Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

DEPLOYMENT, SECURITY & CAPSTONE PROJECT (WEEKS 11-12)

PREPARING ANGULAR APPS FOR PRODUCTION

CHAPTER 1: INTRODUCTION TO PRODUCTION OPTIMIZATION IN ANGULAR

1.1 Why Optimize Angular Apps for Production?

When developing an Angular application, it runs in **development mode** with debugging features enabled. Before deploying an Angular app to production, we must **optimize performance, security, and efficiency**.

✓ **Faster load times** – Reduces bundle size and improves performance.
✓ **Enhanced security** – Removes unnecessary debugging tools and source maps.
✓ **Efficient memory usage** – Ensures smooth execution on different devices.
✓ **Better SEO & user experience** – Improves website rankings and engagement.

1.2 Key Optimization Areas

Area	Optimization Techniques
Code Optimization	Minification, tree-shaking, lazy loading
Performance Optimization	Ahead-of-Time (AOT) compilation, caching, lazy loading
Security Enhancements	Disabling debugging, enabling Content Security Policy (CSP)
Deployment Best Practices	Using CDN, hosting on Firebase, AWS, or Netlify

---

CHAPTER 2: BUILDING AN OPTIMIZED ANGULAR APPLICATION

2.1 Enabling Production Mode in Angular

By default, Angular runs in **development mode**, which includes extra debugging tools. To enable **production mode**, use:

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


if (environment.production) {
  enableProdMode();
}
```

✓ **Removes unnecessary debugging features**.
✓ **Improves rendering performance**.

2.2 Creating a Production Build

To create an optimized production-ready version of your Angular app, run:

ng build --configuration=production

✓ **Optimizations included in the production build:**

- **Minification** – Reduces file size by removing unnecessary spaces and comments.
- **Tree shaking** – Removes unused code.
- **Ahead-of-Time (AOT) compilation** – Precompiles Angular templates to improve performance.

📌 **To check the output, navigate to:**

cd dist/your-app-name

CHAPTER 3: REDUCING BUNDLE SIZE

3.1 Enabling Tree Shaking

Tree shaking removes **unused imports and dead code,** reducing the final bundle size.

✓ **Ensure unused modules aren't imported** in app.module.ts.
✓ Use **dynamic imports** for lazy loading modules.

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

✓ **Removes unnecessary JavaScript files** from the final build.

3.2 Enabling Compression (Gzip & Brotli)

Compressed files **load faster,** reducing data transfer costs.

✓ **Enable compression in the server** (Apache, Nginx, or Firebase Hosting).

For **Nginx,** add this to nginx.conf:

gzip on;

gzip_types text/plain text/css application/json application/javascript;

gzip_min_length 1000;

CHAPTER 4: IMPROVING PERFORMANCE WITH AOT COMPILATION

4.1 What is AOT (Ahead-of-Time) Compilation?

By default, Angular uses **Just-in-Time (JIT)** compilation, which compiles templates **at runtime,** increasing load time. **AOT** compiles templates **before deployment,** improving performance.

✓ **Faster rendering** – No need to compile templates in the browser.
✓ **Smaller bundle size** – Reduces runtime overhead.
✓ **Better security** – Prevents injection attacks.

AOT is enabled by default in production builds. To manually enable it:

ng build --aot

CHAPTER 5: LAZY LOADING FOR FASTER PAGE LOADS

5.1 What is Lazy Loading?

Lazy loading **delays loading of feature modules** until they are needed, reducing the initial load time.

✓ **Improves first-page load time**.
✓ **Loads only necessary components** when required.

5.2 Implementing Lazy Loading in Angular

Modify app-routing.module.ts:

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

✓ This ensures **AdminModule is loaded only when needed**.

CHAPTER 6: SECURING ANGULAR APPS FOR PRODUCTION

6.1 Disabling Debugging Tools

To prevent attackers from accessing debugging tools, disable **Angular DevTools**:

```
ng build --configuration=production --source-map=false
```

✓ Prevents **reverse engineering** of code.

6.2 Enforcing Content Security Policy (CSP)

Content Security Policy (CSP) prevents **XSS (Cross-Site Scripting) attacks**.

Modify **server headers** to enforce CSP:

add_header Content-Security-Policy "default-src 'self'; script-src 'self'; style-src 'self';";

✓ **Blocks unauthorized script execution,** improving security.

CHAPTER 7: DEPLOYING ANGULAR APPLICATIONS

7.1 Best Hosting Services for Angular Apps

Hosting Service	Features
Firebase Hosting	Fast CDN, free SSL, built-in authentication
Netlify	CI/CD, auto-deployment from Git
AWS S3 + CloudFront	High scalability, low latency
Vercel	Best for serverless deployments

7.2 Deploying to Firebase

Install Firebase tools:

npm install -g firebase-tools

Login and initialize Firebase:

firebase login

firebase init

Deploy the Angular app:

ng build --configuration=production

firebase deploy

✓ **Automatically hosts and delivers Angular apps via CDN**.

Case Study: How a SaaS Company Optimized an Angular App for Production

Background

A **SaaS company** providing real-time analytics needed:
✓ **Faster load times** for large-scale data visualization.
✓ **Efficient state management** without lag.
✓ **Optimized performance for mobile users**.

Challenges

- **Slow performance** due to large JavaScript bundles.
- **High server costs** due to redundant API calls.
- **Security vulnerabilities** with debugging enabled in production.

Solution: Production Optimization Strategies

✓ **Enabled lazy loading** for analytics and reports.
✓ **Compressed assets using Gzip** for faster page loads.
✓ **Implemented AOT compilation** for faster rendering.
✓ **Deployed using Firebase Hosting** with a global CDN.

Results

🚀 **Reduced page load time by 50%**.
📉 **Decreased server costs by 40%**.
🔍 **Improved app security** with CSP enforcement.

By implementing **best practices for production,** the company achieved **high performance and cost efficiency**.

Exercise

1. Enable **AOT compilation** and analyze performance improvements.
2. Implement **lazy loading** for feature modules.
3. Minify and compress assets using **Gzip or Brotli**.
4. Deploy the optimized build to **Firebase, Netlify, or AWS**.

Conclusion

✓ **Preparing Angular apps for production improves performance, security, and efficiency**.
✓ **Lazy loading and AOT compilation reduce bundle size and speed up rendering**.
✓ **Deploying on cloud platforms optimizes global content delivery**.

DEPLOYING ANGULAR APPLICATIONS TO FIREBASE, AWS, AND NETLIFY

CHAPTER 1: INTRODUCTION TO ANGULAR DEPLOYMENT

1.1 Why Deploy an Angular Application?

After developing an Angular application, deploying it makes it accessible to users over the internet. **Key benefits of deployment include**:

✓ **Sharing your application globally**.
✓ **Optimizing performance using cloud hosting**.
✓ **Enabling scalability for real-world traffic**.
✓ **Automating deployment with CI/CD for seamless updates**.

1.2 Common Hosting Options for Angular Applications

Hosting Platform	Features
Firebase	Free hosting, built-in CI/CD, global CDN, and real-time database.
AWS S3 + CloudFront	Scalable cloud hosting with caching and security.
Netlify	Simplified deployment with continuous integration support.

CHAPTER 2: DEPLOYING AN ANGULAR APPLICATION TO FIREBASE

2.1 What is Firebase Hosting?

Firebase Hosting is **Google's static web hosting service**, optimized for:

✓ **Fast content delivery with a global CDN**.
✓ **Free SSL certificates** for secure HTTPS hosting.
✓ **Built-in version control and rollback features**.

2.2 Setting Up Firebase Hosting

Step 1: Install Firebase CLI

npm install -g firebase-tools

✓ Installs Firebase CLI globally.

Step 2: Login to Firebase

firebase login

✓ Logs into Firebase using your Google account.

Step 3: Initialize Firebase in Your Angular Project

Navigate to your Angular project directory and run:

firebase init

✓ Select **Hosting** and choose your Firebase project.
✓ Set the **public directory** to dist/angular-app.
✓ Configure as a **single-page app (SPA)** by selecting Yes when prompted.

Step 4: Build Your Angular App

ng build --configuration=production

✓ Generates the production build inside the dist/ folder.

Step 5: Deploy to Firebase

firebase deploy

✓ Deploys the application and provides a **live URL**.

Chapter 3: Deploying an Angular Application to AWS (S3 + CloudFront)

3.1 Why Use AWS S3 and CloudFront?

✓ **S3** – Stores and serves static files.
✓ **CloudFront** – Distributes content globally using a CDN.
✓ **Scalable, secure, and cost-effective** solution.

3.2 Deploying to AWS S3

Step 1: Create an S3 Bucket

1. Go to **AWS Console** → **S3**.
2. Click **Create Bucket** and name it (e.g., my-angular-app).
3. Set **Block Public Access** → **Uncheck** (to make it public).
4. Enable **Static Website Hosting**.

Step 2: Build the Angular Application

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

✓ Generates a dist/ folder with production files.

Step 3: Upload Files to S3

1. Navigate to the S3 bucket.
2. Click **Upload** and select all files from dist/angular-app/.
3. Set **public read access** for all files.

Step 4: Access the Website

Go to **Properties** → **Static Website Hosting** and copy the **S3 URL**.

3.3 Setting Up AWS CloudFront for Faster Delivery

CloudFront **improves performance** by caching content at **edge locations worldwide**.

Step 1: Create a CloudFront Distribution

1. Go to **AWS CloudFront**.
2. Click **Create Distribution**.
3. Choose the S3 bucket as the **origin**.
4. Set **Viewer Protocol Policy** to **Redirect HTTP to HTTPS**.
5. Click **Create Distribution** and copy the CloudFront URL.

✓ Your application is now **deployed globally via CloudFront**.

Chapter 4: Deploying an Angular Application to Netlify

4.1 Why Use Netlify?

✓ **Simplifies Angular deployment with drag-and-drop hosting.**
✓ **Automatic deployment on Git push.**
✓ **Free HTTPS and CI/CD integration.**

4.2 Deploying Angular to Netlify

Step 1: Build the Angular Application

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

✓ Generates dist/angular-app with optimized files.

Step 2: Sign Up and Create a Netlify Account

1. Go to https://www.netlify.com.
2. Click **Sign Up** and log in with **GitHub or email**.

Step 3: Deploy Manually to Netlify

1. Drag and drop the dist/angular-app folder into the **Netlify dashboard**.
2. Netlify will automatically deploy the app and provide a **live URL**.

4.3 Deploying via GitHub with Continuous Deployment

Step 1: Connect GitHub Repository to Netlify

1. Click **New Site from Git** in Netlify.
2. Select your **GitHub repository**.
3. Set the build command to:

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

4. Set the **Publish directory** to dist/angular-app.

Step 2: Deploy and Automate Builds

✓ **Every Git push** automatically triggers deployment.
✓ Netlify **rebuilds and deploys** the latest version.

CHAPTER 5: BEST PRACTICES FOR DEPLOYING ANGULAR APPLICATIONS

✓ **Use production builds** – ng build --configuration=production optimizes performance.
✓ **Enable caching and compression** – Use **CloudFront, Firebase, or Netlify's CDN**.
✓ **Ensure HTTPS is enabled** – Use **free SSL certificates** provided by Firebase, AWS, or Netlify.
✓ **Use environment variables** – Store API keys and sensitive data securely.

Case Study: How a SaaS Startup Deployed Angular Apps Efficiently

Background

A **SaaS startup** needed to:

✓ **Deploy Angular applications rapidly**.
✓ **Ensure fast global performance**.
✓ **Automate builds and minimize downtime**.

Challenges

- **Manual deployments were slow** and error-prone.
- **Poor global performance** due to lack of CDN.
- **Frequent updates required automated deployments**.

Solution: Using Firebase, AWS S3, and Netlify

✓ **Firebase Hosting** for **quick MVP deployment**.
✓ **AWS S3 + CloudFront** for **scalability and performance**.
✓ **Netlify CI/CD** for **fast automated builds**.

Results

🚀 **Deployment time reduced by 80%**.
🔍 **Website load time improved by 50%** with CDN.
⚡ **Automated builds ensured zero downtime updates**.

By using **multiple hosting solutions**, the startup optimized performance and deployment speed.

Exercise

1. Deploy an Angular app using **Firebase Hosting**.
2. Upload an Angular project to **AWS S3** and serve it with CloudFront.
3. Deploy an Angular project to **Netlify** and enable automatic builds.
4. Compare performance between **Firebase, AWS, and Netlify**.

Conclusion

In this section, we explored:

✓ **How to deploy Angular applications to Firebase, AWS, and Netlify**.

✓ **How to automate deployments with CI/CD**.

✓ **How to optimize performance using global CDNs**.

MANAGING ENVIRONMENT VARIABLES IN ANGULAR

CHAPTER 1: INTRODUCTION TO ENVIRONMENT VARIABLES

1.1 What Are Environment Variables in Angular?

Environment variables in Angular allow you to:

✓ **Store configuration settings** separately for development, testing, and production.
✓ **Prevent sensitive information** (API keys, URLs) from being hardcoded in the application.
✓ **Switch configurations dynamically** based on the environment.

- **Common Use Cases for Environment Variables**:
  - API base URLs (https://dev.api.com vs. https://prod.api.com).
  - Feature toggles (enableLogging: true in dev but false in production).
  - Third-party service credentials (Google Maps API key, Firebase config).

1.2 Why Use Environment Variables?

✓ **Security** – Keeps sensitive credentials out of the main codebase.
✓ **Scalability** – Enables seamless deployment in multiple environments.
✓ **Maintainability** – Centralizes configuration settings for easy updates.

CHAPTER 2: SETTING UP ENVIRONMENT VARIABLES IN ANGULAR

2.1 Default Environment Files in Angular

When you create a new Angular project, **environment configuration files** are generated automatically inside the /src/environments/ directory:

```
/src/environments/
 ├──── environment.ts       # Development environment
 ├──── environment.prod.ts   # Production environment
```

Each file contains an environment object with different settings.

2.2 Configuring Environment Variables

Modify environment.ts for **development settings**:

```
export const environment = {
  production: false,
  apiUrl: 'https://dev.api.com',
  enableLogging: true
};
```

Modify environment.prod.ts for **production settings**:

```
export const environment = {
  production: true,
  apiUrl: 'https://prod.api.com',
  enableLogging: false
};
```

✓ production: false for development, true for production.
✓ apiUrl changes dynamically based on the environment.
✓ enableLogging can be **enabled in development and disabled in production**.

CHAPTER 3: ACCESSING ENVIRONMENT VARIABLES IN ANGULAR COMPONENTS & SERVICES

3.1 Using Environment Variables in a Service

Modify api.service.ts to use apiUrl dynamically:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { environment } from '../environments/environment';

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private baseUrl = environment.apiUrl;

  constructor(private http: HttpClient) {}

  getUsers() {
    return this.http.get(`${this.baseUrl}/users`);
```

```
 }
}
```

✓ Uses environment.apiUrl so the API base URL **automatically changes based on the environment**.

3.2 Using Environment Variables in Components

Modify app.component.ts to display configuration values:

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

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 apiUrl = environment.apiUrl;
 loggingEnabled = environment.enableLogging;
}
```

Modify app.component.html to display the values:

```
<h3>Environment Configuration</h3>
<p>API URL: {{ apiUrl }}</p>
<p>Logging Enabled: {{ loggingEnabled }}</p>
```

✓ Displays **environment-based values dynamically**.

Chapter 4: Switching Environments in Angular

4.1 Using Angular CLI to Select Environments

Angular automatically switches between environments using **build configurations**.

Run the application in **development mode** (uses environment.ts):

ng serve

Run the application in **production mode** (uses environment.prod.ts):

ng serve --configuration=production

Build the application for **production deployment**:

ng build --configuration=production

✓ Ensures that **production variables** are used during the build process.

Chapter 5: Creating Custom Environment Files

5.1 Adding a Staging Environment

For multiple environments (e.g., **staging, testing**), create a new environment file:

1️⃣ **Create a new file** /src/environments/environment.staging.ts

```
export const environment = {
  production: false,
  apiUrl: 'https://staging.api.com',
```

```
 enableLogging: true
};
```

2. Modify angular.json to Include the New Environment

Modify the configurations section in angular.json:

```
"configurations": {
 "production": {
  "fileReplacements": [{
   "replace": "src/environments/environment.ts",
   "with": "src/environments/environment.prod.ts"
  }]
 },
 "staging": {
  "fileReplacements": [{
   "replace": "src/environments/environment.ts",
   "with": "src/environments/environment.staging.ts"
  }]
 }
}
```

3. Serve or Build for Staging

Run the application with **staging configuration**:

```
ng serve --configuration=staging
```

✓ Ensures that **the correct API URL and settings are applied** based on the environment.

CHAPTER 6: STORING ENVIRONMENT VARIABLES SECURELY

6.1 Using .env Files for Sensitive Data

Instead of storing sensitive data in environment.ts, use a **.env file** and load variables dynamically.

1. **Install dotenv package**

npm install dotenv

2. **Create a .env file** in the root directory:

API_URL=https://secure-api.com

ENABLE_LOGGING=false

3. **Modify environment.ts to Load .env Variables**

```
export const environment = {
  production: false,
  apiUrl: process.env['API_URL'] || 'https://default-api.com',
  enableLogging: process.env['ENABLE_LOGGING'] === 'true'
};
```

✓ Keeps **sensitive credentials out of the main codebase**.

Case Study: How a SaaS Company Managed Environments in Angular

Background

A **SaaS company** needed separate **development, staging, and production environments** for:

✓ **Testing new features before release**.
✓ **Using different API endpoints based on environment**.
✓ **Securing credentials like API keys and authentication tokens**.

Challenges

- Developers **hardcoded API URLs**, causing **deployment failures**.
- No separation between **development and production settings**.
- Sensitive credentials were **exposed in source code**.

Solution: Implementing Environment Variables in Angular

✓ Used environment.ts, environment.prod.ts, and environment.staging.ts.
✓ Configured angular.json to switch **automatically between environments**.
✓ Moved sensitive data to **.env files** to **improve security**.

Results

🚀 **Faster deployments**, as developers no longer needed to manually change URLs.
🔒 **Secured API keys**, preventing accidental exposure.
📈 **Better application stability**, reducing environment-related issues by **40%**.

By **optimizing environment variable management**, the company **streamlined development and deployment**.

Exercise

1. **Modify the environment.ts file** to include a new variable (appName).
2. **Create an additional environment file** for testing (environment.testing.ts).
3. **Use environment.apiUrl in a service** to fetch data dynamically.
4. **Switch to a different environment** using ng serve --configuration=testing.
5. **Secure API credentials** using .env files and load them dynamically.

Conclusion

In this section, we explored:

✓ **How to use environment variables in Angular for different configurations**.

✓ **How to switch between development, staging, and production settings**.

✓ **How to securely manage API keys and credentials**.

IMPLEMENTING JWT AUTHENTICATION IN ANGULAR

CHAPTER 1: INTRODUCTION TO JWT AUTHENTICATION IN ANGULAR

1.1 What is JWT Authentication?

JSON Web Token (JWT) is a secure method for user authentication in web applications. It allows users to **log in once** and use a token for subsequent requests instead of sending credentials repeatedly.

✓ **Stateless authentication** – No session storage required on the server.
✓ **Compact & Secure** – Uses cryptographic signing to prevent tampering.
✓ **Scalable** – Works well with distributed systems and APIs.

1.2 How JWT Works in Angular Authentication

1. **User logs in** with credentials (email & password).
2. **Server verifies credentials** and returns a **JWT token**.
3. **Client stores JWT** in **localStorage or sessionStorage**.
4. **Client sends JWT** in **HTTP headers** for secure API requests.
5. **Server verifies JWT** before responding to requests.

✓ This eliminates the need for **session-based authentication,** making it more scalable.

CHAPTER 2: SETTING UP AN ANGULAR AUTHENTICATION MODULE

2.1 Install Dependencies

Install **Angular HTTP Client** and JWT helper library:

ng new angular-auth

cd angular-auth

ng add @angular/material

npm install @autho/angular-jwt

✓ @autho/angular-jwt helps manage and decode JWTs.

2.2 Create Authentication Service

Generate an **Auth Service** to handle login, token storage, and user authentication.

ng generate service services/auth

Modify auth.service.ts:

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


@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private authUrl = 'https://your-api.com/auth';
  private tokenKey = 'authToken';
```

```
 private authStatus = new
BehaviorSubject<boolean>(this.hasToken());


 constructor(private http: HttpClient) {}


 login(credentials: { email: string, password: string }):
Observable<any> {
  return this.http.post<{ token: string }>(`${this.authUrl}/login`,
credentials).pipe(
    tap(response => {
     localStorage.setItem(this.tokenKey, response.token);
     this.authStatus.next(true);
    })
  );
}


 logout() {
  localStorage.removeItem(this.tokenKey);
  this.authStatus.next(false);
}


 isAuthenticated(): boolean {
  return !!localStorage.getItem(this.tokenKey);
```

```
 }

 private hasToken(): boolean {
  return !!localStorage.getItem(this.tokenKey);
 }

 getAuthStatus() {
  return this.authStatus.asObservable();
 }
}
```

✓ **Handles login, token storage, and authentication checks**.
✓ Uses **RxJS BehaviorSubject** to update authentication status in real-time.

CHAPTER 3: CREATING LOGIN AND REGISTER COMPONENTS

3.1 Generate Components

ng generate component components/login

ng generate component components/register

3.2 Implement Login Form

Modify login.component.ts:

```
import { Component } from '@angular/core';
import { AuthService } from '../../services/auth.service';
```

```
import { Router } from '@angular/router';


@Component({
 selector: 'app-login',
 templateUrl: './login.component.html'
})
export class LoginComponent {
 email = '';
 password = '';
 errorMessage = '';

 constructor(private authService: AuthService, private router:
Router) {}

 login() {
  this.authService.login({ email: this.email, password: this.password
}).subscribe({
    next: () => this.router.navigate(['/dashboard']),
    error: err => this.errorMessage = 'Invalid credentials'
  });
 }
}
```

Modify login.component.html:

```
<form (submit)="login()">
  <input type="email" [(ngModel)]="email" placeholder="Email" required>
  <input type="password" [(ngModel)]="password" placeholder="Password" required>
  <button type="submit">Login</button>
  <p *ngIf="errorMessage">{{ errorMessage }}</p>
</form>
```

✓ **Logs in the user and redirects to the dashboard.**
✓ **Displays error messages for invalid credentials.**

CHAPTER 4: PROTECTING ROUTES USING AUTHENTICATION GUARDS

4.1 Create an AuthGuard

Generate the guard:

```
ng generate guard guards/auth
```

Modify auth.guard.ts:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { AuthService } from '../services/auth.service';

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

```
export class AuthGuard implements CanActivate {


 constructor(private authService: AuthService, private router:
Router) {}


 canActivate(): boolean {
  if (!this.authService.isAuthenticated()) {
   this.router.navigate(['/login']);
   return false;
  }
  return true;
 }
}
```

✓ **Restricts access to protected routes for unauthenticated users.**

4.2 Apply 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 }
```

```
];
```

✓ **Prevents unauthenticated users from accessing the dashboard.**

CHAPTER 5: AUTOMATICALLY ATTACHING JWT TO API REQUESTS

5.1 Implement an HTTP Interceptor

Create an auth.interceptor.ts file:

ng generate service interceptors/auth-interceptor

Modify auth-interceptor.service.ts:

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler) {
  const token = localStorage.getItem('authToken');

  if (token) {
   const cloned = req.clone({
    setHeaders: { Authorization: `Bearer ${token}` }
   });
   return next.handle(cloned);
  }
```

```
  return next.handle(req);
 }
}
```

✓ **Automatically adds the JWT token** to each API request.

5.2 Register the Interceptor

Modify app.module.ts:

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { AuthInterceptor } from './interceptors/auth-interceptor.service';

providers: [
 { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
]
```

✓ Ensures **all outgoing API requests include authentication tokens**.

Case Study: How a SaaS Company Secured User Authentication Using JWT

Background

A **SaaS company** faced security challenges:

✓ **Session hijacking** due to insecure cookie-based authentication.
✓ **Slow authentication** in high-traffic applications.
✓ **Unauthorized API access vulnerabilities**.

Challenges

- User **sessions expired unexpectedly**.
- **Storing sensitive session data on the server** increased security risks.
- **Scalability issues** when managing user authentication.

Solution: Implementing JWT Authentication in Angular

The company:

✓ Migrated to **JWT-based authentication**.
✓ Implemented **route guards** to restrict unauthorized access.
✓ Used **interceptors** to attach JWT tokens to API requests.

Results

🚀 **Authentication speed improved by 40%**.
🔒 **Stronger security**, preventing session hijacking.
⚡ **Easier scalability**, with stateless authentication.

By using **JWT authentication in Angular**, the SaaS platform **secured user access and improved performance**.

Exercise

1. Implement a **JWT authentication service** that stores and retrieves tokens.
2. Create a **LoginComponent** that authenticates users using JWT.

3. Implement an **AuthGuard** to restrict access to a protected dashboard.
4. Use an **HTTP interceptor** to attach JWT tokens to API requests.

Conclusion

In this section, we explored:

✓ **How JWT authentication secures Angular applications.**
✓ **How to create a login system using Angular services and JWT.**
✓ **How to protect routes using AuthGuards.**
✓ **How to attach JWTs to API requests using interceptors.**

SECURING APIS & HANDLING USER AUTHORIZATION IN ANGULAR

CHAPTER 1: INTRODUCTION TO API SECURITY & USER AUTHORIZATION

1.1 Why Secure APIs in Angular Applications?

APIs are the **backbone of web applications**, providing data exchange between **front-end clients (Angular apps) and back-end servers**. Proper security ensures:

✓ **Protection against unauthorized access**.
✓ **Prevention of data breaches**.
✓ **Compliance with security standards (OAuth, JWT, etc.)**.
✓ **Enhanced user authentication & role-based access control**.

1.2 Common API Security Threats

Threat	Description	Solution
SQL Injection	Attackers manipulate SQL queries.	Use **parameterized queries** & ORM.
Cross-Site Scripting (XSS)	Injects malicious scripts into client-side code.	Enable **Content Security Policy (CSP)**.
Cross-Site Request Forgery (CSRF)	Trick users into executing unwanted actions.	Use **CSRF tokens** for API requests.
Man-in-the-Middle (MITM) Attacks	Intercepts communication between client & server.	Use **HTTPS & TLS encryption**.

Brute Force Attacks	Attackers guess login credentials.	Implement **rate limiting & account lockout**.

CHAPTER 2: IMPLEMENTING API AUTHENTICATION IN ANGULAR

2.1 What is Authentication?

Authentication verifies **who the user is**.

✓ In Angular apps, authentication is done via **tokens** (e.g., JWT – JSON Web Token).
✓ **Frontend sends credentials,** server verifies them, and returns an authentication token.

2.2 Setting Up a Secure API Endpoint (Node.js + Express)

Create an authentication API using **Node.js & Express**:

Step 1: Install Dependencies

npm install express jsonwebtoken cors body-parser bcryptjs dotenv

✓ **jsonwebtoken** → Handles JWT authentication.
✓ **bcryptjs** → Hashes passwords securely.

Step 2: Set Up Authentication API (server.js)

```
require('dotenv').config();

const express = require('express');

const jwt = require('jsonwebtoken');

const bcrypt = require('bcryptjs');

const cors = require('cors');
```

```
const app = express();

app.use(express.json());

app.use(cors());


const users = [{ id: 1, username: "admin", password: "$2a$10$5X.a1"
}]; // Hashed password


app.post('/api/login', (req, res) => {

 const { username, password } = req.body;

 const user = users.find(user => user.username === username);


 if (!user || !bcrypt.compareSync(password, user.password)) {

  return res.status(401).json({ message: "Invalid credentials" });

 }


 const token = jwt.sign({ id: user.id, username: user.username },
process.env.JWT_SECRET, { expiresIn: "1h" });

 res.json({ token });

});


const PORT = process.env.PORT || 3000;

app.listen(PORT, () => console.log(`Server running on port
${PORT}`));
```

✓ **JWT is generated and returned** on successful login.
✓ **Passwords are hashed using bcryptjs** for security.

2.3 Setting Up Authentication in Angular

Step 1: Install @autho/angular-jwt for JWT Handling

```
npm install @autho/angular-jwt
```

Step 2: Create an Authentication Service (auth.service.ts)

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

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private apiUrl = 'http://localhost:3000/api/login';

  constructor(private http: HttpClient) {}

  login(credentials: { username: string; password: string }): Observable<any> {
    return this.http.post<any>(this.apiUrl, credentials);
  }
```

```
 setToken(token: string) {
  localStorage.setItem('token', token);
 }

 getToken(): string | null {
  return localStorage.getItem('token');
 }

 logout() {
  localStorage.removeItem('token');
 }

 isAuthenticated(): boolean {
  return !!this.getToken();
 }
}
```

✓ **Saves JWT token in localStorage** after login.
✓ **Checks authentication status** via isAuthenticated().

2.4 Creating a Login Component

Step 1: Create the Component

ng generate component login

Step 2: Implement Login Logic (login.component.ts)

```
import { Component } from '@angular/core';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

@Component({
 selector: 'app-login',
 templateUrl: './login.component.html'
})
export class LoginComponent {
 username = '';
 password = '';
 errorMessage = '';

 constructor(private authService: AuthService, private router: Router) {}

 login() {
  this.authService.login({ username: this.username, password: this.password }).subscribe({
    next: (response) => {
     this.authService.setToken(response.token);
     this.router.navigate(['/dashboard']);
```

```
    },
    error: () => {
      this.errorMessage = 'Invalid credentials';
    }
   });
 }
}
```

✓ **Calls API for authentication,** saves JWT token, and navigates to **dashboard** on success.

2.5 Protecting Routes Using Route Guards

Route Guards prevent unauthorized users from accessing secure pages.

Step 1: Generate the Auth Guard

ng generate guard auth

Step 2: Implement Guard Logic (auth.guard.ts)

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

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

```
export class AuthGuard implements CanActivate {
 constructor(private authService: AuthService, private router: Router) {}

 canActivate(): boolean {
  if (this.authService.isAuthenticated()) {
   return true;
  } else {
   this.router.navigate(['/login']);
   return false;
  }
 }
}
```

✓ Redirects users to **login page** if they are **not authenticated**.

2.6 Applying the Route Guard in app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';
import { LoginComponent } from './login/login.component';
import { AuthGuard } from './auth.guard';
```

```
const routes: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] } // Protected route
];

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

✓ **Ensures that only authenticated users can access the dashboard**.

CHAPTER 3: ENHANCING SECURITY WITH ROLE-BASED AUTHORIZATION

3.1 Implementing Role-Based Access Control (RBAC)

Modify server.js to include user roles:

```
const users = [{ id: 1, username: "admin", password: "hashedPass", role: "admin" }];

app.post('/api/login', (req, res) => {
  const user = users.find(u => u.username === req.body.username);
```

```
  const token = jwt.sign({ id: user.id, role: user.role },
process.env.JWT_SECRET);
  res.json({ token });
});
```

Modify auth.service.ts to retrieve roles:

```
getUserRole(): string {
  const token = this.getToken();
  if (!token) return '';

  const decodedToken = JSON.parse(atob(token.split('.')[1]));
  return decodedToken.role;
}
```

Modify auth.guard.ts to check roles:

```
canActivate(): boolean {
  if (this.authService.getUserRole() === 'admin') {
    return true;
  }
  this.router.navigate(['/unauthorized']);
  return false;
}
```

✓ **Restricts access to admin-only pages**.

Conclusion

✓ **Implemented JWT authentication in Angular & Node.js.**

✓ **Protected routes using Angular Route Guards.**

✓ **Added Role-Based Access Control (RBAC) for enhanced security.**

CAPSTONE PROJECT: BUILD & DEPLOY A FULL-STACK ANGULAR APPLICATION

CHAPTER 1: INTRODUCTION TO THE CAPSTONE PROJECT

1.1 What is a Full-Stack Angular Application?

A **full-stack Angular application** consists of:

✓ **Frontend** – Developed using **Angular** to create a responsive user interface.
✓ **Backend** – A server-side API using **Node.js and Express.js** to handle business logic.
✓ **Database** – A **MongoDB** database to store and retrieve data.

This project will demonstrate **end-to-end application development,** including:

✓ Building a **dynamic frontend** with Angular.
✓ Developing a **REST API** with Node.js.
✓ Connecting to a **MongoDB database**.
✓ Implementing **authentication and state management**.
✓ Deploying the application to **Firebase, AWS, or Netlify**.

CHAPTER 2: PROJECT OVERVIEW – TASK MANAGEMENT APP

2.1 Project Scope

We will build a **Task Management App** where users can:

✓ **Sign up and log in** using authentication.
✓ **Create, update, delete tasks**.

✓ **View task lists with real-time updates**.

✓ **Deploy the app to Firebase, AWS, or Netlify**.

CHAPTER 3: SETTING UP THE FULL-STACK APPLICATION

3.1 Installing Angular and Node.js

Step 1: Install Angular CLI and Create a New Angular App

npm install -g @angular/cli

ng new task-manager

cd task-manager

✓ Creates a new **Angular project** inside the task-manager folder.

Step 2: Install and Set Up Node.js Backend

mkdir backend

cd backend

npm init -y

✓ Initializes a **Node.js project** for the backend API.

Step 3: Install Express.js and MongoDB Driver

npm install express mongoose cors dotenv jsonwebtoken bcryptjs

✓ Installs **Express (for API), MongoDB driver, JWT (for authentication), and CORS**.

CHAPTER 4: BUILDING THE BACKEND API (NODE.JS + MONGODB)

4.1 Creating the Express Server

Step 1: Create server.js in the backend Folder

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
require('dotenv').config();

const app = express();
app.use(express.json());
app.use(cors());

mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"));

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ Connects to **MongoDB** and starts an **Express server**.

4.2 Creating the Task Model

Step 1: Define models/Task.js

```
const mongoose = require('mongoose');
```

```
const TaskSchema = new mongoose.Schema({
 title: String,
 description: String,
 status: { type: String, default: "pending" }
});

module.exports = mongoose.model('Task', TaskSchema);
```

✓ Defines a **Task model** with **title, description, and status** fields.

4.3 Building API Routes for CRUD Operations

Step 1: Define routes/taskRoutes.js

```
const express = require('express');
const Task = require('../models/Task');
const router = express.Router();

// Create Task
router.post('/tasks', async (req, res) => {
 const task = new Task(req.body);
 await task.save();
 res.status(201).json(task);
});
```

```
// Get Tasks
router.get('/tasks', async (req, res) => {
  const tasks = await Task.find();
  res.json(tasks);
});

// Update Task
router.put('/tasks/:id', async (req, res) => {
  const task = await Task.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(task);
});

// Delete Task
router.delete('/tasks/:id', async (req, res) => {
  await Task.findByIdAndDelete(req.params.id);
  res.json({ message: "Task deleted" });
});

module.exports = router;
```

✓ Defines API routes for **creating, reading, updating, and deleting tasks**.

4.4 Connecting Routes to Server

Modify server.js to include the routes:

```
const taskRoutes = require('./routes/taskRoutes');

app.use('/api', taskRoutes);
```

✓ Now, tasks can be managed via **/api/tasks endpoint**.

CHAPTER 5: DEVELOPING THE ANGULAR FRONTEND

5.1 Creating Task Service to Interact with API

Step 1: Generate a Service

```
ng generate service services/task
```

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

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

@Injectable({
  providedIn: 'root'
})
export class TaskService {
  private apiUrl = 'http://localhost:5000/api/tasks';

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

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

 createTask(task: any): Observable<any> {
  return this.http.post(this.apiUrl, task);
 }

 updateTask(id: string, task: any): Observable<any> {
  return this.http.put(`${this.apiUrl}/${id}`, task);
 }

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

✓ Handles **HTTP requests** to the backend.

5.2 Building the Task Component

Step 1: Generate a Task Component

ng generate component components/task

Step 2: Implement Task Listing in task.component.ts

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

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

 constructor(private taskService: TaskService) {}

 ngOnInit() {
  this.taskService.getTasks().subscribe(data => this.tasks = data);
 }
}
```

✓ Fetches **tasks from the API and displays them**.

CHAPTER 6: DEPLOYING THE APPLICATION

6.1 Deploying the Backend to Heroku

Step 1: Initialize a Git Repository

git init

git add .

git commit -m "Deploy Backend"

Step 2: Deploy to Heroku

heroku create

git push heroku main

heroku open

✓ Deploys the backend API to **Heroku**.

6.2 Deploying the Frontend to Firebase

Step 1: Install Firebase CLI

npm install -g firebase-tools

firebase login

Step 2: Initialize Firebase Hosting

firebase init

✓ Select **Hosting** and choose your Firebase project.

Step 3: Build and Deploy

ng build --configuration=production

firebase deploy

✓ **Application is now live** on Firebase Hosting!

Exercise

1. **Complete the Task Manager App** by adding authentication.
2. Deploy the **backend to Heroku**.
3. Deploy the **frontend to Firebase, AWS, or Netlify**.
4. Optimize performance using **lazy loading and caching**.

Conclusion

In this section, we:

✓ **Built a full-stack Angular app** with a Node.js backend and MongoDB.

✓ **Created a frontend using Angular to interact with APIs**.

✓ **Deployed the backend to Heroku and the frontend to Firebase**.

FINAL CAPSTONE PROJECT: DEVELOP AND DEPLOY A FULL-STACK ANGULAR APPLICATION INTEGRATING AUTHENTICATION, STATE MANAGEMENT, API COMMUNICATION, AND PERFORMANCE OPTIMIZATION

FINAL CAPSTONE PROJECT: FULL-STACK ANGULAR APPLICATION DEVELOPMENT & DEPLOYMENT

Step 1: Setting Up the Full-Stack Project

1.1 Backend: Node.js with Express & MongoDB

- **Initialize the Backend Project**

mkdir full-stack-app

cd full-stack-app

mkdir backend && cd backend

npm init -y

- **Install Required Dependencies**

npm install express mongoose cors dotenv jsonwebtoken bcryptjs

npm install nodemon --save-dev

✓ express – Backend framework.
✓ mongoose – MongoDB Object Modeling.
✓ cors – Enables cross-origin requests.
✓ dotenv – Loads environment variables.
✓ jsonwebtoken – Implements JWT-based authentication.
✓ bcryptjs – Hashes user passwords.

1.2 Create a Simple API in server.js

Create server.js:

```
require('dotenv').config();

const express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');


const app = express();

app.use(express.json());

app.use(cors());


mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log('MongoDB Connected'))

  .catch(err => console.error(err));


app.get('/', (req, res) => res.send('API is Running'));


app.listen(5000, () => console.log('Server running on port 5000'));
```

- **Create a .env File for Configuration**

```
MONGO_URI=mongodb+srv://your-mongodb-uri

JWT_SECRET=your-secret-key
```

✓ Stores **database connection** and **JWT secret** securely.

Step 2: Implementing Authentication in Backend

2.1 Create a User Model (models/User.js)

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String
});

module.exports = mongoose.model('User', UserSchema);
```

2.2 Create Authentication Routes (routes/auth.js)

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');

const router = express.Router();

// Register User
router.post('/register', async (req, res) => {
  const { name, email, password } = req.body;
```

```
  const hashedPassword = await bcrypt.hash(password, 10);
  const user = new User({ name, email, password: hashedPassword });
  await user.save();
  res.json({ message: 'User Registered' });
});

// Login User
router.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(400).json({ message: 'Invalid Credentials' });
  }
  const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });
  res.json({ token, user });
});

module.exports = router;
```

✓ Implements **JWT-based authentication** for secure login and registration.

2.3 Connect Routes in server.js

Modify server.js to include authentication routes:

app.use('/api/auth', require('./routes/auth'));

✓ The authentication API is now **ready** at http://localhost:5000/api/auth.

Step 3: Setting Up the Angular Frontend

3.1 Create an Angular Project

Run the following commands:

cd ..

ng new frontend

cd frontend

npm install @angular/material @angular/forms @angular/router @ngrx/store ngx-toastr

✓ @angular/material – UI components.
✓ @angular/forms – Form handling.
✓ @ngrx/store – State management.
✓ ngx-toastr – Toast notifications.

3.2 Set Up Authentication Service (auth.service.ts)

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

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

import { BehaviorSubject } from 'rxjs';

```
@Injectable({
 providedIn: 'root'
})
export class AuthService {
 private apiUrl = 'http://localhost:5000/api/auth';
 private userSubject = new BehaviorSubject(null);

 constructor(private http: HttpClient) {}

 register(user: any) {
  return this.http.post(`${this.apiUrl}/register`, user);
 }

 login(user: any) {
  return this.http.post(`${this.apiUrl}/login`, user);
 }

 logout() {
  localStorage.removeItem('token');
  this.userSubject.next(null);
 }
```

```
 getUser() {
   return this.userSubject.asObservable();
 }
}
```

✓ Implements **login, registration, and logout functions**.

3.3 Create Login Component (login.component.ts)

```
import { Component } from '@angular/core';
import { AuthService } from '../services/auth.service';

@Component({
 selector: 'app-login',
 templateUrl: './login.component.html'
})
export class LoginComponent {
 email = '';
 password = '';

 constructor(private authService: AuthService) {}

 login() {
   this.authService.login({ email: this.email, password: this.password
})
```

```
    .subscribe(response => {
      localStorage.setItem('token', response.token);
    });
  }
}
```

✓ Authenticates users and **stores the JWT token**.

Step 4: Implementing State Management with NgRx

Modify store/auth.reducer.ts:

```
import { createReducer, on } from '@ngrx/store';
import { loginSuccess, logout } from './auth.actions';

export const initialState = { user: null };

const _authReducer = createReducer(
  initialState,
  on(loginSuccess, (state, { user }) => ({ ...state, user })),
  on(logout, state => ({ ...state, user: null }))
);

export function authReducer(state, action) {
  return _authReducer(state, action);
```

}

✓ Stores **user authentication state globally**.

Step 5: Optimizing Performance with Lazy Loading

Modify app-routing.module.ts to enable lazy loading:

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

✓ Loads **DashboardModule only when needed** to **reduce initial load time**.

Step 6: Deploying the Application

6.1 Deploy Backend to Heroku

```
heroku create my-fullstack-app
git push heroku main
```

✓ Deploys the **backend to Heroku**.

6.2 Deploy Frontend to Firebase

```
ng build --configuration=production
firebase deploy
```

✓ Deploys the **frontend to Firebase Hosting**.

Conclusion

✓ **Developed a Full-Stack Angular Application** with authentication.
✓ **Implemented NgRx for state management**.
✓ **Optimized performance using lazy loading**.
✓ **Deployed backend to Heroku and frontend to Firebase**.