its own routing module
   - Supports **lazy loading** for performance
4. **Routing Module**
   - AppRoutingModule handles root-level routes
   - Feature modules have their own FeatureRoutingModule

– **Folder Structure example of large project like E-commerce Website:**

```
src/
├── app/
│   ├── core/                    # Singleton services (Auth, API, Intercepto
│   ├── shared/                  # Reusable UI components, pipes, directives
│   ├── features/
│   │   ├── auth/                # Login, Register, Forgot Password
│   │   ├── products/            # Product list, details, filters
│   │   ├── cart/                # Cart view, quantity updates
│   │   ├── checkout/            # Address, payment, order summary
│   │   ├── orders/              # Order history, order details
│   │   ├── user/                # Profile, settings
│   │   └── admin/               # Product management, user management
│   ├── layouts/                 # Header, footer, sidebar
│   ├── app-routing.module.ts
│   └── app.module.ts
│
├── assets/                      # Images, icons, mock data
├── environments/                # environment.ts and environment.prod.ts
└── index.html
```

1.2 Advanced Routing & State Handling

1.2.1 Implementing Lazy Loading with Feature Modules

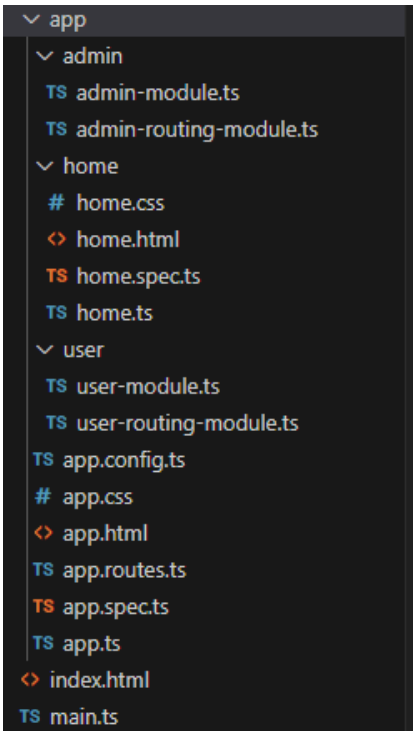
– A **Feature Module** in Angular is a way to **organize related components, directives, pipes, and services** into **self-contained units** of functionality.

– It helps you **divide a large app into smaller, manageable pieces** — each focused on a specific feature or domain (like Users, Products, Dashboard, etc.).

In Simple Terms

– A **Feature Module** is like a mini Angular app inside your main app — with its own components, routing, and logic, all grouped by feature.

– Angular 17 supports **hybrid architecture**:
   - You can use **standalone components** inside feature modules.
   - Feature modules can export standalone components.
   - Lazy loading works with both module-based and standalone setups.



Example with two feature modules (AdminModule, UserModule) with Lazy Loading:

Step 1: Create New App

- ng new **lazy-admin-user** –routing
  Where –routing means Adds a AppRoutingModule with basic setup for routing.
- cd lazy-admin-user

Step 2: Create Home Component

- ng g component home

home.component.ts

```
import { Component } from '@angular/core';
import { RouterLink } from '@angular/router';
@Component({
  selector: 'app-home',
  imports: [RouterLink],
  templateUrl: './home.html',
  styleUrl: './home.css',
})
export class Home {}
```

home.component.html

```
<h2>Welcome Home</h2>
   <nav>
     <a routerLink="/admin">Go to Admin</a> |
     <a routerLink="/user">Go to User</a>
   </nav>
```

Step 3: Create Lazy-Loaded Feature Modules

Create Admin Module

```
ng g module admin --routing
```

Create User Module

```
ng g module user --routing
```

Step 4: Check app-routes.ts

– After running the above commands, it should look like this:

```
import { Routes } from '@angular/router';
import { Home } from './home/home';
import { PageNotFound } from './page-not-found/page-not-found';
export const routes: Routes = [
  { path: '', component: Home },
  { path: 'admin',  loadChildren: () =>import('./admin/admin-module').then(m => m.AdminModule) },
  { path: 'user', loadChildren: () => import('./user/user-module').then(m => m.UserModule)  },
 { path:'**',component:PageNotFound  }  //WildCard Routing
];
```

Step 5: Add Components to Each Module

Admin Module

```
ng g component admin/admin-dashboard
```

User Module

```
ng g component user/user-profile
```

admin-dashboard.ts

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

@Component({
  selector: 'app-admin-dashboard',
  imports: [RouterLink],
  templateUrl: './admin-dashboard.html',
  styleUrl: './admin-dashboard.css',
})
export class AdminDashboard {}
```

admin-dashboard.html

```
    <h2>Admin Dashboard</h2>
    <p>Welcome, Admin! You can manage users and settings here.</p>
    <a routerLink="/">Back to Home</a>
```

admin-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AdminDashboard } from './admin-dashboard/admin-dashboard';

const routes: Routes = [
  {path:'',component:AdminDashboard}
];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AdminRoutingModule { }
```

admin.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { AdminRoutingModule } from './admin-routing-module';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    AdminRoutingModule
  ]
})
export class AdminModule { }
```

user-profile.ts

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

@Component({
  selector: 'app-user-profile',
  imports: [RouterLink],
  templateUrl: './user-profile.html',
  styleUrl: './user-profile.css',
})
export class UserProfile {}
```

user-profile.html

```
<h2> User Profile</h2>
    <p>Welcome, User! This is your profile page.</p>
    <a routerLink="/">Back to Home</a>
```

user.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { UserRoutingModule } from './user-routing-module';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    UserRoutingModule
  ]
})
export class UserModule { }
```

user-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { UserProfile } from './user-profile/user-profile';

const routes: Routes = [
  {path:'',component:UserProfile}
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class UserRoutingModule { }
```

app.html

```
<router-outlet>
</router-outlet>
```

Step 6: Run the App

- ng serve

Then visit:

- / → Home page (loaded immediately)
- /admin → Angular loads **AdminModule** on demand
- /user → Angular loads **UserModule** on demand

Step 7: Verify Lazy Loading Works

1. Open **DevTools(inspect) → Network → JS files OR check Debugger->src(folder)**
2. When you click on **Admin**, a new chunk loads:admin-admin-module.js
3. When you click on **User**, another chunk loads:user-user-module.js

1.2.2 Route Guards: CanActivate, CanDeactivate for securing routes

- **What are Route Guards?**

– In Angular, **route guards** are interfaces that allow you to control access to specific routes in your application.

– They act as gatekeepers, deciding whether a user can navigate to a route, leave a route, or even load a lazy-loaded module.

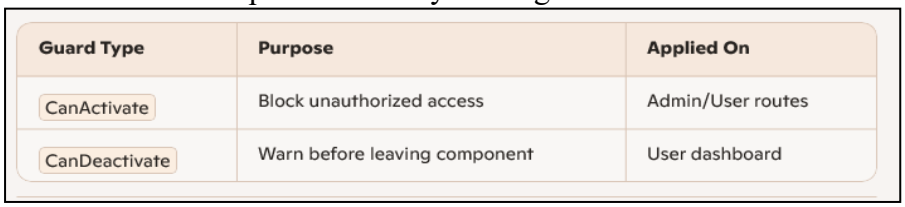
– Guards are especially useful when combined with **lazy loading**, ensuring that unauthorized users cannot even load certain parts of your application.

- **Types of Angular Route Guards**

– Angular provides several built-in guard interfaces:

Guard Interface	Description
CanActivate	Checks if the route can be activated (navigated to).
CanDeactivate	Checks if the user can leave the current route.
CanLoad	Prevents the loading of lazy-loaded modules.
CanActivateChild	Checks if child routes can be activated.
Resolve	Fetches data before the route is activated.

– we'll focus on the two most important for lazy loading:



Guard Type	Purpose	Applied On
CanActivate	Block unauthorized access	Admin/User routes
CanDeactivate	Warn before leaving component	User dashboard

Example:

Step 1: Create a New App

– ng new route-guard-examples --routing

– cd route-guard-examples

Step 2: Create Component with Lazy Loading

– ng generate component admin

– ng generate component user

Step 3: Create Guards

(a) AuthGuard → Protect routes using CanActivate

– ng generate guard guards/auth

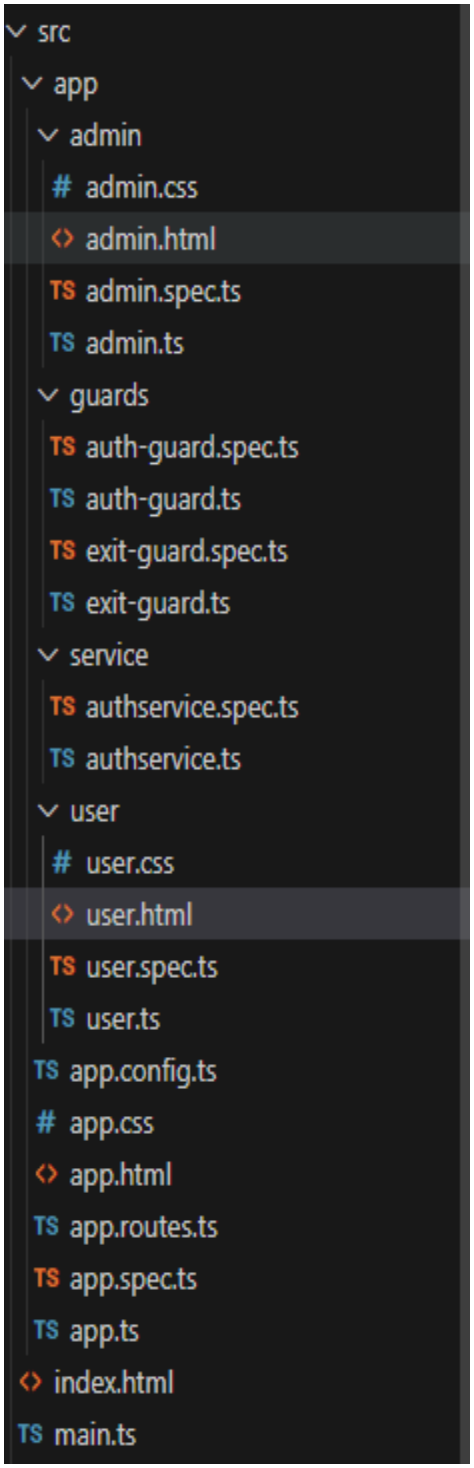
(b) ExitGuard → Confirm before leaving route (CanDeactivate)

– ng generate guard guards/exit

Step 4: Create Service AuthService for authentication

- ng g s service/authservice

Step 5: Final Folder Structure



admin.ts

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

@Component({
  selector: 'app-admin',
  imports: [],
  templateUrl: './admin.html',
  styleUrl: './admin.css',
})
export class Admin {}
```

admin.html

```
<h2>Admin Dashboard</h2>
<p>Only authorized users can view this page.</p>
```

user.ts

```
import { Component } from '@angular/core';
import { Authservice } from '../service/authservice';
import { inject } from '@angular/core';

@Component({
  selector: 'app-user',
  imports: [],
  templateUrl: './user.html',
  styleUrl: './user.css',
})
export class User {
private auth = inject(Authservice);

  login() {
    this.auth.login();
    alert('Logged in successfully!');
  }

  logout() {
    this.auth.logout();
    alert(' Logged out!');
  }
}
```

user.html

```
<h2>User Page</h2>
<p>Welcome to the User page.</p>

<button (click)="login()">Login</button>
<button (click)="logout()">Logout</button>
```

authservice.ts

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

@Injectable({
  providedIn: 'root',
})
export class Authservice {
  private loggedIn = false; // default state

  isAuthenticated(): boolean {
    return this.loggedIn;
  }

  login(): void {
    this.loggedIn = true;
  }

  logout(): void {
    this.loggedIn = false;
  }
}
```

auth-guard.ts – CanActivate

```
import { inject } from '@angular/core';
import { Router, CanActivateFn } from '@angular/router';
import { Authservice } from '../service/authservice';

export const authGuard: CanActivateFn = (route, state) => {
   const router = inject(Router);
  const authService = inject(Authservice);

  const isLoggedIn = authService.isAuthenticated(); // ✅ Dynamic check

  if (!isLoggedIn) {
    alert('Access denied! Redirecting to User page...');
    router.navigateByUrl('/user');
    return false;
  }
  return true;
};
```

exit-guard.ts – CanDeactivate

```
import { CanDeactivateFn } from '@angular/router';
import { User } from '../user/user';

export const exitGuard: CanDeactivateFn<User> = () => {
  return confirm('Are you sure you want to leave this page?');
};
```

app.routes.ts

```
import { Routes } from '@angular/router';
import { authGuard } from './guards/auth-guard';
import { exitGuard } from './guards/exit-guard';
import { User } from './user/user';

export const routes: Routes = [
  {
    path: 'admin',
    loadComponent: () =>
    import('./admin/admin').then(m => m.Admin),
    canActivate: [authGuard],
  },
  {
    path: 'user',
    canDeactivate: [exitGuard],
    loadComponent: () =>
      import('./user/user').then(m => m.User),
  },
  { path: '', component:User },
  { path: '**', redirectTo: 'user' },
];
```

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { App } from './app/app';
import { provideRouter } from '@angular/router';
import { routes } from './app/app.routes';

bootstrapApplication(App,{
  providers: [provideRouter(routes)],
});
```

app.ts

```
import { Component, signal } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet,RouterLink],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('route-guard-examples');
}
```

app.html

```
<h1>Angular 17 Route Guards Demo</h1>
<nav>
  <a routerLink="/admin">Admin</a> |
  <a routerLink="/user">User</a>
</nav>
<router-outlet></router-outlet>
```

Step 6: Run and Test

- ng serve
- Open http://localhost:4200/user
- Try clicking on **Admin** in your nav bar (or manually go to /admin):
  - If **not logged in**, it alerts and redirects to /user.
- Click **Login** → then go to /admin again.
  - This time, you'll be allowed in because AuthService.loggedIn = true.

1.2.3 Route Resolvers for preloading data

- Angular Route Resolvers provide a mechanism to pre-fetch data before a component associated with a route is rendered, improving user experience by ensuring data is available upon navigation.
- In Angular 17, the modern approach utilizes ResolveFn for creating resolvers.

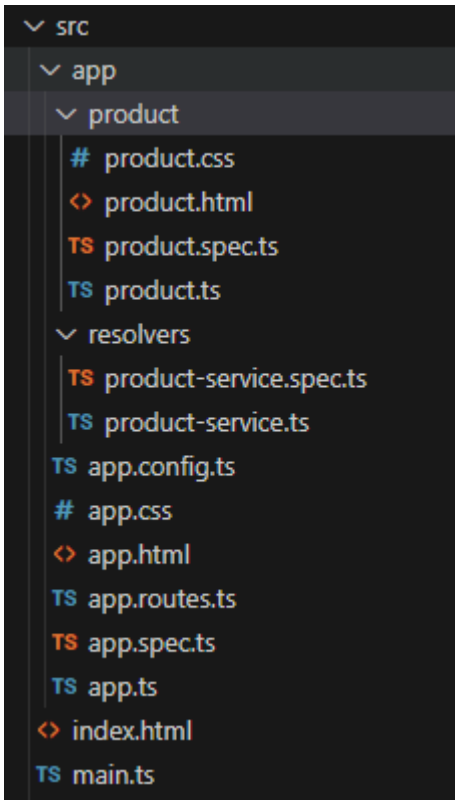
Example:

When navigating to /product, the **resolver** loads product data (before route activation).
Then the component shows a **button** — clicking it displays the resolved data.

- **ng new ResolverExample**

Step 1: Create the Resolver Service

- ng generate service resolvers/ProductService



product-service.ts

```
import { Injectable } from '@angular/core';
import { Resolve } from '@angular/router';
import { of } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class ProductService implements Resolve<any> {
  resolve() {
    console.log('UserResolver called!');
    return of({
       id: 101,
       pname: 'Laptop',
       price: 95000
    });  }}
```

Step 2: Configure the Route

app.routes.ts

```
import { Routes } from '@angular/router';
import { ProductService } from './resolvers/product-service';

export const routes: Routes = [
  {
    path: 'product',
    loadComponent: () => import('./product/product').then((m) => m.Product),
    resolve: { product: ProductService }, // attach resolver
  },
  { path: '', redirectTo: 'product', pathMatch: 'full' },
];
```

Step 3: Create the Component

- **ng g c product**

product.ts

```
import { CommonModule } from '@angular/common';
import { Component, inject } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-product',
  imports: [CommonModule],
  templateUrl: './product.html',
  styleUrl: './product.css',
})
export class Product {
private route = inject(ActivatedRoute);
  product = this.route.snapshot.data['product']; // resolved data
  display = false;

  showProduct() {
    this.display = true;
    console.log('Product Data:', this.product);
  }
}
```

product.html

```
    <h2>Product Page</h2>
    <button (click)="showProduct()">Show Product Details</button>

    <div *ngIf="display">
      <p><strong>ID:</strong> {{ product.id }}</p>
      <p><strong>Name:</strong> {{ product.name }}</p>
      <p><strong>Price:</strong> ₹{{ product.price }}</p>
    </div>
```

app.ts

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('ResolverExample');
}
```

App.html

```
<router-outlet>
</router-outlet>
```

Benefits of using Resolvers:

- **Improved User Experience:**
  Data is loaded before the component renders, preventing blank screens or loading spinners within the component.
- **Simplified Component Logic:**
  Components focus on rendering and interacting with data, offloading data fetching concerns to resolvers.
- **Better Error Handling:**
  Resolvers can handle errors during data fetching, preventing navigation to a route if necessary data is unavailable.

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

- **What is State?**

- **State** is simply the **current data or condition** of your application at a given time.
- Think of your app as a living system — its state defines what it looks like and how it behaves right now.

Example:

In a social media app:

- The current logged-in user → user state
- The list of posts → posts state

So, **state** is everything your UI needs to know to render itself correctly.

- **Why Managing State Matters**

- As your app grows, state becomes **harder to manage**:
  - Multiple components need the same data.
  - One component changes data that affects others.
  - API calls and UI interactions happen asynchronously.
- This leads to problems like:
  - Data inconsistency.
  - Unnecessary re-renders.
  - Complex event chains.
- So, we need **structured, reactive state management**.

❖ **Handling State with RxJS(Subject & BehaviorSubject)**

- **What is RxJS?**

- **RxJS (Reactive Extensions for JavaScript)** is a library that allows you to handle **asynchronous data streams**.

- Observable → A stream of data over time.
- **Subject / BehaviorSubject** → Special observables you can emit values into.
- Operators → Functions like map, filter, combineLatest, etc., to transform streams.

- **RxJS Subjects and BehaviorSubjects** are powerful constructs for managing reactive data and state in Angular applications. They allow components and services to communicate efficiently through observable streams.

- **What is a Subject?**
- A Subject is a special type of Observable that can broadcast values to multiple subscribers (multicasting) and also acts as an Observer (meaning you can push values into it using the next() method).
- Unlike regular Observables, Subjects are both an Observable and an Observer, allowing them to emit new values after initial subscription.
- **Key Characteristic**: A regular Subject does not have an initial value and does not store the last emitted value. Subscribers will only receive values that are emitted after they have subscribed.

Example

Step:1. Create the Angular Service: ng generate service counterService

counter-service.ts

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class CounterService {
    private countSubject = new Subject<number>();

    count$ = this.countSubject.asObservable();
    private currentCount: number = 0;

     increment() {
      this.currentCount++;
      this.countSubject.next(this.currentCount); // Emit the new count
     }

     reset() {
       this.currentCount = 0;
       this.countSubject.next(this.currentCount); // Emit the reset count
     }}
```

Step:2. Create the Component: ng generate component counter

counter.ts

```
import { Component } from '@angular/core';
import { CounterService } from '../counter-service';

@Component({
  selector: 'app-counter',
  imports: [],
  templateUrl: './counter.html',
```

```
    styleUrl: './counter.css',
  })
  export class Counter {
  count: number = 0;

    constructor(private cnt: CounterService) {}

    ngOnInit() {
      // Subscribe to the Subject to get the latest counter value (after any update)
      this.cnt.count$.subscribe((newCount) => {
        this.count = newCount;
      });
    }

    increment() {
      this.cnt.increment(); // Increment the counter
    }

    reset() {
      this.cnt.reset(); // Reset the counter
    }}
```

app.html

```
<h1>Example of RxJS Subject</h1>
<app-counter></app-counter>
```

app.ts

```
import { Component, signal } from '@angular/core';
import { Counter } from './counter/counter';
@Component({
  selector: 'app-root',
  imports: [Counter],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('SubjectExample');
}
```

counter.html

```
<div>
  <h1>Counter: {{ count }}</h1>
  <button (click)="increment()">Increment</button>
  <button (click)="reset()">Reset</button>
</div>
```

- **What is a BehaviorSubject?**

- A **BehaviorSubject** is similar to a Subject but requires an initial value and always emits the current value to new subscribers.
- **Key Characteristics**:
  - **Requires an Initial Value**: You must provide a starting value when creating a BehaviorSubject.
  - **Stores the Latest Value**: It always holds the current (last emitted) value.
  - **Instant Emission to New Subscribers**: When a new component subscribes, it immediately receives the latest value held by the BehaviorSubject, in addition to any subsequent changes.

Example: (Follow same step as subject example only change in counter-service.ts file.

counter-service.ts

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class CounterService {
  // BehaviorSubject to store the current counter value, starting at 0
  private countSubject = new BehaviorSubject<number>(0);

  // Observable to expose the current counter value
  count$ = this.countSubject.asObservable();

  // Method to increment the counter
  increment() {
    this.countSubject.next(this.countSubject.value + 1);
  }

  // Method to reset the counter
  reset() {
    this.countSubject.next(0);
  }}
}
```

1.3 Reactive Forms in Real Applications

1.3 Reactive Forms:

- Reactive forms in Angular are used to manage the state of form inputs in a reactive way.
- Reactive Forms (“model-driven” forms) give you a programmatic way to build, manage and validate forms.
- Some of their advantages:
  - The form model (via FormGroup, FormControl, FormArray) lives in the component class, so you have full control.
  - They enable synchronous access to data, changes, and statuses (valid/invalid) via observables.
  - Easier to test compared with template-driven forms.
  - More scalable for complex forms: nested groups, dynamic fields, conditional logic.
- In many real apps (e-commerce, banking, healthcare, CRM), Reactive Forms are the go-to because of these strengths.

1. **FormGroup**: Represents a group of form controls. It aggregates the values of each control into a single object.
2. **FormControl**: Represents a single input field — a form control instance manages the value, validation status, and user interactions of an input field.
3. **FormBuilder**: A service that provides convenient methods for creating instances of FormGroup and FormControl.
4. **Validators**: Functions used for synchronous and asynchronous validation of form controls.

Example: ng g c ReactiveEx

reactive-ex.html

```
<h2>Simple Reactive Form Example</h2>
   <form [formGroup]="userForm" (ngSubmit)="onSubmit()">
     <div>
       <label>Name:</label>
       <input formControlName="name" placeholder="Enter your name"><br><br>
       <label>Email:</label>
       <input formControlName="email" placeholder="Enter your email"><br><br>
       <label>Hobby:</label>
       <input formControlName="hobby" placeholder="Enter your hobby"><br><br>
       <button type="submit">Submit</button>
     </div>
   </form>
   <hr>
   <div *ngIf="data">
     <h3>Submitted Data:</h3>
     <p><strong>Name:</strong> {{ data.name }}</p>
     <p><strong>Email:</strong> {{ data.email }}</p>
     <p><strong>Hobby:</strong> {{ data.hobby }}</p>
   </div>
```

reactive-ex.ts

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

@Component({
  selector: 'app-reactive-ex',
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './reactive-ex.html',
  styleUrl: './reactive-ex.css',
})
export class ReactiveEx {
userForm = new FormGroup({
    name: new FormControl(),
    email: new FormControl(),
    hobby: new FormControl('')
});
data: any = null;
onSubmit() {
    this.data = this.userForm.value;  }}
```

app.html

```
<app-reactive-ex>/app-reactive-ex>
```

app.ts

```
import { Component, signal } from '@angular/core';
import { ReactiveEx } from './reactive-ex/reactive-ex';

@Component({
  selector: 'app-root',
  imports: [ReactiveEx],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('ReactiveFormEx');
}
```

1.3.1 Dynamic form generation using FormArray

What is a FormArray?

- A **FormArray** is a collection of FormControl, FormGroup, or even other FormArray instances. It's used when the number of form elements isn't fixed — for example:
  - Multiple phone numbers or email addresses
  - A list of skills or tags
  - A list of order items (in e-commerce)
  - A dynamic questionnaire (add/remove questions)

Example: ng g c FormArrayEx
form-array-ex.ts

```
import { Component } from '@angular/core';
import { ReactiveFormsModule, FormBuilder, FormGroup, FormArray, FormControl } from
          '@angular/forms';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-form-array-ex',
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './form-array-ex.html',
  styleUrl: './form-array-ex.css',
})
export class FormArrayEx {
 orderForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.orderForm = this.fb.group({
      items: this.fb.array([
        this.createItem() // start with one item
      ])    });   }
```

```
// Create a new order item FormGroup
createItem(): FormGroup {
  return this.fb.group({
    productName: new FormControl(''),
  });
}
// Getter for easy access to FormArray
get items(): FormArray {
  return this.orderForm.get('items') as FormArray;
}
// Add new item
addItem() {
  this.items.push(this.createItem());
}
// Remove item by index
removeItem(index: number) {
  this.items.removeAt(index);
}
// Submit form
onSubmit() {
  console.log(this.orderForm.value);
  alert(JSON.stringify(this.orderForm.value, null, 2));
}}
```

form-array-ex.html

```
<h2>Order Items Form</h2>

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

    <div formArrayName="items">
      <div *ngFor="let item of items.controls; let i = index" [formGroupName]="i" style="margin-bottom:
          10px; border-bottom: 1px solid #ccc; padding-bottom: 10px;">

        <label>Product Name:</label>
        <input type="text" formControlName="productName" placeholder="Enter product name">
        <button type="button" (click)="addItem()">+ Add Item</button>
        <button type="button" (click)="removeItem(i)">Remove Item</button>
      </div>
    </div>

      <br>
    <button type="submit">Submit Order</button>
  </form>

  <hr>
  <h3>Order Data:</h3>
  <pre>{{ orderForm.value | json }}</pre>
```

app.ts

```
import { Component, signal } from '@angular/core';
import { ReactiveEx } from './reactive-ex/reactive-ex';
import { FormArrayEx } from './form-array-ex/form-array-ex';
@Component({
  selector: 'app-root',
  imports: [ReactiveEx, FormArrayEx],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('ReactiveFormEx');
}
```

app.html

```
<app-form-array-ex></app-form-array-ex>
```

1.3.2 Custom Validators and Asynchronous Validation

- **What is Form Validation in Angular**
  - Form validation ensures that **user input meets certain criteria** before submitting the form.
    - **Reactive Forms** are **model-driven**, meaning the form logic is defined in the component class.
    - Validation rules are attached to **FormControl** instances in the form model.
  - In Angular, validation functions are categorized into two types based on their execution timing and return values: **synchronous** and **asynchronous**.

❖ **Synchronous Validation**

- Synchronous validators are functions that run immediately when the form value changes. They perform simple, client-side checks that do not require waiting for external operations.
- They return validation errors as an object (`ValidationErrors` or `{ [key: string]: any }`) if invalid, or `null` if valid, immediately.
- **Common built-in examples** include `Validators.required` , `Validators.minLength` , `Validators.maxLength` , `Validators.email` ,
- `Validators.min` , and `Validators.max` .
- In reactive forms, they are passed as the second argument when instantiating a `FormControl`.

Example:

app.html

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

  <label>Username:</label>
  <input type="text" formControlName="username" />

  <div *ngIf="form.controls.username.errors?.['required']">
    Username is required.
  </div>
  <div *ngIf="form.controls.username.errors?.['minlength']">
    Must be at least 4 characters.
  </div>
```

```
  <div *ngIf="form.controls.username.errors?.['pattern']">
    Only letters, numbers, and underscores allowed.
  </div>

  <button [disabled]="form.invalid">Submit</button>

</form>

<div *ngIf="sdata" style="margin-top: 20px;">
  <h3>Submitted Data:</h3>
  <pre>{{ sdata | json }}</pre>
</div>
```

app.ts

```
import { CommonModule } from '@angular/common';
import { inject, Component } from '@angular/core';
import { FormBuilder, Validators, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ReactiveFormsModule,CommonModule],
  templateUrl: './app.html'
})

export class App {

  fb = inject(FormBuilder);
 sdata :any=null;
  form = this.fb.group({
    username: ['', [
      Validators.required,
      Validators.minLength(4),
      Validators.pattern(/^[a-zA-Z0-9_]+$/)
    ]]
  });

  submitForm() {

    console.log(this.form.value);

    if (this.form.valid) {
      this.sdata = this.form.value;
    }

  }
}
```

- **Asynchronous Validation**

- Asynchronous validators perform more complex validations that involve operations that might take time, such as making an HTTP request to a server or interacting with a promise-based library. They run only after all synchronous validators have passed.
- They return a `Promise` or an `Observable` that later emits a set of validation errors or `null`. The observable must be finite, meaning it must complete.
- The most common use case is checking for username or email availability against a backend database.
- In reactive forms, they are passed as the third argument when instantiating a `FormControl` (or within the `asyncValidators` property of the options object). While an async validation is in progress, the form control enters a `PENDING` state, which can be used to show visual feedback like a loading spinner in the UI.

Example:

app.html

```
<form [formGroup]="form" (ngSubmit)="submitForm()">
    <label>Username:</label>
    <input type="text" formControlName="username" />

    <div *ngIf="form.controls.username.pending">Checking availability...</div>
    <div *ngIf="form.controls.username.errors?.['usernameTaken']">
      Username is already taken!
    </div>
    <button type="submit" [disabled]="form.pending">Submit</button>
  </form>
  <div *ngIf="submittedData">
    <h3>Submitted Data:</h3>
    <pre>{{ submittedData | json }}</pre>
  </div>
```

app.ts

```
import { Component, inject } from '@angular/core';
import { FormBuilder, FormControl, Validators, ReactiveFormsModule, AsyncValidatorFn,
AbstractControl, ValidationErrors } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { Observable, timer, of } from 'rxjs';
import { map } from 'rxjs/operators';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule],
  templateUrl: './app.html',
})
export class App {
  fb = inject(FormBuilder);
  submittedData: any = null;
```

```
 // Async validator: rejects "admin" as username
 usernameAsyncValidator: AsyncValidatorFn = (control: AbstractControl): Observable<ValidationErrors |
null> => {
   const value = control.value;
   if (!value) return of(null);

   return timer(500).pipe(
     map(() => (value === 'admin' ? { usernameTaken: true } : null))
   );  };
 // Form setup
 form = this.fb.group({
   username: ['', {
     validators: [Validators.required],
     asyncValidators: [this.usernameAsyncValidator],
     updateOn: 'blur'
   }]
 });
 submitForm() {
   if (this.form.invalid) return;
   this.submittedData = this.form.value;
 }}
```

Key Differences:

Feature	**Synchronous Validation**	**Asynchronous Validation**
Execution Time	Immediate	After all sync validators pass, requires waiting
Return Type	`ValidationErrors` or `null`	`Promise<ValidationErrors
Use Case	Client-side checks (e.g., required field, format)	Server-side checks (e.g., unique email lookup)
Form Status	`VALID` or `INVALID`	Can be `PENDING` during execution
`FormControl` Arg	Second argument	Third argument

❖ **Custom Validator**

- Creating a custom validator in Angular involves writing a function that takes a form control instance as an argument and returns a map of validation errors (or `null` if the control is valid).
- The process differs slightly for reactive and template-driven forms, but the core function is the same.

❖ **Custom Synchronous Validator (Reactive Forms)**

- A **synchronous validator** runs instantly on the same call stack (no async operations). It returns either:
  - **null** → meaning the value is valid
  - `ValidationErrors` **an error object** → meaning the value is invalid

Example:

Step: 1. Create the validator function: (Right click on app folder-> new file->give name (numbers.validator.ts)

numbers.validator.ts

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';
export function noNumbersValidator(): ValidatorFn {

  return (control: AbstractControl): ValidationErrors | null => {

    const value = control.value ?? '';
    // If the input contains a number
    const hasNumber = /\d/.test(value);
    return hasNumber ? { noNumbers: true } : null;
  };}
```

- The function takes an `AbstractControl` as a parameter. `AbstractControl` is the base class for `FormControl`, `FormGroup`, and `FormArray`.
- It returns `null` if the validation passes, and a `ValidationErrors` object (e.g., `{ 'forbiddenName': true }`) if it fails.
- To pass parameters, as in this example (the `nameRe` regex), you wrap the validator logic in an outer factory function that accepts the parameters and returns the actual `ValidatorFn`.

Step: 2. Add the validator to a `FormControl`:

First create component: ng g c username

- In your component or form service, add the validator function to the `FormControl`'s validator array.

username.ts

```
import { Component } from '@angular/core';
    import { FormControl, FormGroup, Validators, ReactiveFormsModule } from '@angular/forms';
    import { noNumbersValidator } from '../numbers.validator';
    import { CommonModule } from '@angular/common';
@Component({
      selector: 'app-username',
      standalone: true,
      imports: [ReactiveFormsModule, CommonModule],
      templateUrl: './username.html',
    })
```

```
export class Username{
  form = new FormGroup({
    username: new FormControl('', [
      Validators.required,
      noNumbersValidator()
    ])
  });
  get username() {
    return this.form.get('username') as FormControl;
  }   }
```

username.html

```
<h2>Create Username</h2>

  <form [formGroup]="form">
    <label>Username:</label>
    <input type="text" formControlName="username" />

    <div class="error" *ngIf="username.errors?.['noNumbers']">
      Username cannot contain numbers
    </div>

    <div class="error" *ngIf="username.errors?.['required']">
      Username is required
    </div>

    <p>Current Value: {{ username.value }}</p>
  </form>
```

app.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [Username],
  templateUrl: './app.html',
})
export class App{}
```

app.html

```
<div style="padding: 20px;">
    <app-username ></app-username>
  </div>
```

❖ **Custom Asynchronous Validator (Reactive Forms)**

- Asynchronous validators are used for validation that requires time, such as an HTTP request to a server (e.g., checking for a unique username).
- An async validator performs validation **after an asynchronous operation**, usually returning:
- **null** → value is valid
- **ValidationErrors** → value is invalid
- An async validator must return an **Observable** or **Promise**.

Example:

Step: 1. Create the async validator function/service:: (Right click on app folder-> new file->give name (numbers.async-validator)

- An async validator function must return a `Promise` or an `Observable` that eventually emits `null` or a `ValidationErrors` object.

numbers.async-validator

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, Validators, ReactiveFormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { noNumbersAsyncValidator } from '../numbers.async-validator';

@Component({
  selector: 'app-username',
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule],
  templateUrl: './username.html',
})
export class Username {

  form = new FormGroup({
    username: new FormControl(                                                    ",
      {
        validators: [Validators.required],
        asyncValidators: [noNumbersAsyncValidator()],
        updateOn: 'blur' // optional, triggers async validator on blur
      }
    )
  });
  get username() {
    return this.form.get('username') as FormControl;
  }
}
```

Step: 2. Add the async validator to a `FormControl`:

First create component: ng g c username

- In your component or form service, add the validator function to the `FormControl`'s validator array.

username.ts

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

```
import { noNumbersAsyncValidator } from '../numbers.async-validator';

@Component({
  selector: 'app-username',
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule],
  templateUrl: './username.html',
})
export class Username {

  form = new FormGroup({
    username: new FormControl('',
      {
        validators: [Validators.required],
        asyncValidators: [noNumbersAsyncValidator()],
        updateOn: 'blur' // optional, triggers async validator on blur
      }
    ) });

  get username() {
    return this.form.get('username') as FormControl;
  }
}
```

username.html

```
<h2>Create Username</h2>

<form [formGroup]="form">
  <label>Username:</label>
  <input type="text" formControlName="username" /> <!-- Async error -->

  <div class="error" *ngIf="username.errors?.['noNumbersAsync']">
    Username cannot contain numbers (async)
  </div>

  <div class="error" *ngIf="username.errors?.['required']"> <!-- Required error -->
    Username is required
  </div>

  <div *ngIf="username.pending">Checking...</div> <!-- Show loading state -->

  <p>Current Value: {{ username.value }}</p>
</form>
```

app.ts

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

@Component({
  selector: 'app-root',
```

```
  standalone: true,
  imports: [Username],
  templateUrl: './app.html',
})
export class App{}
```

app.html

```
<div style="padding: 20px;">
    <app-username ></app-username>
   </div>
```

1.3.3 Centralized error handling and displaying validation messages

- **Centralized error handling and displaying validation messages** in **Angular Reactive Forms** is crucial when you have many form fields and don't want to write repetitive *ngIf blocks for each field.

- **What is Centralized Error Handling**
  - Instead of showing validation messages individually in the template like this:
    <div *ngIf="form.get('username')?.errors?.required">Username is required
    <div *ngIf="form.get('username')?.errors?.minlength">Username is too short
  - We **centralize the error logic** in the **component** so the template just asks:
    "Give me the error message for this field if it exists."
  - **Benefits:**
    - Cleaner templates
    - Reusable across multiple forms
    - Easy to update error messages in one place

Example: (ng g c register)

register.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, ReactiveFormsModule, AbstractControl, ValidationErrors }
        from '@angular/forms';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-register',
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule],
  templateUrl: './register.html'
})
export class Register {

  form = new FormGroup({
    username: new FormControl('', {
      validators: [
        Validators.required,
        Validators.minLength(3),
        this.noSpacesAllowedValidator
      ]
```

```
  }),
  email: new FormControl('', {
    validators: [
      Validators.required,
      Validators.email
    ]
  }),
  password: new FormControl('', {
    validators: [
      Validators.required,
      Validators.minLength(6)
    ]
  })
});

// Error Map
errorMessages: { [key: string]: { [key: string]: string } } = {
  username: {
    required: 'Username is required',
    minlength: 'Username must be at least 3 characters',
    noSpacesAllowed: 'No spaces allowed'
  },
  email: {
    required: 'Email is required',
    email: 'Invalid email format'
  },
  password: {
    required: 'Password is required',
    minlength: 'Password must be at least 6 characters'
  }
};

noSpacesAllowedValidator(control: AbstractControl): ValidationErrors | null {
  const value = control.value || '';
  const hasSpace = /\s/.test(value);
  return hasSpace ? { noSpacesAllowed: true } : null;
}

// Error helper
getError(controlName: string): string | null {
  const control = this.form.get(controlName);

  if (control && control.touched && control.errors) {
    for (const errorKey in control.errors) {
      if (this.errorMessages[controlName][errorKey]) {
        return this.errorMessages[controlName][errorKey];
      }     }    }
  return null;  }}
```

register.html

```
<h2>Registration Form</h2>

<form [formGroup]="form">
  <label>Username:</label>
  <input type="text" formControlName="username" />
  <div class="error" *ngIf="getError('username')">
    {{ getError('username') }}
  </div>
  <br />
  <label>Email:</label>
  <input type="email" formControlName="email" />
  <div class="error" *ngIf="getError('email')">
    {{ getError('email') }}
  </div>
  <br />
  <label>Password:</label>
  <input type="password" formControlName="password" />
  <div class="error" *ngIf="getError('password')">
    {{ getError('password') }}
  </div>
  <br />
  <button [disabled]="form.invalid">Submit</button>
</form>
<p>Form Value: {{ form.value | json }}</p>
```

app.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [Register],
  templateUrl: './app.html',
})
export class App {}
```

app.html

```
<app-register></app-register>
```

1.3.4 Submitting forms to APIs and form state management

- **Submitting forms to APIs** and **form state management** in **Angular 17 Reactive Forms**. This is essential for real-world applications where forms interact with backend services.

Step: 1 Submitting Forms to APIs

- In Angular, form submission usually happens via **Reactive Forms** using HttpClient.

Steps:

1. **Create a reactive form** using FormBuilder

2. **Bind form controls to template** with [formGroup] and formControlName
3. **Listen to form submission** with (ngSubmit)
4. **Send data to API** using HttpClient.post

Example:

app.html

```
<app-register></app-register>
```

app.ts

```
import { Component } from '@angular/core';
import { Register } from './register/register';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [Register],
  templateUrl: './app.html',
})
export class App {}
```

register.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, ReactiveFormsModule } from '@angular/forms';
import { HttpClient } from '@angular/common/http';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-register',
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule],
  templateUrl: './register.html'
})
export class Register {
  constructor(private http: HttpClient) {}

  form = new FormGroup({
    name: new FormControl('', { validators: [Validators.required] }),
    email: new FormControl('', { validators: [Validators.required, Validators.email] })
  });
  message = '';
  submittedData: any = null; // Store submitted data

  onSubmit() {
    if (this.form.invalid) {
      this.message = 'Please fill all required fields.';
      return;
    }
    const apiUrl = 'https://jsonplaceholder.typicode.com/users'; // fake API

    this.http.post(apiUrl, this.form.value).subscribe({
```

```
      next: (res) => {
        console.log('API Response:', res);
        this.message = 'Form submitted successfully!';
        this.submittedData = res; // Save API response
      },
      error: (err) => {
        console.error(err);
        this.message = 'Error submitting form.';
      }    });  }}
```

register.html

```
<h2>Submit Form</h2>
<form [formGroup]="form" (ngSubmit)="onSubmit()">

  <label>Name:</label>
  <input type="text" formControlName="name" />
  <div class="error" *ngIf="form.get('name')?.touched && form.get('name')?.invalid">
    Name is required
  </div>
  <br />
  <label>Email:</label>
  <input type="email" formControlName="email" />
  <div class="error" *ngIf="form.get('email')?.touched && form.get('email')?.invalid">
    Valid email is required
  </div>
  <br />
  <button type="submit" [disabled]="form.invalid">Submit</button>
  <p style="color: green; margin-top: 10px;">{{ message }}</p>
</form>
<div *ngIf="submittedData" style="margin-top:20px; padding:10px; border:1px solid #ccc;">
  <h3>Submitted Data:</h3>
  <pre>{{ submittedData | json }}</pre>
</div>
```

❖ **Form State Management**

- Angular forms have built-in **states** that help manage **validation, user interaction, and submission**:

Property	Description
valid	true if form passes all validators
invalid	true if form fails validation
touched	true if user focused and left a control
untouched	Opposite of touched
dirty	true if value changed by user
pristine	Opposite of dirty
pending	true if async validator is running

Example:

app.ts

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators, ReactiveFormsModule } from '@angular/forms';
import { AbstractControl, ValidationErrors } from '@angular/forms';
import { Observable, of } from 'rxjs';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './app.html',
})

export class App implements OnInit {
  myForm!: FormGroup;
  constructor(private fb: FormBuilder) {}
  fakeAsyncValidator(control: AbstractControl): Observable<ValidationErrors | null> {
    return of(null).pipe(delay(1500));
  }

  ngOnInit() {
    this.myForm = this.fb.group({
      name: ['', {
        validators: [Validators.required],
        asyncValidators: [this.fakeAsyncValidator.bind(this)],
        updateOn: 'change'
      }]
    });  }}
```

app.html

```
<form [formGroup]="myForm">
  <label>Name:</label>
  <input formControlName="name" />
  <hr />
  <h3>Form Control States</h3>
  <p>Valid: {{ myForm.get('name')?.valid }}</p>
  <p>Invalid: {{ myForm.get('name')?.invalid }}</p>

  <p>Touched: {{ myForm.get('name')?.touched }}</p>
  <p>Untouched: {{ myForm.get('name')?.untouched }}</p>

  <p>Dirty: {{ myForm.get('name')?.dirty }}</p>
  <p>Pristine: {{ myForm.get('name')?.pristine }}</p>
  <p>Pending (async validation running): {{ myForm.get('name')?.pending }}</p>
</form>
```

Output:

- **Before typing or touching the input:**

valid: false
invalid: true
touched: false
untouched: true
dirty: false
pristine: true
pending: false

- **After clicking and leaving input (blur event):**

touched: true
untouched: false

- **After typing:**

dirty: true
pristine: false
valid OR invalid (depends on rules)

- **When async validator runs (for 1.5 sec):**

pending: true

- **When async validation completes:**

pending: false

1.4 Building Reusable UI Components & Design Patterns

- Building reusable UI components in Angular involves using modern features like **standalone components** and applying specific **design patterns** to ensure consistency, scalability, and maintainability across an application or multiple projects.
- We'll create a reusable **Button Component** and show how to use it anywhere in your app.
  Example: (**ng g c ui-button)**

ui-button.ts

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

@Component({
  selector: 'ui-button',
  standalone: true,
  templateUrl: './ui-button.html',
  styleUrls: ['./ui-button.css']
})
export class UIButton {
  @Input() label = 'Button';  // Button text
  @Input() type: 'primary' | 'secondary' = 'primary';  // Button style
  @Output() clicked = new EventEmitter<void>();  // Click event

  handleClick() {
    this.clicked.emit();
  }}
```

ui-button.html

```
<button
  [class.primary]="type === 'primary'"
  [class.secondary]="type === 'secondary'"
  (click)="handleClick()"
>
  {{ label }}
</button>
```

ui-button.css

```
button {
  padding: 8px 16px;
  border-radius: 4px;
  border: none;
  cursor: pointer;
  font-size: 14px;
  color: white;
}

.primary {
  background-color: #007bff;
}

.secondary {
  background-color: #6c757d;
}
```

app.ts

```
import { Component, signal } from '@angular/core';
import { UIButton } from './ui-button/ui-button';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [UIButton],
  templateUrl: './app.html',
  styleUrls: ['./app.css']
})
export class App {
  // Example signal to demonstrate Angular 17 reactive data
  title = signal('Reusable Button Example');

  onPrimaryClick() {
    alert('Primary button clicked!');
  }

  onSecondaryClick() {
    alert('Secondary button clicked!');
  }
}
```

app.html

```
<h1>{{ title() }}</h1>

<ui-button
  label="Save"
```

```
  type="primary"
  (clicked)="onPrimaryClick()"
></ui-button>

<ui-button
  label="Cancel"
  type="secondary"
  (clicked)="onSecondaryClick()"
></ui-button>
```

app.css

```
h1 {
  font-family: Arial, sans-serif;
  margin-bottom: 16px;
}

ui-button {
  margin-right: 8px;
}
```

1.4.1 Creating reusable card, modal, and alert components

- **Reusable card Component**
  - Creating a reusable card component in Angular involves using **@Input()** for simple data passing, or more advanced techniques like **content projection (<ng-content>)** and **template outlets (*ngTemplateOutlet)** for greater flexibility.

Method 1: Using @Input() (Simple Cards)

- For a simple card where only the basic data changes (like a title and some text), using the @Input() decorator is the most straightforward approach.

1. Generate the component:

```
ng generate component card
```

2. Define the inputs in card.ts:

card.ts

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

@Component({
  selector: 'app-card',
  standalone: true, // Use 'standalone: true' in modern Angular
  templateUrl: './card.html',
  styleUrls: ['./card.css']
})
export class CardC {
  @Input() title: string = '';
  @Input() content: string = '';
}
```

3. Define the template in card.html:

card.html

```
<div class="card">
  <div class="card-header">
    <h2>{{ title }}</h2>
  </div>
  <div class="card-body">
    <p>{{ content }}</p>
  </div>
</div>
```

4. Use it in a parent component:
app.ts

```
import { Component, signal } from '@angular/core';
import { UIButton } from './ui-button/ui-button';
import { CardC } from './card/card';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CardC],
  templateUrl: './app.html',
  styleUrls: ['./app.css']
})
export class App {
  title = signal('Reusable Button Example');

  onPrimaryClick() {
    alert('Primary button clicked!');
  }

  onSecondaryClick() {
    alert('Secondary button clicked!');
  }
}
```

app.html

```
<app-card
  title="First Card Title"
  content="This is the content for the first card.">
</app-card>

<app-card
  title="Second Card Title"
  content="This is different content for the second card.">
</app-card>
```

Method 2: Using Content Projection (<ng-content>) (Flexible Cards)

- For cards where the internal structure might vary (e.g., one card has an image, another has a list), **content projection** (or transclusion) allows the parent component to define the entire body of the card.

1. Modify the template in card.html:

card.html

```
<div class="card">
  <div class="card-header">
    <!-- Optional: use <ng-content select="[card-header]"> for specific slots -->
    <ng-content select="[card-header]"></ng-content>
  </div>
  <div class="card-body">
    <ng-content></ng-content>
  </div>
</div>
```

card.ts

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-card',
  standalone: true, // Use 'standalone: true' in modern Angular
  templateUrl: './card.html',
  styleUrls: ['./card.css']
})
export class CardC {
  @Input() title: string = '';
  @Input() content: string = '';
}
```

2. Use it in a parent component:

- The parent now decides what goes inside the app-card tags:

app.html

```
<app-card>
  <div card-header>
    <h2>Image Card</h2>
  </div>
<img src="assets/pics.jpg" alt="My Image" width="250" height="250">
  <p>This card has an image inside.</p>
</app-card>
<app-card>
  <div card-header>
    <h2>List Card</h2>
  </div>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul></app-card>
```

Note:

- Image should be in src/assets/ folder.
- In angular.json file add following code if not exist:
  "assets": [

```
"src/favicon.ico",
"src/assets"    ]
```

- **Reusable modal Component**
  - Creating a reusable modal component in Angular is slightly more involved than a card component, as modals typically require managing visibility, handling user interactions (like closing), and preventing background scrolling.
  - The most robust approach involves using **content projection** to allow the modal's content to be customized, and a **service** to manage opening and closing the modal from various parts of your application.

1. Generate the component:

```
ng generate component modal
```

- **2. Define the template (modal.html):**
  We use ng-content here so the parent component can insert custom content.

modal.html

```
<div *ngIf="isOpen" class="modal-overlay" (click)="close()">
  <div class="modal-content" (click)="$event.stopPropagation()">
    <div class="modal-header">
      <!-- Slot for custom header (optional) -->
      <ng-content select="[modal-header]"></ng-content>
      <button class="close-button" (click)="close()">&times;</button>
    </div>
    <div class="modal-body">
      <!-- Main content slot -->
      <ng-content></ng-content>
    </div>
    <div class="modal-footer">
      <!-- Slot for custom footer/buttons (optional) -->
      <ng-content select="[modal-footer]"></ng-content>
    </div>
  </div>
</div>
```

3. Define the logic (modal.ts):

- We manage the isOpen state and allow the parent to hook into the close event using @Output().

modal.ts

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

@Component({
  selector: 'app-modal',
  standalone: true,
```

```
  imports: [CommonModule],
  templateUrl: './modal.html',
  styleUrls: ['./modal.css']
})
export class Modal {
  isOpen = false;
  @Output() modalClose = new EventEmitter<void>();

  open() {
    this.isOpen = true;
    document.body.classList.add('modal-open'); // Prevents background scrolling
  }

  close() {
    this.isOpen = false;
    this.modalClose.emit();
    document.body.classList.remove('modal-open');
  }
}
```

4. Add basic styling (modal.css):

```
.modal-overlay {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, 0.6);
  display: flex;
  justify-content: center;
  align-items: center;
  z-index: 1000;
}
.modal-content {
  background: white;
  padding: 20px;
  border-radius: 5px;
  max-width: 500px;
  width: 90%;
  z-index: 1001;
}
.modal-header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  border-bottom: 1px solid #eee;
  padding-bottom: 10px;
}
.close-button {
  cursor: pointer;
  border: none;
```

```
  background: none;
  font-size: 1.5rem;
}
/* Add this to your main styles.scss or global styles file to prevent body scroll */
body.modal-open {
  overflow: hidden;
}
```

Step 2: Use the Modal in a Parent Component

- You can place the modal component in your main application template and control its visibility using ViewChild.

1. Use it in app.component.html (or another parent):

app.html

```
<button (click)="openMyModal()">Open Modal</button>
<app-modal #myModal (modalClose)="handleModalCloseEvent()">
  <h2 modal-header>Custom Modal Title</h2>
    <!-- Main body content -->
  <p>This is dynamic content projected into the modal body.</p>
    <div modal-footer>
    <button (click)="myModal.close()">Cancel</button>
    <button class="primary-btn">Save Changes</button>
  </div>
</app-modal>
```

2. Control it from the parent's logic (app.component.ts):

app.ts

```
import { Component, ViewChild } from '@angular/core';
import { Modal } from './modal/modal';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [Modal],
  templateUrl: './app.html',
})
export class App {
  // Access the child component instance
  @ViewChild('myModal') private modalComponent!: Modal;
  openMyModal() {
    this.modalComponent.open();
  }
  handleModalCloseEvent() {
    console.log('Modal was closed by user interaction.');
  }}
```

- **Reusable alert Component**

- Creating a reusable alert component in Angular is an excellent use case for combining **@Input()** properties to customize the alert's appearance (e.g., success, danger, warning) and an optional **service** to display temporary, dynamic alerts across the application.

1. Generate the component:

ng generate component alert

2. Define the logic and inputs (alert.component.ts):

- We use an input for the type to apply different CSS classes dynamically using ngClass.

alert.ts

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

@Component({
  selector: 'app-alert',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './alert.html',
  styleUrls: ['./alert.css']
})
export class Alert {
  // Can be 'info', 'success', 'warning', or 'danger'
  @Input() type: 'info' | 'success' | 'warning' | 'danger' = 'info';
  @Input() dismissible: boolean = false;
  isVisible = true;
  close() {
    this.isVisible = false;
  }}
```

3. Define the template (alert.html):

- We use ngClass to apply the correct styling based on the type input and <ng-content> for the message body.

alert.html

```
<div *ngIf="isVisible" class="alert" [ngClass]="'alert-' + type">
  <div class="alert-content">
    <ng-content></ng-content>
  </div>
  <button *ngIf="dismissible" class="close-btn" (click)="close()">
    &times;
  </button>
</div>
```

4. Add basic styling (alert.css):

- Define styles for different alert types.

alert.css

```
.alert {
  padding: 15px;
  margin-bottom: 20px;
  border: 1px solid transparent;
  border-radius: 4px;
  display: flex;
  justify-content: space-between;
  align-items: center;
}

.alert-info {
  color: #31708f;
  background-color: #d9edf7;
  border-color: #bce8f1;
}

.alert-success {
  color: #3c763d;
  background-color: #dff0d8;
  border-color: #d6e9c6;
}
.alert-warning {
  color: #8a6d3b;
  background-color: #fcf8e3;
  border-color: #faebcc;
}
.alert-danger {
  color: #a94442;
  background-color: #f2dede;
  border-color: #ebccd1;
}
.close-btn {
  background: none;
  border: none;
  font-size: 1.5rem;
  cursor: pointer;
  margin-left: 15px;
}
```

5. Using the Alert Component

- You can now use the alert component anywhere in your application just by adding its selector and setting the inputs.

app.html

```
<h3>Static Alerts</h3>

<app-alert type="success">
  <strong>Success!</strong> Your operation was successful.
</app-alert>

<app-alert type="warning" [dismissible]="true">
  <strong>Warning!</strong> Something might be wrong.
</app-alert>

<app-alert type="danger">
  <strong>Error!</strong> An error occurred.
</app-alert>
```

app.ts

```
import { Component, ViewChild } from '@angular/core';
import { Alert } from './alert/alert';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [Alert],
  templateUrl: './app.html',
})
export class App {
}
```

1.4.2 Component interaction with RxJS and Shared Services

- **What are RxJS Operations?**
  - In Angular applications, RxJS (Reactive Extensions for JavaScript) is a powerful library used for handling asynchronous and event-based programming using observables.
  - RxJS provides a wide range of operators that enable you to manipulate, transform, combine, and manage observables in a flexible and functional way.
  - These operators make it easier to work with data streams and asynchronous operations in a reactive manner.
- **What is a Shared Service?**
  - A **shared service** is a service intentionally designed to be used by **multiple components**, often to:
    - Share data
    - Synchronize state
    - Send messages/events
    - Communicate between sibling or deeply nested components
    - Broadcast updates
  - Most commonly implemented using:
    - **RxJS:** BehaviorSubject, Subject, ReplaySubject or **Signals** in Angular 17

Example:

Component A sends a message → Component B receives it(using RxJS BehaviorSubject inside a shared service)

Step 1: Create a Shared Service(ng g s share-service)

share-service.ts

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class ShareService {
  // BehaviorSubject holds and broadcasts data
  private messageSource = new BehaviorSubject<string>('Hello from service');

  // Expose as observable
  message$ = this.messageSource.asObservable();

  // Method to update the message
  updateMessage(newMessage: string) {
    this.messageSource.next(newMessage);
  }}
```

Step 2: Create component (ng g c component-A) Inject the Shared Service in Component A (Sender)

component-a.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { ShareService } from '../share-service';

@Component({
  selector: 'component-a',
  standalone: true,
  imports: [FormsModule],
  template: `
    <h3>Component A (Sender)</h3>
    <input [(ngModel)]="msg" placeholder="Type message" />
    <button (click)="send()">Send</button>
  `
})
export class ComponentA {
  private shared = inject(ShareService);
  msg = '';

  send() {
    this.shared.updateMessage(this.msg);   // Send message to service
  }}
```

Step 3: Create component (ng g c component-B) Inject the Shared Service in Component B (Receiver)

component-b.ts

```
import { Component, inject } from '@angular/core';
import { ShareService } from '../share-service';

@Component({
  selector: 'component-b',
  standalone: true,
  template: `
    <h3>Component B (Receiver)</h3>
    <p>Received message: {{ message }}</p>
  `
})
export class ComponentB {
  private shared = inject(ShareService);
  message = '';

  ngOnInit() {
    // Subscribe to message$ stream
    this.shared.message$.subscribe(msg => {
      this.message = msg;
    });
  }}
```

Step 4: Put Both Components in App Component

app.ts

```
import { Component } from '@angular/core';
import { ComponentA } from './component-a/component-a';
import { ComponentB } from './component-b/component-b';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ComponentA, ComponentB],
  template: `
    <h2>RxJS Shared Service Example</h2>

    <component-a></component-a>
    <hr />
    <component-b></component-b>
  `
})
export class App{}
```

1.4.3 Use of ng-template, ng-container, ng-content for structural flexibility

- In **Angular 17**, ng-template, ng-container, and ng-content are powerful tools for **structural flexibility** in component templates.
- They serve different purposes but often work together to create **dynamic, reusable, and efficient UI structures**.

1. ng-template – Define Reusable Template Fragments

- **Purpose**: Holds HTML that is **not rendered by default** but can be instantiated dynamically.
- **Key Points**:

- Inspired by the native <template> element.
- Works with structural directives like *ngIf, *ngFor, or ngTemplateOutlet.
- Useful for **conditional rendering** and **dynamic content projection**.

app.html

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

<div *ngIf="isLoading; else content">
  <ng-container *ngTemplateOutlet="loading"></ng-container>
</div>

<ng-template #content>
  <p>Data loaded successfully!</p>
</ng-template>
```

app.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './app.html',
})
export class App {
  isLoading = true;

  ngOnInit() {
    // Simulate API call
    setTimeout(() => {
      this.isLoading = false;
    }, 2000);
  }}
```

2. ng-container – Logical Grouping Without Extra DOM

- **Purpose**: Groups elements **without adding extra HTML tags** to the DOM.
- **Key Points**:
  - Useful for applying structural directives to multiple elements.
  - Helps keep DOM clean and semantic.
  - Often used with *ngIf or *ngFor.

app.ts

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

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './app.html'
})
export class App {
  isLoggedIn = false;

  toggle() {
    this.isLoggedIn = !this.isLoggedIn;
  }}
```

app.html

```
<button (click)="toggle()">
  {{ isLoggedIn ? 'Logout' : 'Login' }}
</button>

<ng-container *ngIf="isLoggedIn; else guest">
  <p>Welcome back! You are logged in.</p>
</ng-container>

<ng-template #guest>
  <p>You are not logged in.</p>
</ng-template>
```

3. ng-content – Content Projection (Slots)

- **Purpose**: Allows **parent components** to project their own HTML into a child component.
- **Key Points**:
  - Similar to **slots** in Web Components.
  - Supports **selective projection** using select attribute.
  - Enables **highly reusable components**.

Example: (ng g c cards)

cards.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-card',
  standalone: true,
  template: `
    <div style="border:1px solid #ccc; padding:10px; border-radius:5px;">
      <ng-content></ng-content>
    </div>  `
})
export class Cards{}
```

app.html

```
<h2>Using ng-content (Simple Example)</h2>
```

```
<app-card>
  <h3>Welcome!</h3>
  <p>This content is inserted into the card using ng-content.</p>
</app-card>
```

app.ts

```
import { Component } from '@angular/core';
import { Cards } from './cards/cards';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [Cards],
  templateUrl: './app.html'
})
export class App {}
```

Directive	Purpose	Renders In DOM?	Typical Usage
ng-template	Template that is stored (not rendered unless used)	No	Conditional blocks, dynamic rendering, template refs
ng-container	Logical container without DOM element	No	Grouping structural directives without extra divs
ng-content	Inserts parent HTML into child component	Yes (projected content)	Building flexible UI components (cards, modals, layouts)

1.4.4 Smart vs Dumb Components: Best practices

- The **Smart vs. Dumb component pattern** is a core best practice in Angular architecture that separates presentational logic from business logic, leading to better organization, reusability, and testability.

- **Smart Component**
  - Smart components are the "brains" of the application.
  - They are typically top-level components on a page or within a specific feature.
  - Responsible for **logic, data fetching, state management**
  - Know about services, HTTP calls, NgRx, RxJS, Signals
  - Pass data **down** to child components via @Input()
  - Listen to events from children via @Output()
  - Often called **"controllers"** of a feature

- **Dumb Components (Presentational/Pure Components)**
  - Dumb components are focused solely on the UI and presentation.
  - They are highly reusable building blocks that are unaware of the application's broader context.
  - Responsible for **displaying data**
  - **Do not know about services or global state**
  - Receive data via @Input()
  - Emit events via @Output()
  - Highly reusable and testable

- Often purely HTML + small logic for UI

- **Benefits of Separating Smart and Dumb Components**

Benefit	Explanation
Reusability	Dumb components can be used in multiple places because they are independent
Testability	Dumb components are easier to unit test (no services or HTTP)
Separation of concerns	Keeps UI logic and business logic separate
Easier maintenance	Smart components handle state and data; dumb components handle rendering
Performance optimization	Smart components can decide when to re-render or fetch data

- **Best Practices**
  - **Smart Components**

  - Keep them at **container level** (feature folder)
  - Handle: HTTP calls, state management, business logic
  - Use **RxJS or Signals** to manage streams of data
  - Don't contain any **presentation logic**
  - Pass **data to dumb components via @Input()**
  - Listen for child events via @Output()

  - **Dumb Components**
  - Only care about **how to display data**
  - **No services, no state management**
  - Take **data via @Input()**
  - Emit actions via **@Output()**

user-list.ts (Dumb)

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

@Component({
  selector: 'app-user-list',
  standalone: true,
   imports: [CommonModule],
  template: `
    <ul>
      <li *ngFor="let user of users">
        {{ user.name }}
        <button (click)="deleteUser.emit(user.id)">Delete</button>
      </li>
    </ul>
  `
})
export class UserList {
  @Input() users: { id: number; name: string }[] = [];
  @Output() deleteUser = new EventEmitter<number>();
}
```

user-container.ts (smart)

```
import { Component, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { UserList } from '../user-list/user-list';
import { UserService } from '../user-service';


@Component({
  selector: 'app-user-container',
  standalone: true,
  imports: [CommonModule, UserList],
  template: `
    <h2>User Management</h2>
    <app-user-list
      [users]="users"
      (deleteUser)="onDelete($event)">
    </app-user-list>
  `
})
export class UserContainer {
  private userService = inject(UserService);
  users = this.userService.getUsers();

  onDelete(userId: number) {
    this.userService.deleteUser(userId);
    this.users = this.userService.getUsers(); // update view
  }}
```

- **user-service.ts(ng g s userService)**

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

@Injectable({ providedIn: 'root' })
export class UserService {
  private users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' }
  ];
  getUsers() {
    return [...this.users]; // return a copy
  }
  deleteUser(id: number) {
    this.users = this.users.filter(user => user.id !== id);
  }}
```

app.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { UserContainer } from './user-container/user-container';
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, UserContainer],
  template: `
    <app-user-container></app-user-container>
  `
})
export class App {}
```

Aspect	Smart Component	Dumb Component
Also called	Container Component	Presentational Component
Purpose	Manages data, state, API calls, business logic	Displays data, handles UI, emits events
Data handling	Knows about services, stores, or Signals	Receives data via @Input()
Events	Listens to child events via @Output()	Emits events via @Output()
Reusability	Low (specific to feature)	High (can be reused anywhere)
Testing	Harder (needs mocking services)	Easier (pure UI logic)
DOM	May contain minimal HTML	Purely HTML/Template, possibly ng-content

1.5 Application Deployment & Performance Optimization

- Application deployment in Angular involves creating a production-ready build and hosting the static files on a web server or cloud platform.
- Performance optimization is an ongoing process that uses specific techniques to ensure speed, responsiveness, and a great user experience.
- Everything you need to take an Angular app from development → optimized → deployed.

1.5.1 Angular build process, environments, and optimization flags

- **Angular Build Process**

  - When you run: ng build

Angular performs these steps:

1. **TypeScript Compilation**
   - Converts TypeScript → JavaScript
   - Validates types
2. **Ahead-of-Time (AOT) Compilation**
   - Angular templates (HTML) are compiled at build time.
   - This removes the Angular compiler from the final bundle → smaller + faster.
3. **Bundling**
   - Webpack groups your app into optimized JS bundles:
     - main.js
     - styles.css
     - vendor.js
     - lazy-loaded chunk.js

4 . Minification & Uglification

- Removes whitespace
- Shortens variable names
- Removes unused code (**tree-shaking**)

5. **Asset Optimization**
   - Compress images

- Optimize fonts
- Hash filenames for cache-busting (main.093f1.js)

- **Angular Environments**
  - Angular uses environment files to switch configuration between **development** and **production**.

    environment.ts (development):
    ```
    export const environment = {
      production: false,
      apiUrl: 'http://localhost:3000'
    };
    ```

    environment.prod.ts (production):
    ```
    export const environment = {
      production: true,
      apiUrl: 'https://api.myapp.com'
    };
    ```

  - Angular replaces the file automatically during build.

- **Optimization Flags**
  - The most common build flags:
    - **Production build**

      ng build --configuration production

    Enables:
    - AOT
    - Minification
    - Tree-shaking
    - Script optimization
    - Bundle hashing

    - **Build with source maps (debug production)**

      ng build --configuration production --source-map

    - **Build with stats for analysis**

      ng build --configuration production --stats-json

    - **Disable build optimizations (debug only)**

      ng build --configuration development

1.5.2 Deploying Angular Applications Using Firebase Hosting

- Firebase Hosting is one of the easiest ways to deploy Angular apps.

Step 1: Install Firebase CLI

npm install -g firebase-tools

Step 2: Login

firebase login

Step 3: Initialize Firebase

Inside your Angular root folder:

firebase init hosting

- Choose:
- Hosting

"dist" folder path: dist/YOUR_APP_NAME/browser (Angular v17+)

or

dist/YOUR_APP_NAME/ (Angular v16 and below)

Step 4: Build the Angular app

ng build --configuration production

Step 5: Deploy

firebase deploy

You get:

- HTTPS by default
- Free global CDN
- Automatic caching
- Instant URL

Example URL:

https://yourapp.web.app

1.5.3 Performance Tuning: trackBy, OnPush, Lazy Loading

1. **trackBy in ngFor**
   - Angular re-renders the full list **unless trackBy is used**.

   Bad (slow):

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

   Good (fast):

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

   In component:

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

   - Prevents re-rendering when other items change
   - Improves list performance (especially large lists)

2. **OnPush Change Detection**
   - Default Angular change detection runs frequently.
   - Using OnPush tells Angular to update only when:
     - @Input changes
     - An event triggers change
     - A signal/observable emits

   Add to component:

   ```
   import { ChangeDetectionStrategy } from '@angular/core';
   @Component({
     ...
     changeDetection: ChangeDetectionStrategy.OnPush
   })
   export class UserList {}
   ```

   - Reduces CPU usage
   - Makes apps feel faster

3. **Lazy Loading Routes**
   - Loads code only when needed.

   Routing example:

   ```
   {
     path: 'admin',
     loadChildren: () =>
   ```

```
    import('./admin/admin.module').then(m => m.AdminModule)
}
```

Or Angular standalone components:

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

- Reduces initial bundle
- Faster load time

4. **Lazy Loading Components**
   - Use *ngIf + dynamic imports:

```
async openDetails() {
  const { DetailsComponent } = await import('./details.component');
}
```

Notes:

Topic	**Key Idea**	**Benefit**
Build process	AOT, minify, tree-shake	Smaller + faster bundles
Environments	Dev/prod replacements	Correct API endpoints
Optimization flags	--configuration production	Fast production builds
Firebase Hosting	CLI deploy	Fast, free, global CDN
trackBy	Identifies list items	Faster rendering
OnPush	Limited change detection	Big performance boost
Lazy loading	Load only needed code	Faster initial load