Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

SERVICES, DEPENDENCY INJECTION & STATE MANAGEMENT (WEEKS 5-6)

CREATING AND USING ANGULAR SERVICES

CHAPTER 1: INTRODUCTION TO ANGULAR SERVICES

1.1 What Are Angular Services?

In **Angular**, services are **reusable pieces of logic** that help manage **data, business logic, and API calls** across multiple components.

✓ **Encapsulates business logic** – Keeps components lightweight and focused.
✓ **Facilitates data sharing** – Shares data between components without duplicating logic.
✓ **Improves modularity** – Promotes code reusability and separation of concerns.
✓ **Handles API calls** – Manages communication with backend services.

1.2 Why Use Services in Angular?

✓ **Prevents redundant code** by centralizing logic.
✓ **Easier maintenance** – Any updates to logic affect all consuming components.

✓ **Improves testability** – Services can be **unit tested independently**.

CHAPTER 2: CREATING AN ANGULAR SERVICE

2.1 Generating a Service Using Angular CLI

Use Angular CLI to generate a new service:

ng generate service user

or

ng g s user

✓ Creates user.service.ts inside the src/app/ directory.

2.2 Structure of an Angular Service

A basic service contains:

- **@Injectable() decorator** – Marks it as a service that can be injected.
- **Methods** for handling business logic.

Example: A Simple User Service (user.service.ts)

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


@Injectable({
  providedIn: 'root' // Makes service available throughout the app
})
export class UserService {
  private users = ['Alice', 'Bob', 'Charlie'];
```

```
  constructor() { }

  getUsers() {
    return this.users;
  }
}
```

✓ **Encapsulates data logic** (users array).
✓ **Provides a getUsers() method** to fetch user data.

CHAPTER 3: INJECTING AND USING A SERVICE IN A COMPONENT

3.1 What is Dependency Injection (DI)?

Dependency Injection (DI) is a **design pattern** used in Angular to **inject services into components**.

✓ **Reduces component dependency** on data sources.
✓ **Improves testability and maintainability**.

3.2 Injecting a Service into a Component

Modify app.component.ts to use UserService:

```
import { Component, OnInit } from '@angular/core';
import { UserService } from './user.service';

@Component({
```

```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  users: string[] = [];

  constructor(private userService: UserService) {}

  ngOnInit() {
    this.users = this.userService.getUsers();
  }
}
```

✓ UserService is injected via the **constructor** (private userService: UserService).
✓ ngOnInit() calls getUsers() to fetch the user list.

3.3 Displaying the Data in HTML

Modify app.component.html to display the fetched data:

```
<h2>Users List</h2>
<ul>
  <li *ngFor="let user of users">{{ user }}</li>
</ul>
```

✓ Loops through users and displays them dynamically.

CHAPTER 4: USING SERVICES TO FETCH DATA FROM AN API

4.1 Installing and Importing HttpClientModule

To fetch data from an API, we need **HttpClientModule**.

Step 1: Import HttpClientModule in app.module.ts

```
import { HttpClientModule } from '@angular/common/http';


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

✓ Enables **HTTP requests** in Angular.

4.2 Creating an API Service

Modify user.service.ts to fetch users from an API:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';


@Injectable({
  providedIn: 'root'
```

```
})
export class UserService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/users';

  constructor(private http: HttpClient) {}

  getUsers(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
}
```

✓ HttpClient fetches data from apiUrl.
✓ The method returns an **Observable,** enabling **asynchronous operations**.

4.3 Consuming the API in a Component

Modify app.component.ts to call the API:

```
export class AppComponent implements OnInit {
  users: any[] = [];

  constructor(private userService: UserService) {}

  ngOnInit() {
    this.userService.getUsers().subscribe(data => {
```

```
    this.users = data;
  });
 }
}
```

✓ .subscribe() handles **asynchronous API response**.

4.4 Handling Errors in API Calls

Modify user.service.ts to catch errors:

```
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

getUsers(): Observable<any> {
 return this.http.get(this.apiUrl).pipe(
  catchError(error => {
   console.error('Error fetching users:', error);
   return throwError(error);
  })
 );
}
```

✓ Logs and returns **error messages** when API calls fail.

CHAPTER 5: CREATING A SHARED SERVICE FOR STATE MANAGEMENT

5.1 What is a Shared Service?

A **shared service** allows multiple components to **share data and interact** without directly communicating.

✓ **Useful for state management** (e.g., shopping cart, user authentication).
✓ **Prevents unnecessary API calls** by storing shared data.

5.2 Example: Shared Authentication Service

Step 1: Create auth.service.ts

ng g s auth

Step 2: Implement Authentication Logic

Modify auth.service.ts to manage user authentication:

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


@Injectable({
 providedIn: 'root'
})
export class AuthService {
 private isLoggedIn = false;


 login() {
  this.isLoggedIn = true;
 }
```

```
 logout() {
  this.isLoggedIn = false;
 }

 checkLoginStatus() {
  return this.isLoggedIn;
 }
}
```

✓ **Manages login/logout state**.

5.3 Injecting AuthService in Components

Modify app.component.ts:

```
export class AppComponent {
 constructor(private authService: AuthService) {}

 login() {
  this.authService.login();
 }

 logout() {
  this.authService.logout();
```

```
 }


 isLoggedIn(): boolean {
   return this.authService.checkLoginStatus();
 }
}
```

Modify app.component.html:

```
<button *ngIf="!isLoggedIn()" (click)="login()">Login</button>
<button *ngIf="isLoggedIn()" (click)="logout()">Logout</button>
<p *ngIf="isLoggedIn()">Welcome, user!</p>
```

✓ Clicking **Login** updates the state in AuthService.
✓ Components can **share login status** without direct communication.

Case Study: How a SaaS Company Used Angular Services for API Calls

Background

A SaaS company built a **CRM dashboard** for tracking customer interactions.

Challenges

✓ **Redundant API calls** slowed down performance.
✓ **Data inconsistencies** between different pages.
✓ **Complex logic inside components** made debugging difficult.

Solution: Using Angular Services for API Calls

✓ Moved all **API logic into a shared service**.
✓ Used **RxJS caching** to prevent duplicate API calls.
✓ Implemented a **singleton service** for authentication.

```
@Injectable({
 providedIn: 'root'
})
export class CustomerService {
 private cache: any = null;
 private apiUrl = 'https://api.crm.com/customers';

 constructor(private http: HttpClient) {}

 getCustomers(): Observable<any> {
  if (this.cache) {
   return of(this.cache);
  }
  return this.http.get(this.apiUrl).pipe(
   tap(data => this.cache = data)
  );
 }
}
```

Results

🚀 **Reduced API calls by 70%,** improving performance.
🔍 **Easier debugging** due to separation of concerns.
⚡ **Consistent data across all dashboard components**.

By using **Angular Services,** the company **optimized performance and improved maintainability**.

Exercise

1. Create an Angular **User Service** that fetches user data from an API.
2. Inject the service into a component and display the **user list**.
3. Add error handling for API failures.
4. Create an **Auth Service** for managing login/logout state.

Conclusion

In this section, we explored:
✓ **How Angular Services manage API calls and business logic**.
✓ **How Dependency Injection (DI) simplifies service usage**.
✓ **How shared services help in state management**.

UNDERSTANDING DEPENDENCY INJECTION (DI) IN ANGULAR

CHAPTER 1: INTRODUCTION TO DEPENDENCY INJECTION (DI) IN ANGULAR

1.1 What is Dependency Injection (DI)?

Dependency Injection (DI) is a **design pattern** in which a class receives its dependencies from an external source rather than creating them itself.

✓ **Promotes modularity and reusability** – Components and services remain independent.
✓ **Improves testability** – Easily mock dependencies for unit testing.
✓ **Enhances maintainability** – Reduces tight coupling between components.

In Angular, DI is a **core feature** that manages the creation and sharing of services across components.

1.2 How Dependency Injection Works in Angular

Angular has a **hierarchical dependency injection system,** which means:

✓ Services are provided at different levels (**root, module, or component**).
✓ Components can receive dependencies **without creating them**.
✓ Dependencies are **shared across the application** when needed.

Example of Dependency Injection

```
@Injectable({
```

```
  providedIn: 'root'
})
export class LoggerService {
  log(message: string) {
    console.log(message);
  }
}

@Component({
  selector: 'app-logger',
  template: `<button (click)="logMessage()">Log</button>`
})
export class LoggerComponent {
  constructor(private logger: LoggerService) {}

  logMessage() {
    this.logger.log("Button clicked!");
  }
}
```

✓ The **LoggerService** is injected into LoggerComponent, allowing reuse without creating a new instance.

CHAPTER 2: UNDERSTANDING PROVIDERS IN ANGULAR DI

2.1 What is a Provider?

A **provider** tells Angular **how to create a dependency**.

✓ Services need to be registered as **providers** in Angular's DI system.
✓ Providers define **how objects are created and managed**.

2.2 Registering Providers in Angular

Angular allows providers to be registered at **three levels**:

Provider Level	Scope
Root-Level (providedIn: 'root')	Available throughout the application.
Module-Level (providers: [] in @NgModule)	Available only within the module.
Component-Level (providers: [] in @Component)	Available only within the component.

Example: Providing a Service at Root Level

```
@Injectable({
  providedIn: 'root' // Service is available everywhere
})
export class AuthService {
  isAuthenticated(): boolean {
    return !!localStorage.getItem("userToken");
  }
}
```

✓ Services provided at the root level **stay in memory** and **are shared globally**.

CHAPTER 3: INJECTING SERVICES INTO COMPONENTS AND OTHER SERVICES

3.1 Injecting a Service into a Component

To use a service inside a component**, inject it via the constructor**.

Example: Injecting AuthService into a Component

```
@Component({
 selector: 'app-login',
 template: `<p *ngIf="isLoggedIn">Welcome back!</p>`
})
export class LoginComponent {
 isLoggedIn = false;

 constructor(private authService: AuthService) {
  this.isLoggedIn = authService.isAuthenticated();
 }
}
```

✓ The AuthService is **injected** into the LoginComponent**,** allowing access to authentication logic.

3.2 Injecting a Service into Another Service

Services can **depend on other services,** which Angular handles automatically.

Example: Injecting LoggerService into AuthService

```
@Injectable({
 providedIn: 'root'
})
export class AuthService {
 constructor(private logger: LoggerService) {}

 login(user: string) {
  this.logger.log(`User ${user} logged in.`);
 }
}
```

✓ The **AuthService depends on LoggerService,** but Angular **injects it automatically**.

CHAPTER 4: USING @INJECTABLE() FOR DEPENDENCY INJECTION

4.1 What is @Injectable()?

The @Injectable() decorator marks a class as **available for DI**.

✓ **Ensures proper dependency resolution**.
✓ **Allows services to be provided at different levels**.

Example: Using @Injectable()

```
@Injectable({
```

```
  providedIn: 'root'
})
export class NotificationService {
  showMessage(message: string) {
    alert(message);
  }
}
```

✓ Registers NotificationService **globally** in the DI system.

CHAPTER 5: COMPONENT-LEVEL DEPENDENCY INJECTION

5.1 Providing a Service at the Component Level

Services can be **limited to specific components** by defining them in providers.

Example: Component-Level Provider

```
@Component({
  selector: 'app-local-logger',
  template: `<button (click)="logMessage()">Log</button>`,
  providers: [LoggerService] // Creates a new instance for this component only
})
export class LocalLoggerComponent {
  constructor(private logger: LoggerService) {}
```

```
 logMessage() {
  this.logger.log("Component-level log!");
 }
}
```

✓ A **new instance** of LoggerService is created **only for LocalLoggerComponent**.

CHAPTER 6: HIERARCHICAL DEPENDENCY INJECTION IN ANGULAR

6.1 Understanding Angular's DI Hierarchy

Angular **resolves dependencies** based on the **closest provider definition**.

✓ Services provided **at the root level** are **shared globally**.
✓ Services provided **at the module level** are **shared within that module**.
✓ Services provided **at the component level** are **unique to that component**.

6.2 Example: Module-Level Provider

```
@NgModule({
 providers: [LoggerService] // Available only within this module
})
export class AdminModule {}
```

✓ Services declared **at the module level** are not accessible outside that module.

CHAPTER 7: USING @INJECT() FOR MANUAL DEPENDENCY INJECTION

7.1 What is @Inject()?

The @Inject() decorator allows **manual injection of dependencies**.

✓ Used when **tokens are required** (e.g., injecting values instead of classes).
✓ Helps inject **configuration settings or API URLs**.

Example: Injecting a Configuration Token

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


@Injectable({
  providedIn: 'root'
})
export class ApiService {
  constructor(@Inject('API_URL') private apiUrl: string) {}


  getEndpoint() {
    return this.apiUrl;
  }
}
```

✓ API_URL can be **defined globally** in Angular's DI system.

Case Study: How an E-Commerce Platform Used DI for Scalable Services

Background

An **e-commerce company** needed a scalable **order management system**.

Challenges

✓ **Difficulties in maintaining multiple services**.
✓ **Tightly coupled components,** making testing hard.
✓ **Performance issues** due to redundant service instances.

Solution: Implementing DI for Scalability

✓ Used **DI to inject services** instead of manually creating instances.
✓ Applied **root-level services** for shared logic.
✓ Used **module-specific providers** to optimize memory usage.

Results

🚀 **50% improvement in performance** by reducing redundant service instances.
🔍 **Easier unit testing,** improving debugging speed.
⚡ **Highly modular and maintainable codebase**.

By leveraging **Angular's Dependency Injection system,** the company **optimized performance and maintainability**.

Exercise

1. Create a service UserService that fetches user data.
2. Inject UserService into a component and display user details.
3. Use a **component-level provider** for LoggerService.
4. Inject a **configuration token** into a service using @Inject().

Conclusion

In this section, we explored:

✓ **How Dependency Injection (DI) improves code reusability and testability**.

✓ **How to inject services into components and other services**.

✓ **How to use @Injectable() and @Inject() for DI in Angular**.

✓ **How DI helps build scalable Angular applications**.

USING HTTPCLIENT FOR API CALLS IN ANGULAR

CHAPTER 1: INTRODUCTION TO HTTPCLIENT IN ANGULAR

1.1 Understanding HTTPClient

In **modern web applications,** fetching data from a server is essential for **dynamic content and interactivity**. Angular provides **HTTPClient,** a built-in module for making **API calls to external servers**.

✓ **Fetches data from REST APIs**.
✓ **Handles HTTP requests asynchronously**.
✓ **Supports GET, POST, PUT, DELETE methods**.
✓ **Manages responses with Observables (RxJS)**.
✓ **Includes built-in error handling and interceptors**.

1.2 Why Use HTTPClient Instead of Native Fetch?

Feature	**fetch() (Native)**	**HTTPClient (Angular)**
Observable Support	No	Yes
Automatic JSON Parsing	No (needs .json())	Yes
Error Handling	Manual	Built-in with interceptors
Interceptors	No	Yes (for authentication, logging)

Angular's HTTPClient is **optimized for handling API requests in Angular applications** and integrates **seamlessly with RxJS Observables**.

CHAPTER 2: SETTING UP HTTPCLIENT IN AN ANGULAR PROJECT

2.1 Importing HTTPClientModule

Before using HTTPClient, import the HttpClientModule in app.module.ts.

Example: Importing HTTPClientModule

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

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

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

import { HttpClientModule } from '@angular/common/http'; // Import HTTPClient


@NgModule({

 declarations: [AppComponent],

 imports: [BrowserModule, HttpClientModule], // Add HTTPClientModule

 providers: [],

 bootstrap: [AppComponent]

})

export class AppModule { }
```

✓ Enables HTTPClient in the entire application.

✓ Now, we can **make API requests using HTTPClient**.

CHAPTER 3: MAKING API CALLS WITH HTTPCLIENT

3.1 Using HTTPClient in a Service

Angular follows **a service-based approach** to manage API calls.

Step 1: Generate an API Service

ng generate service api

✓ Creates api.service.ts inside src/app/.

Step 2: Implement API Calls in api.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/users'; // Dummy API

  constructor(private http: HttpClient) { }
```

```
 // GET Request
 getUsers(): Observable<any> {
  return this.http.get(this.apiUrl);
 }
}
```

✓ Uses HttpClient to **fetch users from an API**.
✓ Returns an **Observable,** allowing real-time updates.

3.2 Fetching Data in a Component

Modify app.component.ts to **call the API service** and fetch users.

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from './api.service';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
 users: any = [];

 constructor(private apiService: ApiService) {}
```

```
 ngOnInit() {
  this.apiService.getUsers().subscribe(
   (data) => { this.users = data; },
   (error) => { console.error('Error fetching users', error); }
  );
 }
}
```

✓ Calls getUsers() when the component loads (ngOnInit).
✓ Updates users with the API response.
✓ Handles **errors gracefully** in catch block.

3.3 Displaying Data in HTML

Modify app.component.html to display **fetched users**.

```
<h2>User List</h2>
<ul>
 <li *ngFor="let user of users">
  {{ user.name }} - {{ user.email }}
 </li>
</ul>
```

✓ **Uses *ngFor to loop over API data dynamically**.
✓ The **data updates automatically** when received.

CHAPTER 4: MAKING DIFFERENT TYPES OF HTTP REQUESTS

4.1 Sending a POST Request (Creating Data)

Modify api.service.ts to **add a new user**.

```
addUser(user: any): Observable<any> {
  return this.http.post(this.apiUrl, user);
}
```

✓ Sends a **POST request** to **create a new user**.

Example: Calling addUser() in a Component

```
this.apiService.addUser({ name: 'John Doe', email: 'john@example.com' })
  .subscribe(response => console.log('User Added:', response));
```

✓ The **server saves the new user and returns confirmation**.

4.2 Sending a PUT Request (Updating Data)

Modify api.service.ts to **update user details**.

```
updateUser(id: number, user: any): Observable<any> {
  return this.http.put(`${this.apiUrl}/${id}`, user);
}
```

✓ Uses a **PUT request** to modify an **existing user**.

Example: Updating a User in a Component

```
this.apiService.updateUser(1, { name: 'Jane Doe' })
  .subscribe(response => console.log('User Updated:', response));
```

✓ **Sends the new data** and updates the user.

4.3 Sending a DELETE Request (Removing Data)

Modify api.service.ts to **delete a user**.

```
deleteUser(id: number): Observable<any> {
  return this.http.delete(`${this.apiUrl}/${id}`);
}
```

✓ Uses a **DELETE request** to remove a user by ID.

Example: Calling deleteUser() in a Component

```
this.apiService.deleteUser(1)
  .subscribe(response => console.log('User Deleted:', response));
```

✓ Deletes **user with ID 1** and logs confirmation.

CHAPTER 5: ERROR HANDLING IN HTTP REQUESTS

5.1 Handling Errors with catchError

Use **RxJS catchError** to handle **API failures gracefully**.

Modify api.service.ts to **handle errors globally**.

```
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

getUsers(): Observable<any> {
  return this.http.get(this.apiUrl).pipe(
    catchError((error) => {
```

```
    console.error('Error fetching users', error);
    return throwError(error);
  })
 );
}
```

✓ **Prevents the app from breaking** when the API fails.

Case Study: How an E-Commerce Website Used HTTPClient for Dynamic Content

Background

An e-commerce company needed **real-time product updates** from its API.

Challenges

✓ Products **weren't updating dynamically**.
✓ **High API request failure rates** affected the checkout process.
✓ **No centralized error handling**, making debugging difficult.

Solution: Implementing HTTPClient in Angular

✓ **Used HTTPClientModule for efficient API calls**.
✓ **Added catchError to handle API failures**.
✓ **Implemented GET, POST, PUT, DELETE requests** for dynamic product management.

Results

🚀 **50% faster API response times** due to efficient request handling.
🔍 **Reduced API errors** by **30%** with better error handling.

📈 **Increased user engagement,** as product details updated instantly.

By **optimizing API calls,** the e-commerce platform improved **performance and reliability**.

Exercise

1. Create a **service** named ProductService to fetch product data.
2. Implement GET, POST, PUT, and DELETE methods.
3. Call the API from a **component** and display product details.
4. Implement **error handling** for failed API requests.

Conclusion

In this section, we explored:

✓ **How to use HTTPClient for making API requests in Angular**.
✓ **How to handle GET, POST, PUT, and DELETE requests**.
✓ **How to handle errors gracefully with RxJS catchError**.

HANDLING OBSERVABLES WITH RXJS IN ANGULAR

CHAPTER 1: INTRODUCTION TO OBSERVABLES AND RXJS

1.1 What are Observables?

Observables are a core part of **Reactive Programming** in Angular. They allow handling **asynchronous data streams** efficiently and are provided by the **RxJS (Reactive Extensions for JavaScript) library**.

✓ **Observables manage asynchronous operations** – Handling API calls, real-time updates, and user input.
✓ **They provide a continuous data stream** – Unlike Promises, which handle only a single event.
✓ **They support multiple values over time** – Suitable for event-driven applications.

1.2 Why Use RxJS in Angular?

✓ **Simplifies async data handling** – No need for callbacks.
✓ **Supports powerful operators** – Allows filtering, mapping, and combining data streams.
✓ **Works seamlessly with Angular Services & HTTPClient** – Improves API call management.
✓ **Provides better error handling** – Ensures stable applications.

1.3 Installing RxJS in Angular

RxJS comes **pre-installed** with Angular. To check the installed version:

```
npm list rxjs
```

To install or update RxJS manually:

npm install rxjs

CHAPTER 2: CREATING AND SUBSCRIBING TO OBSERVABLES

2.1 Creating a Simple Observable

An Observable emits **data over time** and requires a **subscriber** to consume it.

Example: Basic Observable

```
import { Observable } from 'rxjs';


const observable = new Observable(observer => {
  observer.next('Hello');
  observer.next('RxJS');
  observer.complete();
});


observable.subscribe(value => console.log(value));
```

✓ **next()** emits values.
✓ **complete()** signals completion.
✓ The subscribe() method **listens for emitted values**.

2.2 Handling Observable Subscriptions

Observables can be **subscribed to** and later **unsubscribed** to prevent memory leaks.

Example: Unsubscribing from an Observable

```
import { Observable } from 'rxjs';


const observable = new Observable(observer => {
  setInterval(() => observer.next('Ping'), 1000);
});


const subscription = observable.subscribe(value =>
console.log(value));


setTimeout(() => {
  subscription.unsubscribe();
  console.log('Unsubscribed');
}, 5000);
```

✓ The unsubscribe() method **stops listening** after 5 seconds.

CHAPTER 3: USING OBSERVABLES WITH ANGULAR SERVICES

3.1 Using Observables in an Angular Service

Observables are commonly used in **Angular Services** to handle **API calls and real-time data**.

Step 1: Create a Service Using Angular CLI

ng generate service services/data

Step 2: Implement an HTTP Request Using Observables

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

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) {}

  getPosts(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
}
```

✓ **Returns an Observable** that emits API response data.
✓ Uses **Angular's HttpClient** to fetch data asynchronously.

3.2 Consuming the Service in a Component

Inject the service into a component and **subscribe to the Observable**.

```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../services/data.service';
```

```
@Component({
 selector: 'app-posts',
 templateUrl: './posts.component.html'
})
export class PostsComponent implements OnInit {
 posts: any[] = [];

 constructor(private dataService: DataService) {}

 ngOnInit() {
  this.dataService.getPosts().subscribe(
   data => this.posts = data,
   error => console.error("Error fetching posts:", error)
  );
 }
}
```

✓ Uses subscribe() to **fetch and store the API response**.

Displaying Data in HTML

```
<div *ngFor="let post of posts">
 <h3>{{ post.title }}</h3>
 <p>{{ post.body }}</p>
```

✓ Iterates through **posts using *ngFor**.

CHAPTER 4: TRANSFORMING DATA WITH RXJS OPERATORS

4.1 What are RxJS Operators?

Operators **modify or filter data streams** before passing them to subscribers.

✓ **Common RxJS Operators**:

Operator	Description	Example Use
map()	Transforms data	Convert API response to a specific format
filter()	Filters values	Only show posts with more than 100 likes
debounceTime()	Controls event frequency	Prevents multiple API calls on fast user input
catchError()	Handles errors	Prevents application crashes

4.2 Using map() to Transform Data

The map() operator **modifies API response data**.

```
import { map } from 'rxjs/operators';


this.dataService.getPosts()
```

```
  .pipe(map(posts => posts.map(post => ({ title:
post.title.toUpperCase() }))))

  .subscribe(transformedPosts => console.log(transformedPosts));
```

✓ Converts **post titles to uppercase** before passing data to the subscriber.

4.3 Using filter() to Select Data

The filter() operator **removes unwanted values** from an observable stream.

```
import { filter } from 'rxjs/operators';


this.dataService.getPosts()

  .pipe(filter(post => post.userId === 1))

  .subscribe(filteredPosts => console.log(filteredPosts));
```

✓ Only retrieves **posts created by user ID 1**.

CHAPTER 5: HANDLING ERRORS IN OBSERVABLES

5.1 Using catchError() to Handle API Errors

RxJS provides **error-handling operators** to **catch and manage errors** gracefully.

Example: Handling HTTP Errors

```
import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';
```

```
this.dataService.getPosts()
 .pipe(
  catchError(error => {
   console.error('Error occurred:', error);
   return throwError(() => new Error('Something went wrong'));
  })
 )
 .subscribe();
```

✓ **Logs the error** and prevents application crashes.

Case Study: How an E-Commerce Platform Used RxJS to Improve Performance

Background

An **e-commerce company** faced issues with:

✓ **Slow API responses** delaying product search results.
✓ **Multiple redundant API calls** on search input.
✓ **Uncaught errors crashing the application**.

Challenges

- **Typing in the search bar triggered too many requests**.
- **Users saw outdated results** due to slow updates.
- **Lack of error handling** caused failed API calls.

Solution: Implementing RxJS for API Optimization

The team optimized search requests using:

✓ **debounceTime(500)** – Delayed API calls until **typing stopped for 500ms**.
✓ **distinctUntilChanged()** – Prevented duplicate API calls for the same search term.
✓ **catchError()** – Handled errors without breaking the UI.

```
import { debounceTime, distinctUntilChanged, switchMap, catchError } from 'rxjs/operators';


this.searchInput.valueChanges.pipe(
  debounceTime(500),
  distinctUntilChanged(),
  switchMap(searchTerm => this.dataService.searchProducts(searchTerm)),
  catchError(() => [])
).subscribe(results => this.products = results);
```

Results

🚀 **Reduced API calls by 70%,** improving speed.
🔍 **Faster search experience,** with real-time updates.
⚡ **Better error handling,** preventing app crashes.

By using **RxJS observables,** the company improved **performance, efficiency, and reliability**.

Exercise

1. Create an **Observable** that emits values every 2 seconds.
2. Use .pipe(map()) to convert emitted values to uppercase.

3. Implement **error handling** with catchError().
4. Use debounceTime() to delay API calls in a search field.

Conclusion

In this section, we explored:

✓ **How Observables work in Angular using RxJS**.

✓ **How to use Observables with HTTP services and Angular components**.

✓ **How to transform data streams using RxJS operators**.

✓ **How to handle errors and optimize performance using catchError() and debounceTime()**.

Introduction to State Management in Angular

Chapter 1: Understanding State Management

1.1 What is State Management?

State management refers to the process of handling **application data** across multiple components and views in an Angular application.

✓ Ensures **data consistency** across components.
✓ Improves **scalability** and performance.
✓ Reduces **unnecessary API calls**.

1.2 Why is State Management Important?

In large applications, managing state manually can lead to **inconsistent data, redundant API calls, and complex debugging**.

🚀 **Examples of state in Angular applications:**

- **User authentication status**
- **Cart items in an e-commerce app**
- **Theme selection (dark/light mode)**

Chapter 2: Types of State in Angular Applications

2.1 Local State (Component State)

✓ **Exists only within a single component**.
✓ Used for **temporary UI states** (e.g., toggles, form inputs).

Example: Local State in a Component

```
export class CounterComponent {
 count = 0;

 increment() {
  this.count++;
 }
}
```

```
<p>Count: {{ count }}</p>
<button (click)="increment()">Increment</button>
```

🚀 Best for **small UI-based data** that doesn't need to be shared.

2.2 Shared State (Service-Based State Management)

✓ Used when **multiple components** need access to **the same data**.
✓ Uses **Angular Services** and **RxJS Observables**.

Example: Managing State with a Service

cart.service.ts (Shared Cart State)

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

@Injectable({
 providedIn: 'root'
})
```

```
export class CartService {
 private cartItems = new BehaviorSubject<number>(0);
 cartItems$ = this.cartItems.asObservable();

 addItem() {
  this.cartItems.next(this.cartItems.value + 1);
 }
}
```

🚀 **Why use BehaviorSubject?**

- **Stores the latest value** and allows **real-time updates**.
- **Multiple components** can subscribe to cartItems$.

2.3 Global State (Using Libraries like NgRx & Akita)

✓ Used for **large-scale applications**.
✓ Manages **complex data flows** (e.g., multi-page forms, dashboards).

Libraries for **global state management in Angular**:

Library	Key Features
NgRx	Based on Redux pattern, uses actions & reducers.
Akita	Simpler, event-driven, focuses on observability.
NGXS	Minimal boilerplate, inspired by Vuex.

🚀 **Best for:**
✓ Enterprise applications

✓ Real-time dashboards

✓ Complex form flows

CHAPTER 3: IMPLEMENTING STATE MANAGEMENT USING SERVICES

3.1 Creating a Service for Shared State

Use a **service with RxJS** to manage shared state.

Step 1: Generate the Service

ng g s state

Step 2: Define the State in state.service.ts

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

@Injectable({
 providedIn: 'root'
})
export class StateService {
 private username = new BehaviorSubject<string>('Guest');
 username$ = this.username.asObservable();

 updateUsername(newName: string) {
  this.username.next(newName);
 }
}
```

🚀 Uses **BehaviorSubject** to store the username and allow updates.

3.2 Injecting State Service into Components

Modify app.component.ts to use the state:

```
import { Component } from '@angular/core';
import { StateService } from './state.service';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html'
})
export class AppComponent {
 username: string = '';

 constructor(private stateService: StateService) {
  this.stateService.username$.subscribe(name => this.username = name);
 }

 changeName() {
  this.stateService.updateUsername('Alice');
 }
}
```

Modify app.component.html:

Welcome, {{ username }}!

<button (click)="changeName()">Change Name</button>

🚀 **Clicking the button updates the username across the application.**

CHAPTER 4: INTRODUCTION TO NGRX FOR GLOBAL STATE MANAGEMENT

4.1 What is NgRx?

NgRx is a **state management library for Angular** based on the **Redux pattern**.

✓ Uses **Actions, Reducers, and Effects** to manage state.
✓ Ensures **predictable state updates**.
✓ Improves **debugging with Redux DevTools**.

4.2 Setting Up NgRx in Angular

Step 1: Install NgRx

ng add @ngrx/store

ng add @ngrx/effects

Step 2: Define the State Model (state.model.ts)

export interface AppState {

username: string;

}

Step 3: Create an Action (state.actions.ts)

```
import { createAction, props } from '@ngrx/store';

export const setUsername = createAction(
  '[User] Set Username',
  props<{ username: string }>()
);
```

✓ Defines an action to update the username.

Step 4: Create a Reducer (state.reducer.ts)

```
import { createReducer, on } from '@ngrx/store';
import { setUsername } from './state.actions';

export const initialState: AppState = {
  username: 'Guest'
};

const _userReducer = createReducer(
  initialState,
  on(setUsername, (state, { username }) => ({ ...state, username }))
);

export function userReducer(state: any, action: any) {
  return _userReducer(state, action);
```

```
}
```

✓ **Reducers define how the state changes when an action is dispatched**.

4.3 Using NgRx Store in Components

Inject **NgRx Store** into the component:

```
import { Component } from '@angular/core';
import { Store } from '@ngrx/store';
import { setUsername } from './state.actions';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html'
})
export class AppComponent {
 username$ = this.store.select(state => state.username);

 constructor(private store: Store<{ username: string }>) {}

 changeName() {
  this.store.dispatch(setUsername({ username: 'Alice' }));
 }
}
```

Modify app.component.html:

Welcome, {{ username$ | async }}!

<button (click)="changeName()">Change Name</button>

🚀 **NgRx efficiently manages and updates the global state.**

Case Study: How a Social Media App Used NgRx for State Management

Background

A **social media startup** needed real-time updates for:

✓ **User profiles**
✓ **Notifications**
✓ **Messages**

Challenges

- **Inconsistent state** across pages.
- **Duplicate API calls** slowing performance.
- **Manual state management** causing complex logic.

Solution: Implementing NgRx

✓ **Centralized user state** for profile updates.
✓ **Used NgRx Effects** to handle API calls.
✓ **Reduced API calls by 60%,** improving performance.

Results

🚀 **Instant profile updates** across the app.
⚡ **Faster load times** due to caching.
🔍 **Easier debugging** with Redux DevTools.

By adopting **NgRx for state management**, the app improved **performance, scalability, and maintainability**.

Exercise

1. Create an Angular **state service** that stores a **theme (light/dark mode)**.
2. Create a button that **toggles the theme** and updates all components.
3. Implement a **global state management solution using NgRx** for user authentication.

Conclusion

In this section, we explored:

✓ **Different types of state management in Angular** (Local, Shared, Global).

✓ **How to manage shared state using Angular Services**.

✓ **How to implement global state management using NgRx**.

USING RXJS FOR STATE MANAGEMENT IN ANGULAR

CHAPTER 1: INTRODUCTION TO RXJS AND STATE MANAGEMENT

1.1 What is RxJS?

RxJS (Reactive Extensions for JavaScript) is a **powerful library** for handling **asynchronous data streams** in Angular. It is based on **Reactive Programming** principles and provides:

✓ **Observable-based state management** – Handles real-time data updates efficiently.
✓ **Operators for transformation** – Allows filtering, mapping, and reducing data streams.
✓ **Improved performance** – Reduces unnecessary computations and UI updates.
✓ **Better error handling** – Manages failures without blocking the application.

1.2 Why Use RxJS for State Management?

State management in Angular applications ensures **data consistency** across multiple components. RxJS helps:

✓ Manage **global and component-level state** efficiently.
✓ Reduce **unnecessary API calls** by sharing cached data.
✓ Implement **real-time updates** using Observables.

CHAPTER 2: UNDERSTANDING OBSERVABLES AND SUBJECTS

2.1 What is an Observable?

An **Observable** is a stream of data that can be **observed over time**. It is used to **handle asynchronous operations** such as:

✓ HTTP requests
✓ User input events
✓ WebSocket communication

Example: Creating an Observable

```
import { Observable } from 'rxjs';


const observable = new Observable(subscriber => {
  subscriber.next("Hello, RxJS!");
  setTimeout(() => subscriber.next("Data Received"), 2000);
});


observable.subscribe(value => console.log(value));
```

✓ Emits **"Hello, RxJS!"** immediately and **"Data Received"** after 2 seconds.

2.2 What is a Subject?

A **Subject** is a special type of Observable that allows **multicasting** (i.e., multiple subscribers receive the same data).

✓ **Acts as both an Observable and Observer**.
✓ **Shares state across multiple components**.

Example: Creating a Subject for State Management

```
import { Subject } from 'rxjs';
```

```
const state = new Subject<string>();

state.subscribe(value => console.log("Subscriber 1:", value));
state.subscribe(value => console.log("Subscriber 2:", value));

state.next("New State Update!");
```

✓ Both subscribers receive **"New State Update!"**.

CHAPTER 3: USING BEHAVIORSUBJECT FOR STATE MANAGEMENT

3.1 Why Use BehaviorSubject?

BehaviorSubject is a special type of Subject that:

✓ **Stores the latest value** and emits it to new subscribers.
✓ **Maintains state across the application**.

3.2 Example: Using BehaviorSubject for Global State

Step 1: Create a State Management Service

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

@Injectable({
  providedIn: 'root'
})
export class StateService {
```

```
  private state = new BehaviorSubject<string>("Initial State");
  currentState = this.state.asObservable(); // Expose state as observable

  updateState(newState: string) {
    this.state.next(newState);
  }
}
```

✓ **Stores and manages application-wide state.**

Step 2: Inject and Use State in Components

```
import { Component } from '@angular/core';
import { StateService } from './state.service';

@Component({
  selector: 'app-state-demo',
  template: `
    <p>Current State: {{ state }}</p>
    <button (click)="changeState()">Update State</button>
  `
})
export class StateDemoComponent {
  state: string = "";
```

```
 constructor(private stateService: StateService) {
  this.stateService.currentState.subscribe(value => this.state = value);
 }

 changeState() {
  this.stateService.updateState("Updated State!");
 }
}
```

✓ The **component updates in real-time** whenever the state changes.

CHAPTER 4: USING RXJS OPERATORS FOR STATE TRANSFORMATION

4.1 Common RxJS Operators for State Management

Operator	Purpose
map()	Transforms emitted values.
filter()	Filters out unwanted values.
distinctUntilChanged()	Prevents duplicate state updates.
debounceTime()	Limits rapid state changes.

4.2 Example: Filtering State Updates

```
import { BehaviorSubject } from 'rxjs';
import { filter } from 'rxjs/operators';
```

```
const state = new BehaviorSubject<number>(0);


state.pipe(
  filter(value => value % 2 === 0) // Only even numbers
).subscribe(value => console.log("Filtered State:", value));


state.next(1); // No output
state.next(2); // Output: Filtered State: 2
```

✓ Only **even numbers** are processed.

CHAPTER 5: HANDLING ASYNCHRONOUS STATE WITH RXJS

5.1 Using switchMap() for API Calls

Instead of making multiple **API requests,** switchMap() cancels previous calls and processes only the latest request.

Example: Searching with API Requests

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { debounceTime, switchMap } from 'rxjs/operators';
import { Subject } from 'rxjs';


@Component({
  selector: 'app-search',
  template: `<input (input)="search($event)" placeholder="Search">`
```

```
})
export class SearchComponent {
  private searchSubject = new Subject<string>();

  constructor(private http: HttpClient) {
    this.searchSubject.pipe(
      debounceTime(300), // Wait 300ms before firing request
      switchMap(query =>
this.http.get(`https://api.example.com/search?q=${query}`))
    ).subscribe(results => console.log(results));
  }

  search(event: any) {
    this.searchSubject.next(event.target.value);
  }
}
```

✓ Prevents unnecessary API calls **while typing**.

CHAPTER 6: BEST PRACTICES FOR RXJS-BASED STATE MANAGEMENT

✓ Use BehaviorSubject for storing global state.
✓ Use distinctUntilChanged() to avoid unnecessary updates.
✓ Use debounceTime() for search or input processing.
✓ Use switchMap() to optimize API requests.

✓ **Unsubscribe from Observables in ngOnDestroy to prevent memory leaks**.

Case Study: How a FinTech Company Used RxJS for State Management

Background

A **financial dashboard** needed:

✓ **Real-time stock price updates**.
✓ **Efficient API request handling**.
✓ **Reduced duplicate API calls for the same stock data**.

Challenges

- **Frequent API polling** caused **server overload**.
- **Duplicate API calls** slowed down performance.
- **Stock data updates were lagging**.

Solution: Implementing RxJS for State Management

✓ Used BehaviorSubject to **store the latest stock data**.
✓ Used distinctUntilChanged() to **prevent redundant updates**.
✓ Used switchMap() to **handle API polling efficiently**.

Results

🚀 **Real-time updates without lag**.
🔍 **Reduced API calls by 50%,** improving performance.
⚡ **Optimized data handling,** leading to better user experience.

By using **RxJS for state management,** the dashboard **delivered fast, responsive financial data**.

Exercise

1. Create a **state management service** using BehaviorSubject.
2. Inject the service into a component and **update state dynamically**.
3. Use debounceTime() to optimize **real-time search input handling**.
4. Implement switchMap() to **handle API calls efficiently**.

Conclusion

In this section, we explored:

✓ **How RxJS helps manage state efficiently in Angular**.

✓ **How to use Observables, Subjects, and BehaviorSubjects for state**.

✓ **How to optimize state updates using RxJS operators**.

✓ **How to prevent unnecessary API calls and UI updates**.

SETTING UP ROUTING & LAZY LOADING IN ANGULAR

CHAPTER 1: INTRODUCTION TO ROUTING IN ANGULAR

1.1 What is Routing in Angular?

Routing in Angular allows users to navigate between different **views** or **pages** of an application **without reloading the page**. It is an essential feature for **Single Page Applications (SPAs)**.

✓ **Handles navigation between components** dynamically.
✓ **Uses URL-based routing** to change views without full-page reloads.
✓ **Supports route parameters for dynamic content**.
✓ **Enables lazy loading to improve performance**.

1.2 Why Use Routing?

✓ **Enhances user experience** – No need to reload the entire application.
✓ **Organizes components efficiently** – Keeps application structure clean.
✓ **Improves scalability** – Easily add new views/pages without major changes.

CHAPTER 2: SETTING UP ROUTING IN ANGULAR

2.1 Importing RouterModule and Defining Routes

To enable routing in Angular, we must **import and configure RouterModule** in app.module.ts.

Step 1: Import RouterModule in app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
 { path: '', component: HomeComponent }, // Default route
 { path: 'about', component: AboutComponent } // Navigates to About page
];

@NgModule({
 declarations: [AppComponent, HomeComponent, AboutComponent],
 imports: [BrowserModule, RouterModule.forRoot(routes)], // Register routes
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

✓ **RouterModule.forRoot(routes)** registers routes at the **root level**.
✓ **Declares two routes (/ for home and /about for about page)**.

2.2 Navigating Between Pages Using RouterLink

Modify app.component.html to add **navigation links** using routerLink:

```
<nav>
  <a routerLink="/">Home</a> |
  <a routerLink="/about">About</a>
</nav>

<!-- Renders the selected component -->
<router-outlet></router-outlet>
```

✓ **routerLink** defines the navigation paths.
✓ **<router-outlet>** renders the active component dynamically.

2.3 Redirecting Routes & Handling Wildcards

✓ **Redirect users from /home to /**:

```
const routes: Routes = [
  { path: 'home', redirectTo: '', pathMatch: 'full' }
];
```

✓ **Handling Invalid Routes (404 Not Found page)**:

```
const routes: Routes = [
  { path: '**', component: NotFoundComponent } // Wildcard for invalid routes
```

];

✓ This ensures that an error page is displayed when an **unknown URL is entered**.

CHAPTER 3: PASSING ROUTE PARAMETERS

3.1 Using Route Parameters for Dynamic Pages

Route parameters allow us to pass **dynamic data** via URLs.

Step 1: Define a Dynamic Route in app-routing.module.ts

```
const routes: Routes = [
  { path: 'user/:id', component: UserComponent }
];
```

✓ :id is a **route parameter** that changes dynamically.

Step 2: Access Route Parameters in user.component.ts

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

@Component({
  selector: 'app-user',
  templateUrl: './user.component.html'
})
export class UserComponent implements OnInit {
  userId: string = '';
```

```
 constructor(private route: ActivatedRoute) {}


 ngOnInit() {
  this.userId = this.route.snapshot.paramMap.get('id') || '';
 }
}
```

✓ Extracts **:id parameter** from the URL dynamically.

Step 3: Navigate to a Dynamic Route

Modify app.component.html:

```
<a [routerLink]="['/user', 101]">View User 101</a>
```

✓ Navigates to /user/101, dynamically updating the component.

CHAPTER 4: IMPLEMENTING LAZY LOADING IN ANGULAR

4.1 What is Lazy Loading?

Lazy loading **improves performance** by **loading modules only when needed,** instead of loading the entire application at once.

✓ **Reduces initial load time** – Faster page rendering.
✓ **Loads only required modules** – Optimizes memory usage.
✓ **Best for large applications** – Helps scale complex projects efficiently.

4.2 Setting Up Lazy Loading in Angular

Step 1: Generate a Feature Module

ng generate module products --route products --module app.module

✓ Creates products.module.ts and **configures lazy loading automatically**.

Step 2: Modify app-routing.module.ts for Lazy Loading

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

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

✓ loadChildren dynamically loads the ProductsModule only when the user visits /products.

Step 3: Define Child Routes in products-routing.module.ts

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

```
import { RouterModule, Routes } from '@angular/router';

import { ProductListComponent } from './product-list/product-list.component';


const routes: Routes = [

 { path: '', component: ProductListComponent } // Default route for /products

];


@NgModule({

 imports: [RouterModule.forChild(routes)],

 exports: [RouterModule]

})

export class ProductsRoutingModule { }
```

✓ Defines **child routes** inside the lazy-loaded module.

CHAPTER 5: TESTING LAZY LOADING & PERFORMANCE BENEFITS

5.1 Verifying Lazy Loading in the Network Tab

1. Open the browser's **Developer Tools (F12) > Network Tab**.
2. Navigate to **localhost:4200** (home page).
3. Observe that **the products module is not loaded** initially.
4. Click on the **Products page** link.
5. The products module **loads dynamically** in the Network tab.

✓ Confirms **lazy loading is working** and reducing **initial load time**.

Case Study: How a FinTech Company Used Lazy Loading to Improve Performance

Background

A **FinTech company** was developing an **investment dashboard** that:

✓ Included multiple modules (dashboard, transactions, reports, profile).
✓ Needed **fast page load times** for better user experience.
✓ Had performance issues due to **large initial bundle size**.

Challenges

- The entire application loaded at once, causing **slow initial rendering**.
- **Unnecessary modules** were loaded even if users didn't access them.
- **API calls were delayed,** affecting real-time updates.

Solution: Implementing Lazy Loading in Angular

✓ Converted **feature modules (TransactionsModule, ReportsModule)** into **lazy-loaded modules**.
✓ Used loadChildren to **load them only when needed**.
✓ **Optimized routing structure,** reducing **unnecessary dependencies**.

Results

🚀 **60% faster initial load times,** improving user engagement.
⚡ **Reduced memory usage,** since only needed modules were

loaded.

📈 **Better API performance,** as resources were efficiently managed.

By implementing **lazy loading,** the company **significantly improved app performance** and **user experience**.

Exercise

1. Set up **routing in an Angular project** with Home and About pages.
2. Implement **route parameters** to display dynamic user details.
3. Create a **feature module (OrdersModule)** and configure lazy loading.
4. Verify **lazy loading performance** using the browser's Network tab.

Conclusion

In this section, we explored:

✓ **How to set up routing in Angular using RouterModule**.

✓ **How to use route parameters for dynamic navigation**.

✓ **How to implement lazy loading for improved performance**.

IMPLEMENTING GUARDS AND ROUTE RESOLVERS IN ANGULAR

CHAPTER 1: INTRODUCTION TO GUARDS AND ROUTE RESOLVERS IN ANGULAR

1.1 Understanding Angular Routing

Angular provides **routing** to navigate between different views (components) of an application. However, some routes need to be **protected** or **preloaded with data** before rendering.

✓ **Guards** – Restrict access to specific routes based on user authentication, authorization, or other conditions.
✓ **Resolvers** – Fetch necessary data before loading a component, ensuring smooth user experience.

CHAPTER 2: IMPLEMENTING ROUTE GUARDS IN ANGULAR

2.1 What are Route Guards?

Route Guards **control access** to routes based on specific conditions. Angular provides **five types of route guards**:

Guard Type	Description	Example Use Case
CanActivate	Prevents unauthorized access to a route.	Restrict access to a dashboard for logged-in users.
CanDeactivate	Prevents users from leaving a page.	Show a confirmation dialog when leaving an unsaved form.

CanLoad	Prevents loading of a module unless a condition is met.	Restrict loading of an admin module unless the user is an admin.
CanActivateChild	Applies CanActivate checks to child routes.	Prevent access to child pages if the parent page is restricted.
Resolve	Preloads data before activating a route.	Fetch user details before loading a profile page.

✓ **Guards enhance security** and ensure **route accessibility based on conditions**.

2.2 Creating an Authentication Guard (CanActivate)

Step 1: Generate a Guard Using Angular CLI

ng generate guard guards/auth

✓ Creates auth.guard.ts inside guards/ folder.

Step 2: Implement the CanActivate Guard

Modify auth.guard.ts:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';


@Injectable({
  providedIn: 'root'
})
```

```
export class AuthGuard implements CanActivate {


 constructor(private router: Router) {}


 canActivate(): boolean {
  const isLoggedIn = !!localStorage.getItem('userToken');
  if (!isLoggedIn) {
   this.router.navigate(['/login']);
   return false;
  }
  return true;
 }
}
```

✓ **Checks if the user is logged in** (by verifying userToken).
✓ **Redirects unauthorized users** to the login page.

2.3 Applying the Guard to Routes

Modify app-routing.module.ts:

```
import { AuthGuard } from './guards/auth.guard';


const routes: Routes = [
 { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] },
```

```
  { path: 'login', component: LoginComponent }
];

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

✓ Protects the **dashboard route,** allowing access **only to logged-in users**.

CHAPTER 3: IMPLEMENTING CANDEACTIVATE GUARDS

3.1 Preventing Users from Leaving a Page

The CanDeactivate guard **asks users for confirmation** before leaving a route.

Step 1: Generate a CanDeactivate Guard

```
ng generate guard guards/form
```

Step 2: Implement the CanDeactivate Guard

Modify form.guard.ts:

```
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { FormComponent } from
'../components/form/form.component';
```

```
@Injectable({
  providedIn: 'root'
})
export class FormGuard implements CanDeactivate<FormComponent> {
  canDeactivate(component: FormComponent): boolean {
    return component.isFormSaved || confirm("You have unsaved changes. Do you really want to leave?");
  }
}
```

✓ **Checks if the form is saved** before leaving.
✓ **Shows a confirmation popup** if there are unsaved changes.

3.2 Applying the CanDeactivate Guard to a Route

Modify app-routing.module.ts:

```
import { FormGuard } from './guards/form.guard';

const routes: Routes = [
  { path: 'edit-profile', component: FormComponent, canDeactivate: [FormGuard] }
];
```

✓ Ensures users **don't lose data accidentally** when navigating away.

CHAPTER 4: IMPLEMENTING CANLOAD GUARDS FOR LAZY LOADING

4.1 Preventing Unauthorized Module Loading

The CanLoad guard **restricts loading of an entire module** unless conditions are met.

Step 1: Create a CanLoad Guard

ng generate guard guards/admin

Step 2: Implement the CanLoad Guard

Modify admin.guard.ts:

```
import { Injectable } from '@angular/core';
import { CanLoad, Route, UrlSegment, Router } from '@angular/router';

@Injectable({
 providedIn: 'root'
})
export class AdminGuard implements CanLoad {

 constructor(private router: Router) {}

 canLoad(route: Route, segments: UrlSegment[]): boolean {
  const isAdmin = localStorage.getItem('userRole') === 'admin';
  if (!isAdmin) {
```

```
    this.router.navigate(['/']);
    return false;
  }
  return true;
 }
}
```

✓ **Blocks unauthorized users** from loading the admin module.

4.2 Applying CanLoad to a Lazy-Loaded Module

Modify app-routing.module.ts:

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

✓ **Prevents non-admin users** from loading admin-related code.

CHAPTER 5: IMPLEMENTING ROUTE RESOLVERS IN ANGULAR

5.1 What is a Route Resolver?

A **Route Resolver** fetches data **before** navigating to a route, preventing UI flickers caused by slow API calls.

✓ **Preloads data** for a smooth user experience.
✓ **Avoids empty screens** while data loads.
✓ **Ensures required data is available** before rendering a component.

5.2 Creating a Route Resolver

Step 1: Generate a Resolver

ng generate resolver resolvers/user

Step 2: Implement the Resolver

Modify user.resolver.ts:

```
import { Injectable } from '@angular/core';
import { Resolve } from '@angular/router';
import { Observable } from 'rxjs';
import { UserService } from '../services/user.service';

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

  constructor(private userService: UserService) {}
```

```
 resolve(): Observable<any> {
  return this.userService.getUserDetails();
 }
}
```

✓ **Fetches user data** before navigating to a profile page.

5.3 Applying the Resolver to a Route

Modify app-routing.module.ts:

```
import { UserResolver } from './resolvers/user.resolver';

const routes: Routes = [
 { path: 'profile', component: ProfileComponent, resolve: { user: UserResolver } }
];
```

✓ Ensures that **profile data is available before loading the component**.

Case Study: How a Banking App Used Guards and Resolvers for Security and Performance

Background

A banking application needed to:

✓ **Restrict access to financial dashboards** for authorized users.
✓ **Prevent users from leaving transaction pages with unsaved data**.
✓ **Load user account data before rendering account details**.

Challenges

- Unauthorized users **accessing restricted pages**.
- **Data loss issues** when navigating away from transaction forms.
- **Slow API calls** causing UI flickers.

Solution: Implementing Guards and Resolvers

The team implemented:

✓ **CanActivate guards** to protect financial pages.
✓ **CanDeactivate guards** to prevent unsaved data loss.
✓ **Resolvers to preload account details** before navigation.

Results

🚀 **Improved security,** ensuring only authorized users accessed financial data.
⚡ **No more lost transaction data,** reducing customer complaints.
🔄 **Seamless page transitions,** improving user experience.

By using **Angular Guards and Resolvers,** the app achieved **better security, stability, and performance**.

Exercise

1. Create a CanActivate guard to **restrict access** to an admin page.

2. Implement a CanDeactivate guard to **warn users** before leaving an unsaved form.
3. Build a **Route Resolver** that loads user data before opening the dashboard.
4. Apply a CanLoad guard to **prevent unauthorized module loading**.

Conclusion

In this section, we explored:

✓ **How to use Guards (CanActivate, CanDeactivate, CanLoad) to protect routes**.

✓ **How to implement Resolvers to preload data before navigation**.

✓ **How Guards and Resolvers improve security, data integrity, and performance**.

ASSIGNMENT:

BUILD AN ANGULAR APPLICATION WITH DYNAMIC ROUTING AND API CALLS

SOLUTION GUIDE: BUILDING AN ANGULAR APPLICATION WITH DYNAMIC ROUTING AND API CALLS

Step 1: Set Up the Angular Project

1.1 Create a New Angular Project

Run the following command to create a new Angular application:

ng new angular-routing-app

cd angular-routing-app

1.2 Install Required Modules

Since we will be making **API calls**, install HttpClientModule:

npm install --save bootstrap

Modify angular.json to include Bootstrap:

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

Step 2: Configure Routing in Angular

2.1 Generate Components for Navigation

We will create three components:

ng generate component home

ng generate component users

ng generate component user-details

2.2 Set Up Routing in app-routing.module.ts

Modify app-routing.module.ts to define routes:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { UsersComponent } from './users/users.component';
import { UserDetailsComponent } from './user-details/user-details.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent },
  { path: 'users/:id', component: UserDetailsComponent } // Dynamic route
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule { }
```

✓ :id in /users/:id allows **dynamic routing for user details**.

2.3 Add Navigation Links to app.component.html

Modify app.component.html to include navigation:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <ul class="navbar-nav">
    <li class="nav-item"><a class="nav-link"
routerLink="/">Home</a></li>
    <li class="nav-item"><a class="nav-link"
routerLink="/users">Users</a></li>
  </ul>
</nav>

<div class="container mt-4">
  <router-outlet></router-outlet>
</div>
```

✓ routerLink enables **navigation between pages**.
✓ router-outlet dynamically loads the **active route's component**.

Step 3: Fetching API Data Using an Angular Service

3.1 Generate a User Service

Run the following command:

ng generate service user

Modify user.service.ts to fetch data from an API:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
 providedIn: 'root'
})
export class UserService {
 private apiUrl = 'https://jsonplaceholder.typicode.com/users';

 constructor(private http: HttpClient) {}

 getUsers(): Observable<any> {
  return this.http.get(this.apiUrl);
 }

 getUserById(id: number): Observable<any> {
  return this.http.get(`${this.apiUrl}/${id}`);
 }
```

```
}
```

✓ **Uses HttpClient to fetch user data**.

✓ Provides **methods to fetch all users and a specific user**.

Step 4: Displaying Users with API Calls

Modify users.component.ts to fetch and display users:

```
import { Component, OnInit } from '@angular/core';
import { UserService } from '../user.service';

@Component({
 selector: 'app-users',
 templateUrl: './users.component.html'
})
export class UsersComponent implements OnInit {
 users: any[] = [];

 constructor(private userService: UserService) {}

 ngOnInit() {
  this.userService.getUsers().subscribe(data => {
   this.users = data;
  });
 }
```

```
}
```

Modify users.component.html to display users:

```
<h2>Users List</h2>
<ul class="list-group">
  <li class="list-group-item" *ngFor="let user of users">
    <a [routerLink]="['/users', user.id]">{{ user.name }}</a>
  </li>
</ul>
```

✓ Fetches users **on component initialization**.
✓ Displays **clickable user names** linked to **user details pages**.

Step 5: Displaying User Details from Dynamic Routes

Modify user-details.component.ts to fetch **user details based on route parameters**:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { UserService } from '../user.service';

@Component({
  selector: 'app-user-details',
  templateUrl: './user-details.component.html'
})
export class UserDetailsComponent implements OnInit {
```

```
  user: any;


  constructor(private route: ActivatedRoute, private userService:
UserService) {}


  ngOnInit() {
    const userId = this.route.snapshot.params['id'];
    this.userService.getUserById(userId).subscribe(data => {
      this.user = data;
    });
  }
}
```

Modify user-details.component.html to display user details:

```
<div *ngIf="user">
  <h2>{{ user.name }}</h2>
  <p>Email: {{ user.email }}</p>
  <p>Phone: {{ user.phone }}</p>
  <a routerLink="/users" class="btn btn-primary">Back to Users</a>
</div>
```

✓ Retrieves **user details** based on **dynamic route parameters**.

Step 6: Run and Test the Application

Start the development server:

ng serve

Test the Features

✅ **Navigate to http://localhost:4200/** → Home Page.
✅ **Click "Users"** → Displays the user list.
✅ **Click on a user** → Displays detailed information.
✅ **Click "Back to Users"** → Returns to the users list.

Case Study: How a Blogging Platform Used Dynamic Routing and API Calls

Background

A blogging platform needed:
✓ A **list of blog posts** fetched from an API.
✓ **Dynamic routes** for each blog post.
✓ **Navigation between posts** without reloading the page.

Challenges

- **Slow page reloads** due to static routing.
- **Redundant API calls** causing performance issues.
- **Inconsistent data updates** across components.

Solution: Implementing Dynamic Routing & API Calls

✓ Used **HttpClientModule** to fetch blog posts.
✓ Implemented **dynamic routing with :id parameters**.
✓ **Used BehaviorSubject** to cache API data and reduce duplicate requests.

Results

🚀 **Faster page loads** due to dynamic routing.
🔍 **Better user experience** with seamless navigation.
📈 **Reduced API calls by 40%,** improving performance.

By implementing **Angular routing and API services,** the platform **improved efficiency and user engagement**.

Exercise

1. Modify the **Users component** to allow searching for users.
2. Add a **loading spinner** while fetching API data.
3. Implement an **error message** if the API fails.
4. Add a **page not found (404) component** for invalid routes.

Conclusion

In this assignment, we:
✓ Implemented **dynamic routing** using Angular Router.
✓ Created an **Angular service** to fetch **API data**.
✓ Built a **user list with dynamic details pages**.
✓ Used **Bootstrap for styling**.