**2️⃣ Add Dependencies0**

Include these dependencies:

✅ **Spring Web** → For REST APIs

✅ **Spring Boot Security** → For Authentication & Authorization

✅ **Spring Data JPA** → For ORM (Hibernate)

✅ **MySQL Driver** → To connect with MySQL

✅ **Lombok** → To reduce boilerplate code

**🔹 Step 4: Understanding REST API & Testing**

**Concept Explanation**

• **REST API (Representational State Transfer)** is a way for applications to communicate over HTTP.

• **Spring Boot REST Controller** allows us to expose endpoints that handle requests and return responses.

• We use **@RestController** to define a RESTful API, and **@RequestMapping** to specify the base URL.

**Interview Questions & Answers**

❓ **Q1: What is a REST API?**

✅ **A1:** A REST API (Representational State Transfer) is a web service that follows REST principles, allowing different applications to communicate using HTTP methods like GET, POST, PUT, and DELETE.

❓ **Q2: What is @RestController in Spring Boot?**

✅ **A2:** @RestController is a Spring annotation that combines @Controller and @ResponseBody. It tells Spring that this class handles HTTP requests and directly returns JSON or String responses instead of rendering a view.

**🔹 Step 5: Understanding Entity, ORM, and JPA**

**Concept Explanation**

• **Entity (Model):** Represents a table in the database.

• **ORM (Object-Relational Mapping):** Converts Java objects into database records.

• **JPA (Java Persistence API):** A standard for ORM that allows us to interact with the database without writing SQL queries.

• **Hibernate:** A popular JPA implementation in Spring Boot.

📌 **In our code:**

• @Entity marks the class as a database table.

• @Table(name = "users") sets the table name.

• @Id & @GeneratedValue make id the primary key and auto-generate its values.

• @Column ensures data constraints (e.g., nullable = false prevents empty values).

• @Enumerated(EnumType.STRING) is used for the Role enum, storing values as strings in the database.

**Interview Questions & Answers**

❓ **Q3: What is JPA in Spring Boot?**

✅ **A3:** JPA (Java Persistence API) is a specification that defines how Java objects can be mapped to relational database tables. Spring Boot uses Hibernate as the default JPA implementation.

❓ **Q4: What is the difference between JPA and Hibernate?**

✅ **A4:** JPA is a specification (just an interface), while Hibernate is an implementation of JPA. Spring Boot supports both, but Hibernate is commonly used.

❓ **Q5: What does @GeneratedValue(strategy = GenerationType.IDENTITY) do?**

✅ **A5:** It tells Hibernate to auto-increment the id column using the database’s identity column feature.

**🔹 Step 6: Understanding Repository Layer**

**Concept Explanation**

• The **Repository Layer** is responsible for database operations.

• **Spring Data JPA** simplifies database access with **JpaRepository**, so we don’t need to write SQL queries.

• findByEmail(String email) is a custom method that Spring Data JPA automatically translates into:

SELECT \* FROM users WHERE email = ?;

📌 **In our code:**

• JpaRepository<User, Long> → Manages User entity with Long as the primary key.

• Optional<User> findByEmail(String email); → Fetches users by email.

**Interview Questions & Answers**

❓ **Q6: What is Spring Data JPA?**

✅ **A6:** Spring Data JPA is a part of Spring Framework that simplifies database interactions using JPA. It provides built-in methods to perform CRUD operations without writing SQL.

❓ **Q7: What is JpaRepository in Spring Boot?**

✅ **A7:** JpaRepository is an interface that provides CRUD and pagination support for JPA entities. It extends CrudRepository and PagingAndSortingRepository.

❓ **Q8: Why do we use Optional in findByEmail(String email)?**

✅ **A8:** Optional helps avoid NullPointerException by allowing us to check if a value exists before using it.

**🔹 Step 7: Implementing the Service Layer**

**📌 Concept Explanation**

The **Service Layer** is responsible for handling business logic. It acts as a bridge between the **Controller** (API) and the **Repository** (Database).

**🔹 Why do we need a Service Layer?**

1. **Separation of Concerns:** It keeps the Controller lightweight and reusable.

2. **Code Reusability:** Business logic is centralized in one place.

3. **Better Testing:** We can test business logic separately.

**1️⃣ Create the Service Interface**

Inside com.workspace.smart, create a new package **service** and add:

📌 **UserService.java**

✅ **This interface defines methods for user operations.**

**2️⃣ Implement the Service**

Inside service, create:

📌 **UserServiceImpl.java**

✅ **This class implements all user-related logic.**

**🔹 Step 8: Creating API Endpoints**

Now, we’ll create **REST APIs** for **User Management**.

**📌 Concept Explanation**

The **Controller Layer** handles HTTP requests and interacts with the **Service Layer** to process data.

**🔹 API Endpoints We’ll Create**

**HTTP Method** **Endpoint** **Description**

POST /api/users Create a new user

GET /api/users/{id} Get a user by ID

GET /api/users Get all users

PUT /api/users/{id} Update user details

DELETE /api/users/{id}. Delete a user

**1️⃣ Create the User Controller**

Inside controller, create:

📌 **UserController.java**

✅ **This controller manages user-related API calls.**

**🔹 Step 9: Testing API Endpoints**

**📌 Steps to Test the APIs**

1. **Run your Spring Boot application**

mvn spring-boot:runmvn spring-boot:run

2. Open **Postman** and test the following APIs:

**✅ Create User (POST)**

• **URL:** http://localhost:8080/api/users

• **Body (JSON):**

{

"name": "John Doe",

"email": "john@example.com",

"password": "password123",

"role": "EMPLOYEE"

}

**✅ Get User by ID (GET)**

• **URL:** http://localhost:8080/api/users/1

**✅ Get All Users (GET)**

• **URL:** http://localhost:8080/api/users

**✅ Update User (PUT)**

• **URL:** http://localhost:8080/api/users/1

• **Body (JSON):**

{

"name": "John Updated",

"email": "john@example.com",

"password": "newpassword",

"role": "ADMIN"

}

**✅ Delete User (DELETE)**

• **URL:** http://localhost:8080/api/users/1

✅ **If all requests work fine, the User Management module is completed!** 🎯

**📌 Interview Questions & Answers**

❓ **Q1: What is the purpose of the Service Layer?**

✅ **A1:** The Service Layer contains business logic, separating it from the Controller. This makes the code modular, maintainable, and testable.

❓ **Q2: What is the difference between @Service and @Repository?**

✅ **A2:**

• @Service: Used for business logic and processing.

• @Repository: Used for database operations (extends JpaRepository).

❓ **Q3: What does @RequestBody do in Spring Boot?**

✅ **A3:** @RequestBody binds the HTTP request body (JSON) to a Java object.

❓ **Q4: Why do we use ResponseEntity in controllers?**

✅ **A4:** ResponseEntity is used to return HTTP status codes along with the response data.

**FOUND ERROR**

The **401 Unauthorized** error usually occurs when:

1. **Spring Security is enabled by default** (Blocking endpoints).

2. **Authentication is required** but missing.

**🔹 Step 10: Fix 401 Unauthorized (Temporary)**

Since we haven’t added authentication yet, we need to **disable security temporarily** to allow API access.

**Solution: Create Security Configuration Package**

📌 **Add SecurityConfig.java inside config package**

package com.workspace.smart.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.csrf().disable()

.authorizeHttpRequests()

.anyRequest().permitAll()

.and()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

return http.build();

}

}

✅ **This disables security so all APIs work without authentication.**

**Important Annotations & Their Uses**

|  |  |
| --- | --- |
| **Annotation** | **Purpose** |
| @Configuration | Marks a class as a Spring configuration class. |
| @Bean. | Declares a Spring-managed bean. |
| @EnableWebSecurity | Enables Spring Security in the application. |
| @Component | Marks a class as a Spring-managed component. |
| @Service | Marks a class as a business logic layer component. |
| @Repository | Marks a class as a data access layer component. |
| @RestController | Combines @Controller and @ResponseBody for REST APIs. |
| @RequestMapping | Maps a class or method to a specific URL. |
| @PostMapping | Maps HTTP POST requests to a method. |
| @GetMapping | Maps HTTP GET requests to a method. |

**🔹 Step 12: Understanding Spring Security & JWT Authentication**

Now that we’ve temporarily disabled security, let’s properly implement **JWT Authentication** to secure our APIs.

**📌 Important Notes on Spring Security**

**🔹 What is Spring Security?**

Spring Security is a framework that provides authentication and authorization for Java applications.

**🔹 Why do we need security?**

1. **To prevent unauthorized access** (Protect sensitive data).

2. **To authenticate users** before allowing them to use APIs.

3. **To enforce role-based access control** (e.g., Admin vs Employee).

**🔹 What is JWT (JSON Web Token)?**

JWT is a compact, self-contained token used for secure authentication between client and server.

**🔹 Why use JWT instead of sessions?**

1. **Stateless authentication** (No need to store sessions on the server).

2. **Faster authentication** (Token-based, no database lookup required).

3. **Easier to scale** (Works well in microservices).

**🔹 Interview Questions & Scripted Answers**

❓ **Q1: What is Spring Security, and why is it used?**

✅ **A1:**

Spring Security is a framework used to provide authentication and authorization in Java applications. It ensures that only authorized users can access APIs and perform actions based on their roles.

❓ **Q2: What is the difference between Authentication and Authorization?**

✅ **A2:**

• **Authentication** verifies “Who you are” (e.g., logging in with a username/password).

• **Authorization** determines “What you can do” (e.g., an Admin can create users, but an Employee cannot).

❓ **Q3: What is JWT, and how does it work?**

✅ **A3:**

JWT (JSON Web Token) is a self-contained token used for secure authentication. It consists of:

1. **Header** (Algorithm & Token Type)

2. **Payload** (User Information & Roles)

3. **Signature** (Used to verify the token’s integrity)

A client sends a JWT in the request header, and the server verifies it before allowing access.

**❓ Q6: How does Spring Security validate a JWT token?**

✅ **A6:** When a user sends a request, Spring Security:

1. Extracts the token from the **Authorization** header.

2. **Validates** the token signature.

3. Extracts the **username** from the token.

4. Checks if the token is **expired** or **invalid**.

5. If valid, grants access to the requested resource.

🔹 **This ensures secure API access without storing user sessions.**

**❓ Q7: How can we secure APIs using JWT in Spring Boot?**

✅ **A7:**

1️⃣ Use Spring Security to intercept API requests.

2️⃣ Implement a JwtFilter to check for tokens.

3️⃣ Deny access if the token is missing or invalid.

💡 **Bonus Tip:** **Use role-based authorization** to restrict access to different users. Example:

http.authorizeRequests()

.antMatchers("/admin/\*\*").hasRole("ADMIN")

.antMatchers("/user/\*\*").hasRole("USER")

.anyRequest().authenticated();

🔹 **This ensures that only admins can access /admin endpoints.**

**🔹 Step 13: Implementing JWT Authentication (Login & Register)**

**📌 What Will We Do?**

1. **Create Authentication Controller** (/api/auth/login & /api/auth/register)

2. **Generate JWT Token after login**

3. **Validate JWT Token for securing APIs**

**📌 Important Notes on JWT**

• **JWT (JSON Web Token)** is a secure way to authenticate users.

• After logging in, a **JWT token** is generated and sent to the user.

• The token must be included in API requests for authentication.

**🔹 How JWT Works**

1️⃣ **User logs in** → Server verifies credentials → Server generates JWT Token

2️⃣ **User sends JWT in headers** (Authorization: Bearer <token>)

3️⃣ **Server validates JWT** → If valid, grants access to API

**🔹 Step 1: Add Dependencies**

📌 Add this in **pom.xml** under <dependencies>:

**<!-- Spring Security -->**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**<!-- JWT Library -->**

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.11.5</version>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.11.5</version>  
 <scope>runtime</scope>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if you use Gson -->  
 <version>0.11.5</version>  
 <scope>runtime</scope>  
</dependency>

**<!-- Spring Boot Web -->**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

✅ **This adds JWT and Spring Security dependencies.**

**🔹 Step 2: Create JWT Utility Class**

Inside config package, create:

📌 **JwtUtil.java**

✅ **This class generates and validates JWT tokens.**