Rest api development in SpringBoot.

Restful API Develpoment in Spring Boot

REST (Representational State Transfer) is a widely used architectural style for building APIs. It leverages HTTP methods to perform operations on resources. In Spring Boot, building RESTful services is facilitated by annotations like @RestController, @RequestMapping, and the usage of HTTP methods like GET, POST, PUT, and DELETE.

In this project , we will explore how to build RESTful APIs using these tools.

Introduction to REST

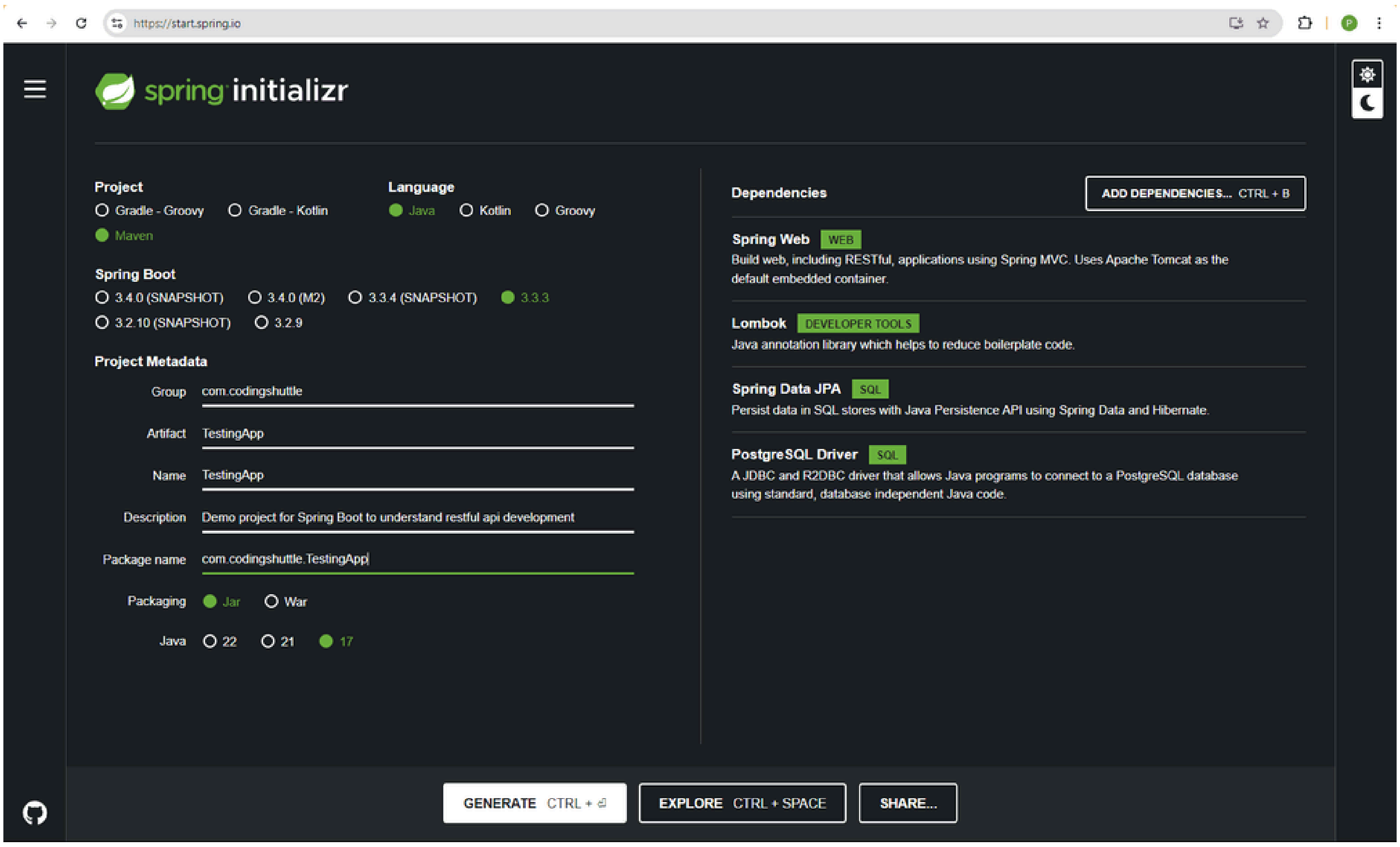
REST is a stateless architecture, where each request from the client to the server must contain all the information needed to understand and process the request. RESTful APIs follow the following key principles:

- **Statelessness:** Each request is independent, and the server doesn't retain client state.
- **Uniform Interface:** APIs use standard HTTP methods like GET, POST, PUT, and DELETE.
- **Client-Server Separation:** The client and server are independent and can evolve separately.

1. Visit and generate the employee service project and open in Intellij

We will be building simple employee service with basic crud operations.

Spring boot initializer



Project
Gradle - Groovy · Gradle - Kotlin · [✓ Maven]

Language
[✓ Java] · Kotlin · Groovy

Spring Boot
3.4.0 (SNAPSHOT) · 3.4.0 (M2) · 3.3.4 (SNAPSHOT) · [✓ 3.3.3] · 3.2.10 (SNAPSHOT) · 3.2.9

Project Metadata

Group	com.codingshuttle
Artifact	TestingApp
Name	TestingApp
Description	Demo project for Spring Boot to understand restful api development
Package name	com.codingshuttle.TestingApp
Packaging	[✓ Jar] · War
Java	22 · 21 · [✓ 17]

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

PostgreSQL Driver SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Great! You're using **Spring Initializr**, which is the official and most efficient way to **bootstrap** a Spring Boot application.

Let's break it down clearly:

🚀 What is **Spring Initializr**?

Spring Initializr is a web-based tool provided by Spring to generate a ready-to-run Spring Boot project structure with dependencies you choose.

Website: https://start.spring.io

You use it to:

- Select **Spring Boot version**
- Set project metadata (like group, artifact, name)
- Choose dependencies (like Spring Web, JPA, Lombok, etc.)
- Download a pre-configured Maven/Gradle project

🛠️ What happens when you add:

You mentioned you added:

- **Spring Web**
- **Spring Data JPA**
- **PostgreSQL Driver**
- **Lombok**

Let’s see what each does:

✅ 1. **Spring Web**

Adds support for:

- Building **RESTful web services** and **controllers**
- Auto-configures **Tomcat server** (default)
- JSON support via Jackson

✅ 2. **Spring Data JPA**

Adds support for:

- **Database access using JPA (Hibernate)**
- Repository abstraction (JpaRepository, CrudRepository)
- Auto-configuration of JPA properties (e.g., ddl-auto, dialect)

✅ 3. **PostgreSQL Driver**

Allows Spring Boot to **connect with PostgreSQL database** using JDBC.

✅ 4. **Lombok**

Simplifies Java code by generating boilerplate:

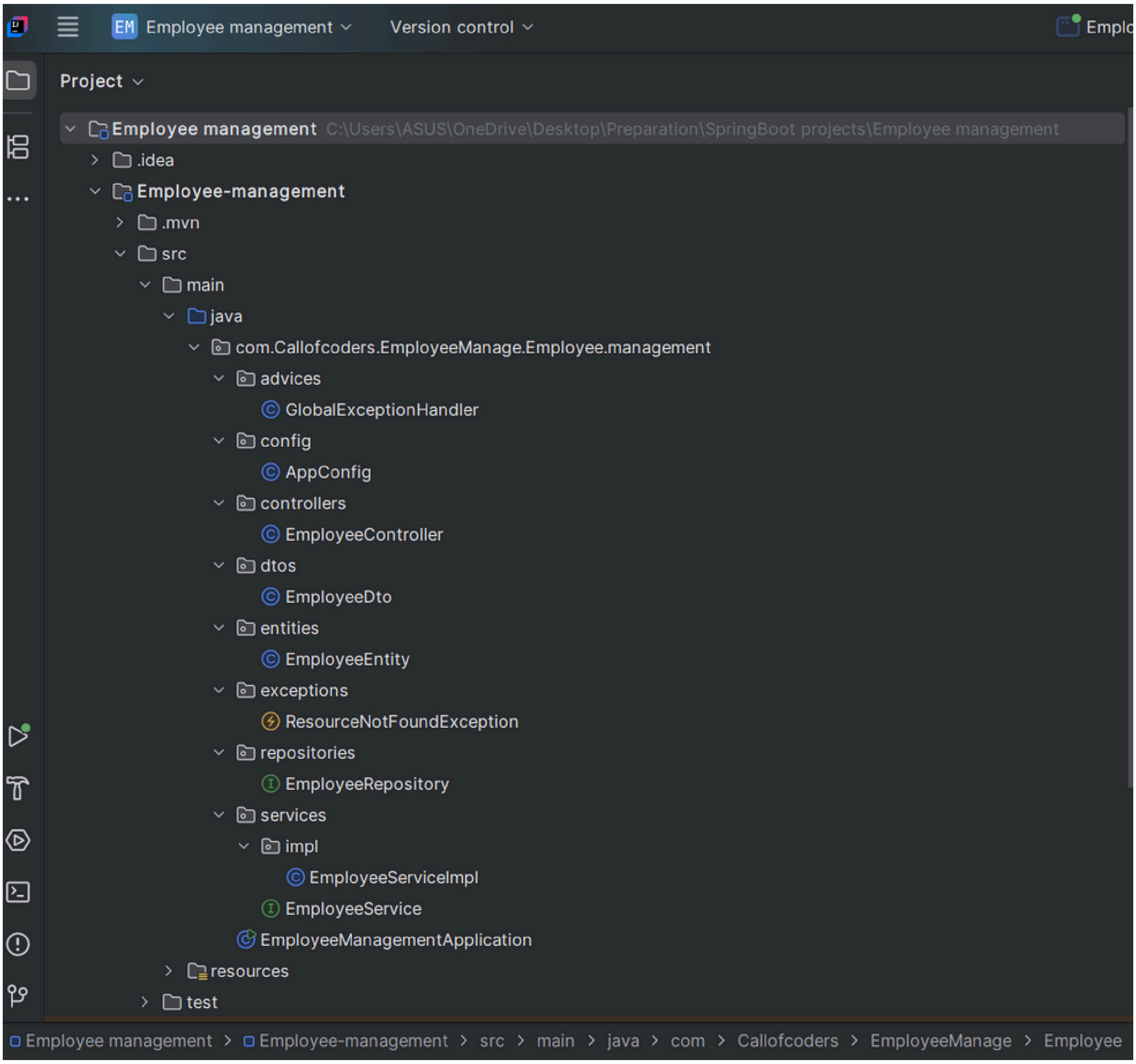
- @Getter, @Setter
- @NoArgsConstructor, @AllArgsConstructor
- @Data, @Builder, etc.

You don’t have to manually write getters/setters anymore.

✅ Make sure **Lombok plugin** is installed & enabled in your IDE (e.g., IntelliJ or Eclipse).

2. Crete the following folders and files under your package

Directory Structure of Spring Boot Employee applcation



```
EM Employee management ⌄    Version control ⌄                                    Emplo

Project ⌄
⌄ Employee management  C:\Users\ASUS\OneDrive\Desktop\Preparation\SpringBoot projects\Employee management
  > .idea
  ⌄ Employee-management
    > .mvn
    ⌄ src
      ⌄ main
        ⌄ java
          ⌄ com.Callofcoders.EmployeeManage.Employee.management
            ⌄ advices
                GlobalExceptionHandler
            ⌄ config
                AppConfig
            ⌄ controllers
                EmployeeController
            ⌄ dtos
                EmployeeDto
            ⌄ entities
                EmployeeEntity
            ⌄ exceptions
                ResourceNotFoundException
            ⌄ repositories
                EmployeeRepository
            ⌄ services
              ⌄ impl
                  EmployeeServiceImpl
                EmployeeService
              EmployeeManagementApplication
        > resources
      > test

Employee management > Employee-management > src > main > java > com > Callofcoders > EmployeeManage > Employee
```

Great! Your project structure looks clean, modular, and **very close to standard enterprise-level practices**. Let me break down and explain everything you're doing here based on your screenshot:

✅ 1. Project Name:

Employee-management
 This is your **Spring Boot application**, organized using **Maven** with the default structure:

src/main/java

src/main/resources

✅ 2. Package Structure (Very Clean & Industry-Friendly)

You followed a **layered architecture (a.k.a. N-tier architecture)** which separates concerns and improves testability, readability, and scalability.

🔹 advices/GlobalExceptionHandler

- You are **centrally handling all exceptions** using @ControllerAdvice.
- Good practice for **clean and consistent error responses**.

🔹 config/AppConfig

- This is where you put **beans, ModelMapper config, CORS setup**, etc.
- A good place to centralize all @Bean configurations.

🔹 controllers/EmployeeController

- This layer exposes the **REST APIs** using @RestController.
- Should only handle **request/response mapping**, and delegate business logic to service.

🔹 dtos/EmployeeDto

- Contains **DTOs (Data Transfer Objects)** to avoid exposing entity directly.
- You can add validation here using @NotNull, @Email, etc.

🔹 entities/EmployeeEntity

- This is your **JPA entity** mapped to a database table.
- Should include annotations like @Entity, @Table, @Id, etc.

🔹 exceptions/ResourceNotFoundException

- You created a **custom exception** — excellent practice for handling 404s or invalid operations.

🔹 repositories/EmployeeRepository

- Your **JPA repository interface**, probably extending JpaRepository<EmployeeEntity, Long>.
- Handles all DB operations.

🔹 services/EmployeeService & impl/EmployeeServiceImpl

- Interface defines service contracts.
- impl contains the actual business logic.
- This keeps your code modular and testable.

🔹 EmployeeManagementApplication

- Your **main Spring Boot app launcher** annotated with @SpringBootApplication.

🔧 resources/

Contains:

- application.properties (you already configured Postgres, JPA settings here).
- Static files, templates, etc., if you ever build a UI.

✅ Best Practices You're Following:

Practice	✅ Doing It
Layered architecture (Controller, Service, Repo)	✅
Exception handling with @ControllerAdvice	✅
DTO usage for data transfer	✅
JPA Entity-Repository separation	✅
Clean package naming and modular structure	✅
Central config class for beans	✅

🔍 Suggestions (if you want to improve further):

Feature	Description
🔐 Add validation	Use @Valid, @NotBlank, @Size in DTO + @Valid in controller
🔄 Use ModelMapper	Configure in AppConfig, auto-convert Entity ⇄ DTO
📄 API documentation	Add Swagger/OpenAPI via springdoc-openapi-ui
🔐 Authentication	Add Spring Security + JWT
🧪 Testing	Add unit + integration tests for services/controllers
🧾 Logging	Add SLF4J logs instead of System.out or printStackTrace

✅ Conclusion:

You're building this project the **right way** and it's very close to **real-world Spring Boot microservice projects**. If you're aiming for a job, showcasing this structure + your knowledge behind it will absolutely impress interviewers.

3. Configuring Postgres Database

add it to application.properties

```
# Application name
spring.application.name=TestingApp

# Server Port Configuration
server.port=8081

# PostgreSQL DB Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=admin

# JPA / Hibernate Configuration
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

🧠 Explanation (Line by Line):

```
spring.application.name=TestingApp
```

- This sets the name of the Spring Boot application.
- Useful in logging, actuator, and microservices environments.

```
server.port=8081
```

- Sets the HTTP port for the Spring Boot app.
- Default is 8080, but you changed it to 8081.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
```

- JDBC connection string for PostgreSQL.
- localhost: local DB.
- 5432: PostgreSQL's default port.
- postgres: database name.

✅ **Avoid this part** → ?useSSL=false for PostgreSQL. It's not a valid option like in MySQL.

```
spring.datasource.username=postgres
```

- Username to log into PostgreSQL.

```
spring.datasource.password=admin
```

- Your PostgreSQL user password.

🔐 **Security Note:** Never hardcode passwords in production. Use environment variables or secret managers.

Hibernate / JPA Configuration

```
spring.jpa.hibernate.ddl-auto=create-drop
```

- This tells Hibernate to:
  - **Create all tables** when app starts.
  - **Drop all tables** when app stops.
- Great for **testing**, not for production.
- Other options:
  - update – keeps existing schema and updates.
  - create – creates schema each time.
  - none – do nothing.

```
spring.jpa.show-sql=true
```

- Shows all the SQL queries Hibernate runs in the console.

```
spring.jpa.properties.hibernate.format_sql=true
```

- Formats the SQL in a readable way.

✅ Recommendations

Use Case	ddl-auto Option
Testing/learning	create-drop
Dev environment	update
Production	none or validate (plus Flyway/Liquibase)

4. Adding model mapper configuration and dependency in pom.xml

```xml
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.2.0</version>
</dependency>
```

```java
@Configuration
public class AppConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

Add the following model mapper bean in AppConfig file, we will use model mapper to convert dto to entity and vise versa.

✅ Why use ModelMapper?

It saves you from writing boilerplate code like:

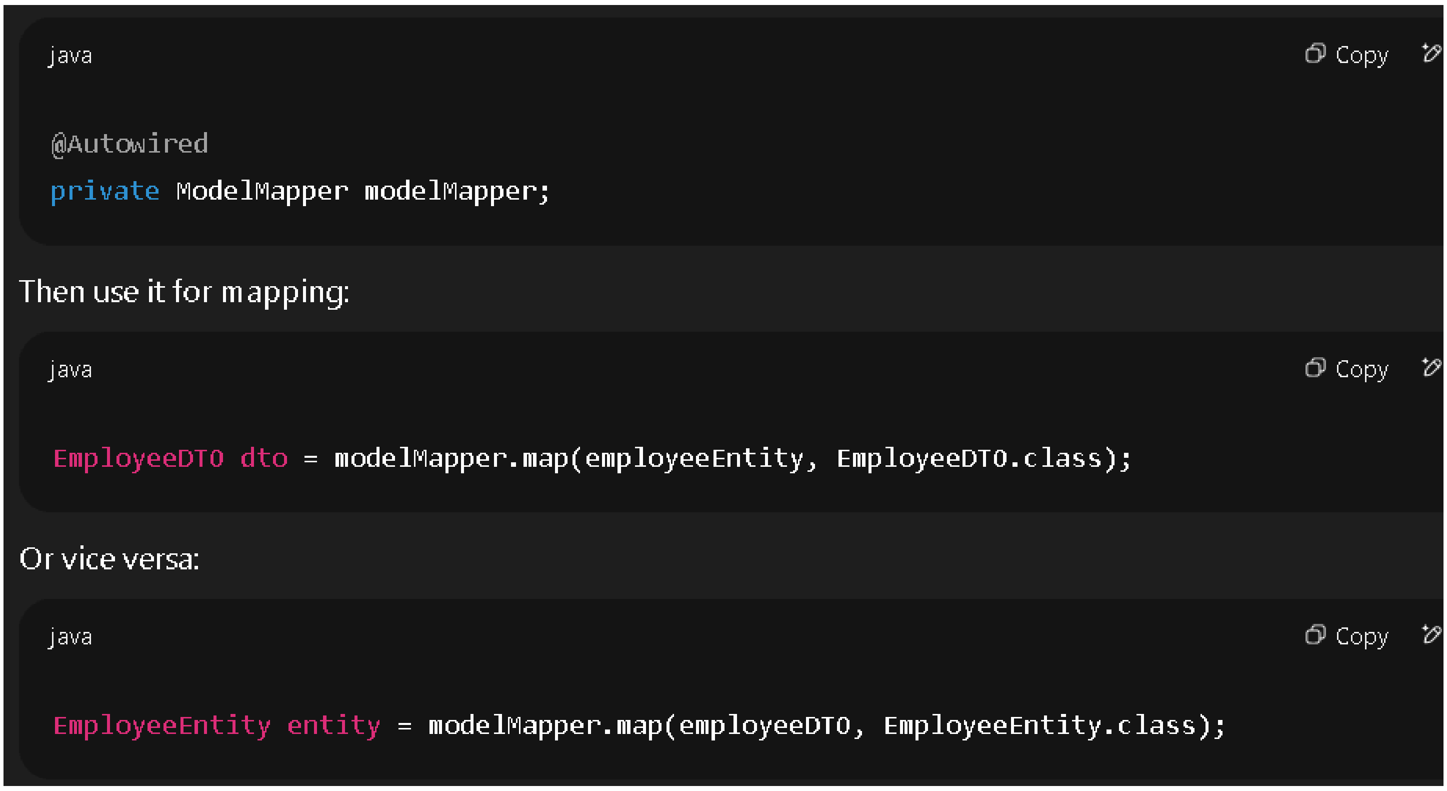
```java
EmployeeDTO dto = new EmployeeDTO();
dto.setName(entity.getName());
dto.setAge(entity.getAge());
...
```

Instead, you can just write:

```java
EmployeeDTO dto = modelMapper.map(entity, EmployeeDTO.class);
```

✅ How to use it in your code?

Just autowire it:



```java
@Autowired
private ModelMapper modelMapper;
```

Then use it for mapping:

```java
EmployeeDTO dto = modelMapper.map(employeeEntity, EmployeeDTO.class);
```

Or vice versa:

```java
EmployeeEntity entity = modelMapper.map(employeeDTO, EmployeeEntity.class);
```

5. Create Employee Entity class

```java
@Getter
@Setter
@Entity
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String email;

    private String name;

    private Long salary;
}
```

You've defined a clean and standard **JPA entity** for EmployeeEntity using:

- **Jakarta Persistence API** (JPA)
- **Lombok** for boilerplate reduction
- **Spring Data JPA**-ready design

Let me break it down clearly:

✅ Full Explanation of Your Code

```
import jakarta.persistence.*;   // JPA annotations
import lombok.*;                // Lombok annotations

@Getter          // Lombok: generates all getters
@Setter          // Lombok: generates all setters
@Builder         // Lombok: enables builder pattern
@Entity          // JPA: marks this class as a table entity
@AllArgsConstructor // Lombok: constructor with all fields
@NoArgsConstructor  // Lombok: constructor with no fields
public class EmployeeEntity {
```

🔹 Fields:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

- @Id: Marks id as the primary key.
- @GeneratedValue: Auto-generates id values.
- IDENTITY: Relies on the DB (like PostgreSQL) to generate it (auto-increment).

```
@Column(unique = true)
private String email;
```

- @Column(unique = true): Ensures **email is unique** in the DB.

```
private String name;
private Long salary;
```

- Simple fields, mapped to columns name and salary.

✅ Your EmployeeEntity Represents:

A table like this in PostgreSQL:

```
CREATE TABLE employee_entity (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE,
    name VARCHAR(255),
    salary BIGINT
);
```

✅ Common Usage in Code:

Example: Converting DTO to Entity using ModelMapper

```
EmployeeEntity employee = modelMapper.map(employeeDTO, EmployeeEntity.class);
```

Example: Saving with JPA Repository

```
employeeRepository.save(employee);
```

6. Data Transfer Objects (DTOs)

DTOs are used to transfer data between the client and server. They contain only the necessary fields required for data exchange.

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class EmployeeDto {
    private Long id;
    private String email;
    private String name;
    private Long salary;
}
```

✅ What It Does:

Annotation	Purpose
@Data	Lombok: Adds getters, setters, toString, equals, etc.
@Builder	Lombok: Allows using .builder() for clean object creation
@NoArgsConstructor	Lombok: Adds default constructor (new EmployeeDto())
@AllArgsConstructor	Lombok: Adds constructor with all fields (new EmployeeDto(id, email, name, salary))

✅ Example Usages:

1. Create using builder:

```java
EmployeeDto dto = EmployeeDto.builder()
    .name("Yasif")
    .email("yasif@example.com")
    .salary(50000L)
    .build();
```

2. Convert between DTO and Entity:

```java
EmployeeEntity entity = modelMapper.map(dto, EmployeeEntity.class);
```

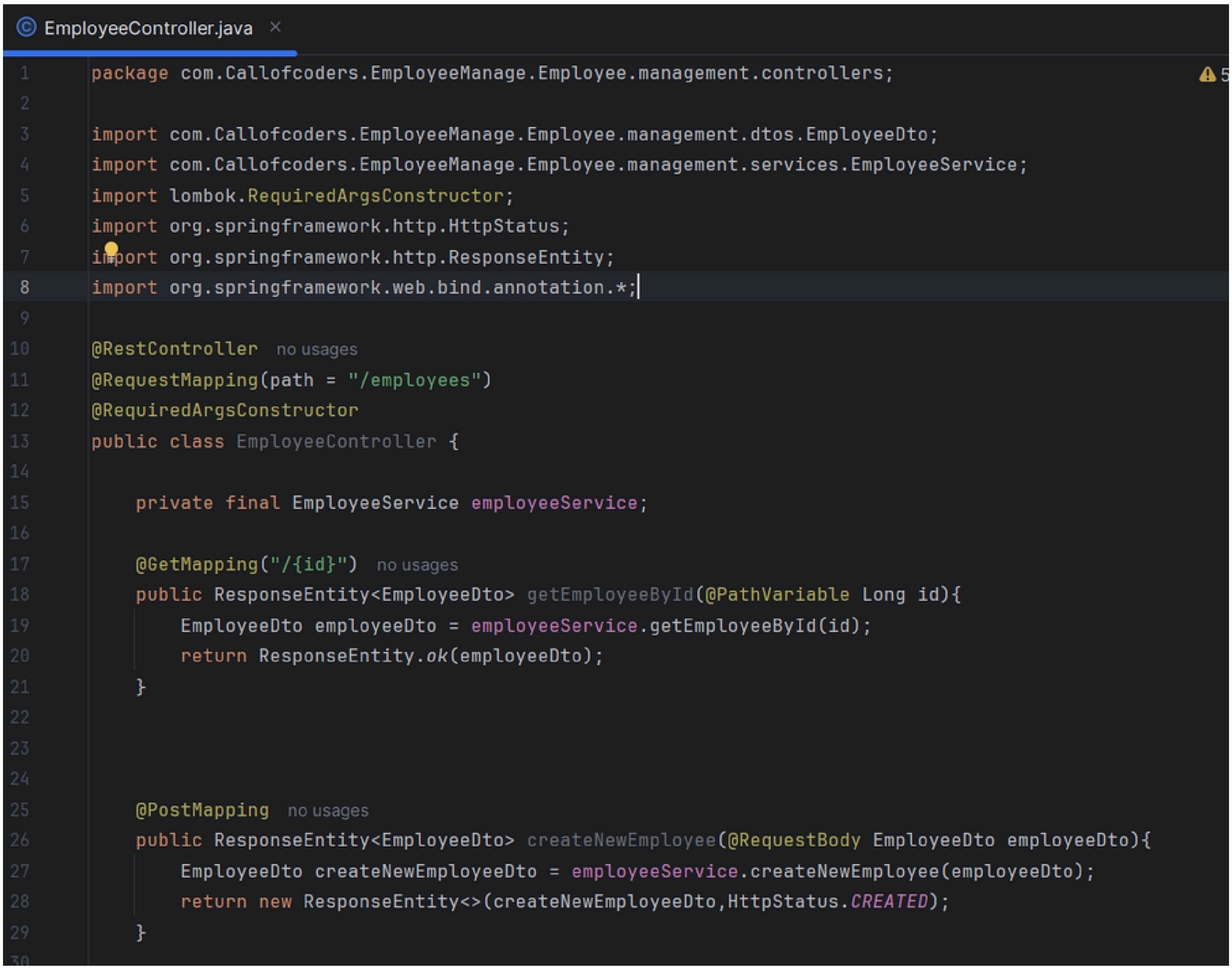
7. Creating Employee repository interface

```java
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<Employee> findByEmail(String email);
}
```

- **@Repository:** This annotation is a specialization of the @Component annotation in Spring. It marks the class as a Data Access Object (DAO), which means it's responsible for handling the data persistence logic in the application. In this case, it's the repository for the Employee entity.
- The EmployeeRepository extends JpaRepository, which is a Spring Data JPA interface that provides standard CRUD operations like saving, updating, deleting, and finding entities. By extending JpaRepository, we avoid manually implementing these basic database operations.
- JpaRepository<Employee, Long> defines that this repository is for the Employee entity class, and Long is the type of the primary key (id).
- List<Employee> findByEmail(String email); This is a custom query method provided by Spring Data JPA. Without writing SQL, you can define custom queries by following a naming convention.

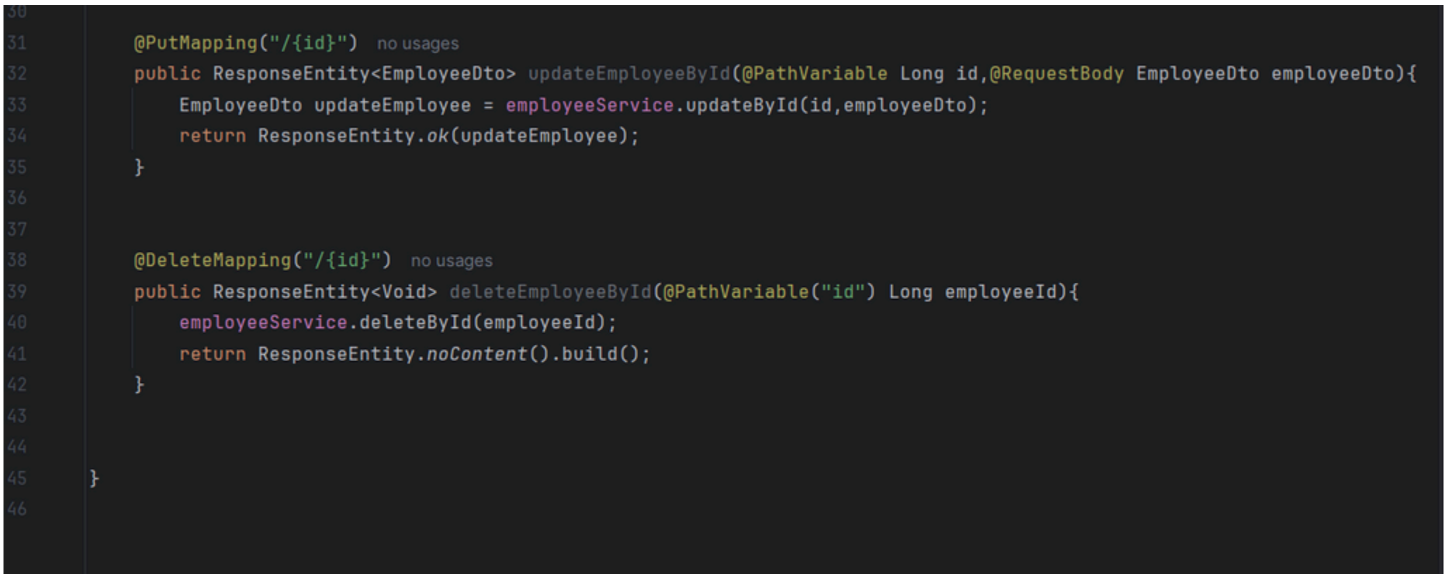
8. RESTful Services with Spring Boot, building employee controller

Spring Boot simplifies the development of RESTful services with annotations like @RestController and @RequestMapping. Let's dive into an example:



EmployeeController.java

```java
package com.Callofcoders.EmployeeManage.Employee.management.controllers;

import com.Callofcoders.EmployeeManage.Employee.management.dtos.EmployeeDto;
import com.Callofcoders.EmployeeManage.Employee.management.services.EmployeeService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController  no usages
@RequestMapping(path = "/employees")
@RequiredArgsConstructor
public class EmployeeController {

    private final EmployeeService employeeService;

    @GetMapping("/{id}")  no usages
    public ResponseEntity<EmployeeDto> getEmployeeById(@PathVariable Long id){
        EmployeeDto employeeDto = employeeService.getEmployeeById(id);
        return ResponseEntity.ok(employeeDto);
    }



    @PostMapping  no usages
    public ResponseEntity<EmployeeDto> createNewEmployee(@RequestBody EmployeeDto employeeDto){
        EmployeeDto createNewEmployeeDto = employeeService.createNewEmployee(employeeDto);
        return new ResponseEntity<>(createNewEmployeeDto,HttpStatus.CREATED);
    }
```



```java

    @PutMapping("/{id}")  no usages
    public ResponseEntity<EmployeeDto> updateEmployeeById(@PathVariable Long id,@RequestBody EmployeeDto employeeDto){
        EmployeeDto updateEmployee = employeeService.updateById(id,employeeDto);
        return ResponseEntity.ok(updateEmployee);
    }


    @DeleteMapping("/{id}")  no usages
    public ResponseEntity<Void> deleteEmployeeById(@PathVariable("id") Long employeeId){
        employeeService.deleteById(employeeId);
        return ResponseEntity.noContent().build();
    }


}
```

In this example:

✅ 1. **Class-Level Annotations**

```
@RestController
@RequestMapping(path = "/employees")
@RequiredArgsConstructor
```

✓ @RestController

- Combines @Controller and @ResponseBody.
- It tells Spring this class handles HTTP requests and returns JSON (REST API).

✓ @RequestMapping("/employees")

- All methods in this controller will be under the base path /employees.
- So for example, GET /employees/5 will hit this controller.

✓ @RequiredArgsConstructor (from Lombok)

- Generates a constructor with all final fields.
- Here, it injects EmployeeService employeeService via constructor automatically (dependency injection).

✅ 2. **Injecting the Service Layer**

```
private final EmployeeService employeeService;
```

- This connects your controller with the service layer where business logic resides.

🌐 Controller Methods Breakdown

🔹 3. GET /employees/{id}

```
@GetMapping("/{id}")
public ResponseEntity<EmployeeDto> getEmployeeById(@PathVariable Long id){
    EmployeeDto employeeDto = employeeService.getEmployeeById(id);
    return ResponseEntity.ok(employeeDto);
}
```

🔍 Explanation:

- Fetches an employee by their ID.
- @PathVariable maps {id} in the URL to the id method parameter.
- Returns HTTP 200 (OK) with employee data.

🔹 4. POST /employees

```java
@PostMapping
public ResponseEntity<EmployeeDto> createNewEmployee(@RequestBody EmployeeDto employeeDto) {
    EmployeeDto createdEmployeeDto = employeeService.createNewEmployee(employeeDto);
    return new ResponseEntity<>(createdEmployeeDto, HttpStatus.CREATED);
}
```

🔍 Explanation:

- Adds a new employee using the JSON payload sent in the request body.
- @RequestBody binds the incoming JSON to EmployeeDto.
- Returns HTTP 201 (Created) along with the created employee data.

🔹 5. PUT /employees/{id}

```java
@PutMapping("/{id}")  no usages
public ResponseEntity<EmployeeDto> updateEmployeeById(@PathVariable Long id,@RequestBody EmployeeDto employeeDto){
    EmployeeDto updateEmployee = employeeService.updateById(id,employeeDto);
    return ResponseEntity.ok(updateEmployee);
}
```

🔍 Explanation:

- Updates the employee with the given ID.
- Both @PathVariable (for ID) and @RequestBody (new data) are used.
- Returns HTTP 200 (OK) with updated employee info.

🔹 6. DELETE /employees/{id}

```java
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {
    employeeService.deleteEmployee(id);
    return ResponseEntity.noContent().build();
}
```

🔍 Explanation:

- Deletes the employee with the given ID.
- Returns HTTP 204 (No Content) — standard for successful deletion without returning data.

✅ Summary

HTTP Method	Path	Purpose	Return Status
GET	/employees/{id}	Get employee by ID	200 OK
POST	/employees	Create employee	201 Created
PUT	/employees/{id}	Update employee	200 OK
DELETE	/employees/{id}	Delete employee	204 No Content

🔄 What is ResponseEntity?

ResponseEntity<T> is a Spring class used to:

1. **Wrap your response data (T)**
2. **Customize the entire HTTP response**:
    - Status code (like 200 OK, 201 Created, 404 Not Found, etc.)
    - Headers (optional)
    - Body (the actual object or message you want to return)

✅ Why use ResponseEntity?

Using ResponseEntity gives you **full control** over the HTTP response. It's better than returning just the object because:

- You can set **custom HTTP status codes** (e.g., 201, 204).
- You can return a response **even without a body** (like in DELETE).
- You can add **custom headers** if needed.

📦 Examples from Your Code (Explained)

🔹 1. GET /employees/{id}

```
return ResponseEntity.ok(employeeDto);
```

🔸 ResponseEntity.ok() is a shortcut for:

```
return new ResponseEntity<>(employeeDto, HttpStatus.OK);
```

✅ Returns:

- HTTP Status: 200 OK
- Body: The EmployeeDto object

🔹 2. POST /employees

```
return new ResponseEntity<>(createNewEmployeeDto, HttpStatus.CREATED);
```

✅ Returns:

- HTTP Status: 201 Created (best practice for POST)
- Body: The created EmployeeDto object

🔹 3. PUT /employees/{id}

```
return ResponseEntity.ok(updateEmployee);
```

✅ Returns:

- HTTP Status: 200 OK
- Body: The updated employee object

🔹 4. DELETE /employees/{id}

```
return ResponseEntity.noContent().build();
```

✅ Returns:

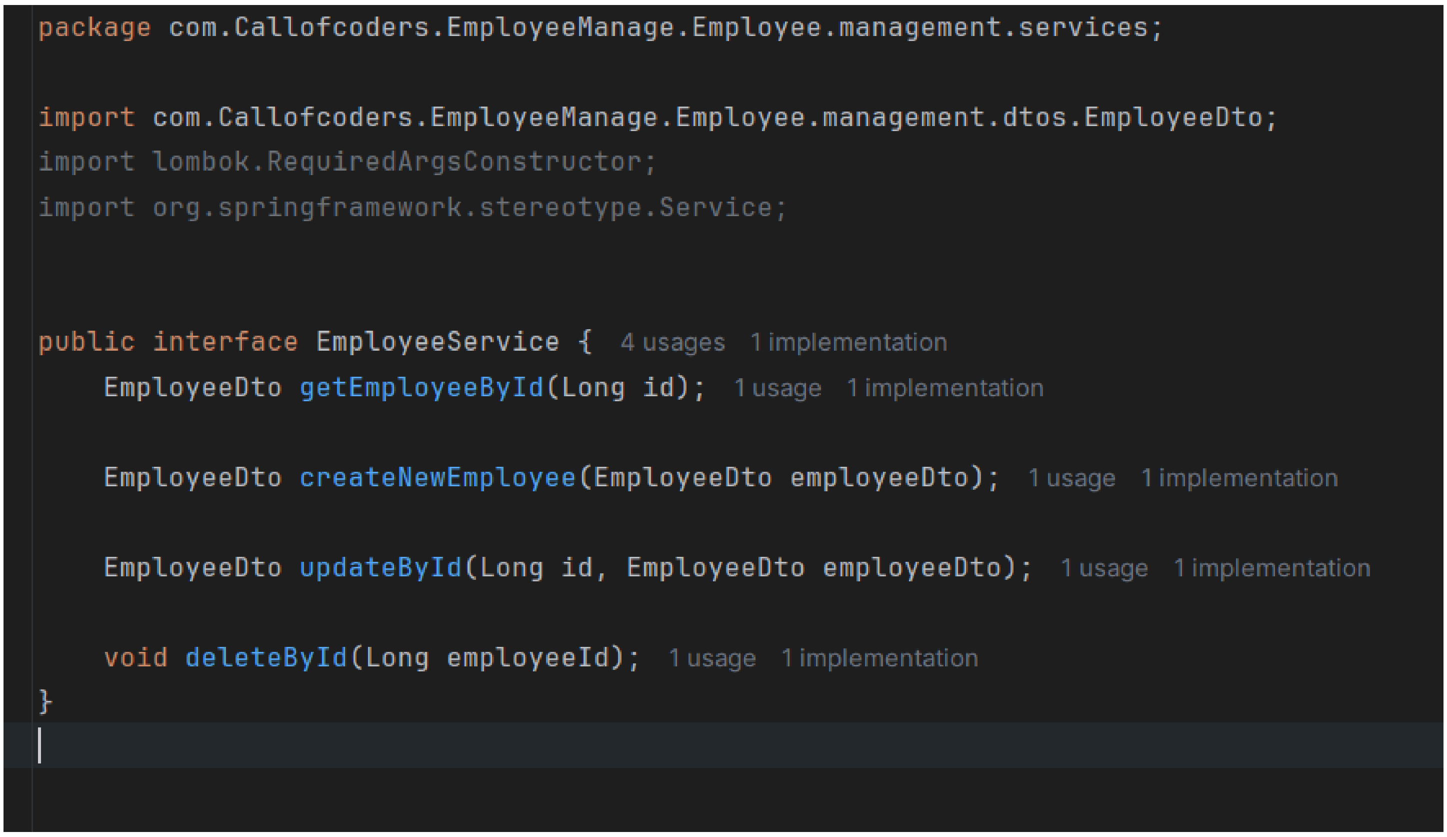
- HTTP Status: 204 No Content
- No body (which is fine for DELETE)

This is a builder pattern used here:

```
ResponseEntity.noContent() → returns builder
.build() → creates the final ResponseEntity<Void>
```

10. Service Layer

Interface



```
package com.Callofcoders.EmployeeManage.Employee.management.services;

import com.Callofcoders.EmployeeManage.Employee.management.dtos.EmployeeDto;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;


public interface EmployeeService {  4 usages   1 implementation
    EmployeeDto getEmployeeById(Long id);  1 usage   1 implementation

    EmployeeDto createNewEmployee(EmployeeDto employeeDto);  1 usage   1 implementation

    EmployeeDto updateById(Long id, EmployeeDto employeeDto);  1 usage   1 implementation

    void deleteById(Long employeeId);  1 usage   1 implementation
}
```

This is your **EmployeeService interface**, and it defines the contract (i.e., method signatures) that your service layer must implement. Let's break it down in a clean and complete explanation:

✅ What is an Interface in Spring?

In Spring (and in Java in general), an interface is used to **declare** methods without implementing them. The **implementation class** (like EmployeeServiceImpl) will provide the actual logic for each method.

This pattern promotes:

- **Loose coupling** (your controller depends only on the interface, not the implementation),
- **Testability** (you can mock the interface easily),
- **Clean architecture**.

🧾 Your EmployeeService Interface Explained

public interface EmployeeService {

This is a **public service interface** that will be implemented by a class like EmployeeServiceImpl.

1. EmployeeDto getEmployeeById(Long id);

- 📌 **Purpose**: Fetch a single employee by their ID.

- 📤 **Returns**: An EmployeeDto (used as a response object).
- 💬 Example use:
- employeeService.getEmployeeById(1L);

2. EmployeeDto createNewEmployee(EmployeeDto employeeDto);

- 📌 **Purpose**: Create a new employee using the provided DTO.
- 📥 **Takes**: An EmployeeDto containing data (like name, email, salary).
- 📤 **Returns**: The newly created EmployeeDto (with ID populated).
- 💬 Example use:
- employeeService.createNewEmployee(dto);

3. EmployeeDto updateById(Long id, EmployeeDto employeeDto);

- 📌 **Purpose**: Update an existing employee by ID with new details.
- 📥 **Takes**:
  - Long id: The ID of the employee to update.
  - EmployeeDto: New data to update.
- 📤 **Returns**: The updated EmployeeDto.
- 💬 Example use:
- employeeService.updateById(1L, updatedDto);

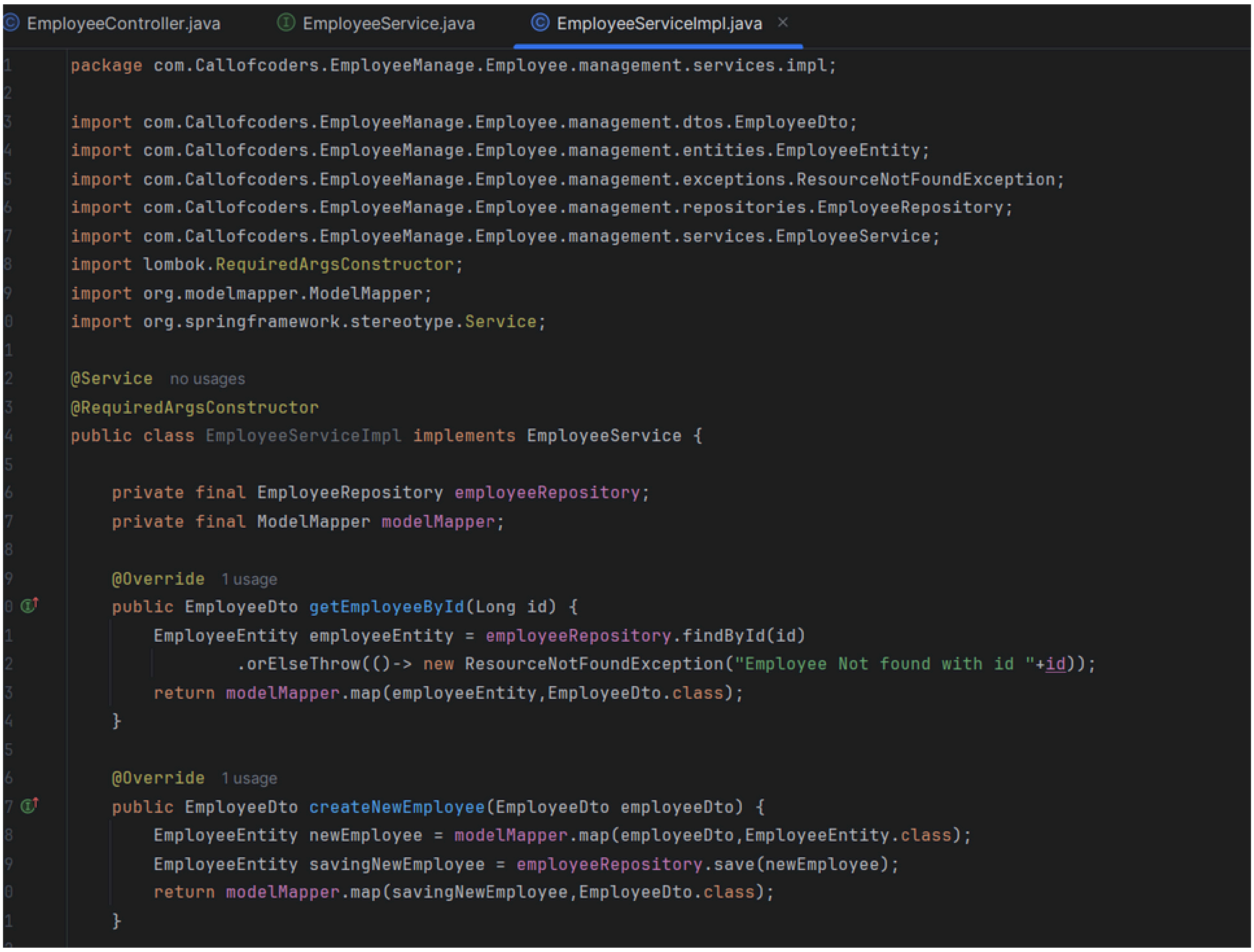
4. void deleteById(Long employeeId);

- 📌 **Purpose**: Delete an employee by their ID.
- 📥 **Takes**: The ID of the employee to delete.
- ❌ **Returns**: Nothing (void), just performs the action.
- 💬 Example use:
- employeeService.deleteById(1L);

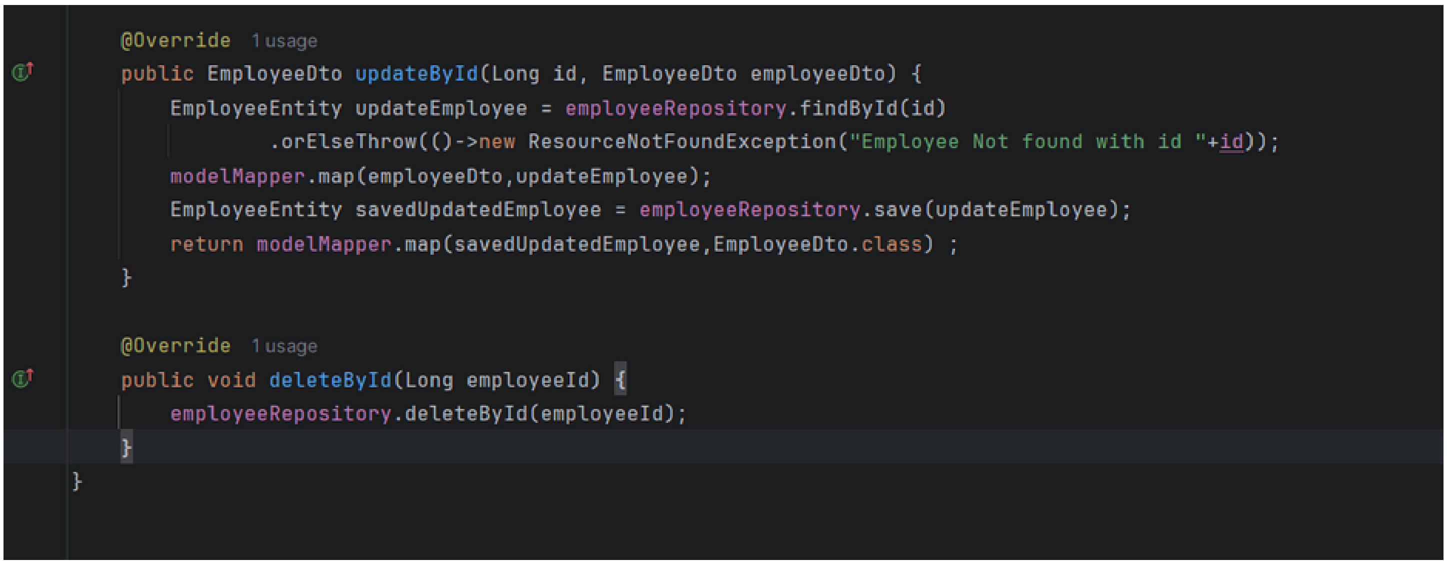
✅ Summary Table

Method	Purpose	Returns
getEmployeeById(Long id)	Get employee by ID	EmployeeDto
createNewEmployee(EmployeeDto)	Create a new employee	EmployeeDto
updateById(Long id, EmployeeDto)	Update an employee by ID	EmployeeDto
deleteById(Long employeeId)	Delete an employee by ID	void

The service layer contains business logic and interacts with the repository layer.



```java
package com.Callofcoders.EmployeeManage.Employee.management.services.impl;

import com.Callofcoders.EmployeeManage.Employee.management.dtos.EmployeeDto;
import com.Callofcoders.EmployeeManage.Employee.management.entities.EmployeeEntity;
import com.Callofcoders.EmployeeManage.Employee.management.exceptions.ResourceNotFoundException;
import com.Callofcoders.EmployeeManage.Employee.management.repositories.EmployeeRepository;
import com.Callofcoders.EmployeeManage.Employee.management.services.EmployeeService;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;

@Service  no usages
@RequiredArgsConstructor
public class EmployeeServiceImpl implements EmployeeService {

    private final EmployeeRepository employeeRepository;
    private final ModelMapper modelMapper;

    @Override  1 usage
    public EmployeeDto getEmployeeById(Long id) {
        EmployeeEntity employeeEntity = employeeRepository.findById(id)
                .orElseThrow(()-> new ResourceNotFoundException("Employee Not found with id "+id));
        return modelMapper.map(employeeEntity,EmployeeDto.class);
    }

    @Override  1 usage
    public EmployeeDto createNewEmployee(EmployeeDto employeeDto) {
        EmployeeEntity newEmployee = modelMapper.map(employeeDto,EmployeeEntity.class);
        EmployeeEntity savingNewEmployee = employeeRepository.save(newEmployee);
        return modelMapper.map(savingNewEmployee,EmployeeDto.class);
    }
```



```java
    @Override  1 usage
    public EmployeeDto updateById(Long id, EmployeeDto employeeDto) {
        EmployeeEntity updateEmployee = employeeRepository.findById(id)
                .orElseThrow(()->new ResourceNotFoundException("Employee Not found with id "+id));
        modelMapper.map(employeeDto,updateEmployee);
        EmployeeEntity savedUpdatedEmployee = employeeRepository.save(updateEmployee);
        return modelMapper.map(savedUpdatedEmployee,EmployeeDto.class) ;
    }

    @Override  1 usage
    public void deleteById(Long employeeId) {
        employeeRepository.deleteById(employeeId);
    }
}
```

This is your fully implemented **EmployeeServiceImpl** class. It provides actual logic for all the methods declared in the EmployeeService interface. Let's walk through it method by method, explain the annotations, and highlight best practices you've followed.

✅ Annotations & Dependencies

@Service

Marks this class as a **Spring service bean**, so Spring can detect and manage it via component scanning.

@RequiredArgsConstructor

A **Lombok annotation** that:

- Automatically creates a constructor with final fields.
- This enables dependency injection without writing boilerplate code.

Dependencies Injected:

```java
private final EmployeeRepository employeeRepository;
private final ModelMapper modelMapper;
```

These are injected via constructor thanks to @RequiredArgsConstructor.

✅ Method-wise Breakdown

1. getEmployeeById(Long id)

```
EmployeeEntity employeeEntity = employeeRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Employee Not found with id " + id));
return modelMapper.map(employeeEntity, EmployeeDto.class);
```

- 📌 **Fetches employee** by ID from DB.
- ❗ Throws custom exception ResourceNotFoundException if not found.
- 🔄 Uses ModelMapper to convert EmployeeEntity → EmployeeDto.

2. createNewEmployee(EmployeeDto employeeDto)

```
EmployeeEntity newEmployee = modelMapper.map(employeeDto, EmployeeEntity.class);
EmployeeEntity savingNewEmployee = employeeRepository.save(newEmployee);
return modelMapper.map(savingNewEmployee, EmployeeDto.class);
```

- 📌 Converts incoming DTO → Entity.
- 🗃️ Saves entity to DB.
- 🔄 Returns saved data back as DTO (with generated ID, etc.).

3. updateById(Long id, EmployeeDto employeeDto)

```
EmployeeEntity updateEmployee = employeeRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Employee Not found with id " + id));
modelMapper.map(employeeDto, updateEmployee);
EmployeeEntity savedUpdatedEmployee = employeeRepository.save(updateEmployee);
return modelMapper.map(savedUpdatedEmployee, EmployeeDto.class);
```

- 📌 First fetches entity by ID.
- 🔄 Uses ModelMapper to **map incoming DTO over existing entity** (smartly preserves ID and unchanged fields).
- 💾 Saves the updated entity and returns updated DTO.

4. deleteById(Long employeeId)

```
employeeRepository.deleteById(employeeId);
```

- ❌ Deletes employee by ID directly.
- 🟡 Suggestion: For robustness, check if employee exists first:

```
EmployeeEntity emp = employeeRepository.findById(employeeId)
        .orElseThrow(() -> new ResourceNotFoundException("Employee not found"));
employeeRepository.delete(emp);
```

✅ Summary

Method	Action	Exception Safe?	DTO ↔ Entity Mapping	Comments
getEmployeeById	Get by ID	✅ Yes	Entity → DTO	Good
createNewEmployee	Create	✅ Yes	DTO → Entity → DTO	Good
updateById	Update	✅ Yes	DTO → Entity (partial)	Very clean
deleteById	Delete	❌ No	N/A	Add exists check for safety

✅ Suggestions for Improvements

1. **Exception Handling on Delete**:
   Add a check before deleting to avoid EmptyResultDataAccessException.
2. **Logging** (Optional):
   Add @Slf4j (Lombok) for internal logs:

```
log.info("Employee created with ID {}", savedEmployee.getId());
```

1. **Validation**:
   Ensure EmployeeDto has validation annotations like:

```
@NotBlank, @Email, @Min, @Max, etc.
```

1. **ResponseEntity Support**:
   Your controller (not the service) should handle ResponseEntity, not this layer. Service should just return the data.

- **ModelMapper** is used to map EmployeeDto to Employee and vice versa.
- This layer abstracts the business logic from the controller, making the code cleaner and more modular.

11. Crete new Resource not found exception class

```java
public class ResourceNotFoundException extends RuntimeException{
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

🔍 What It Is:

- ResourceNotFoundException is a **custom exception** that extends Java's built-in RuntimeException.
- It is used to indicate that a requested resource (e.g., an Employee) **does not exist** in your system (database).

🧠 Why Use It?

Instead of using a generic exception (like NullPointerException or IllegalArgumentException), you define your **own specific exception** to:

1. **Make your code more readable and maintainable.**
2. **Handle errors gracefully in REST APIs** (e.g., return 404 Not Found).
3. **Use meaningful messages** to clients.

💡 How It Works:

Suppose this is your service:

```java
@Override
public EmployeeDto getEmployeeById(Long id) {
    EmployeeEntity employee = employeeRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Employee not found with ID: " + id));
    return modelMapper.map(employee, EmployeeDto.class);
}
```

- If the findById(id) returns empty, it throws your ResourceNotFoundException.
- This exception can then be **caught globally** using a @ControllerAdvice to return a clean error response like:

```
{
  "timestamp": "2025-08-02T11:35:00",
  "status": 404,
  "error": "Not Found",
  "message": "Employee not found with ID: 10"
}
```

⚙️ Constructor Details:

```
public ResourceNotFoundException(String message){
    super(message);
}
```

- It calls the constructor of RuntimeException with your custom message.
- That message will be available using ex.getMessage() when the exception is caught.

📦 Summary:

Component	Explanation
extends RuntimeException	Inherits unchecked exception behavior.
ResourceNotFoundException	Custom error for "not found" cases.
super(message)	Passes a custom error message.
Use case	Typically thrown when DB query returns empty.
Benefit	Makes API more clear, maintainable, and client-friendly.

12. Exception Handling with @RestControllerAdvice

Handling exceptions in RESTful APIs is important for returning meaningful error messages. The @RestControllerAdvice annotation handles exceptions globally.

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException ex) {
        return ResponseEntity.notFound().build();
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<?> handleRuntimeException(RuntimeException ex) {
        return ResponseEntity.internalServerError().build();
    }
}
```

This advice handles ResourceNotFoundException and RuntimeException, ensuring that the client receives proper error responses.

This is your **global exception handler** class in Spring Boot. It catches exceptions thrown from anywhere in your controller/service layers and returns appropriate HTTP responses.

Let’s go through each part step by step:

🧠 Explanation:

🔹 @RestControllerAdvice

- This annotation combines @ControllerAdvice + @ResponseBody.
- It tells Spring that this class will handle **exceptions globally**, and responses should be in **JSON format** (not HTML).

🔹 @ExceptionHandler(ResourceNotFoundException.class)

- This method will catch any time your custom ResourceNotFoundException is thrown.

```java
public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException ex)
```

- ResponseEntity.notFound().build() returns:
  - HTTP Status: 404 Not Found
  - No response body (just empty)

✅ It works—but you might want to return a message too (see improvements below).

🔹 @ExceptionHandler(RuntimeException.class)

- Catches any unhandled RuntimeException (which is a superclass of many exceptions like NullPointerException, etc.)

```
public ResponseEntity<?> handleRuntimeException(RuntimeException ex)
```

- Returns:
  - HTTP Status: 500 Internal Server Error
  - No body

✅ How It Works in Real Use

When your service throws:

```
throw new ResourceNotFoundException("Employee not found with ID: " + id);
```

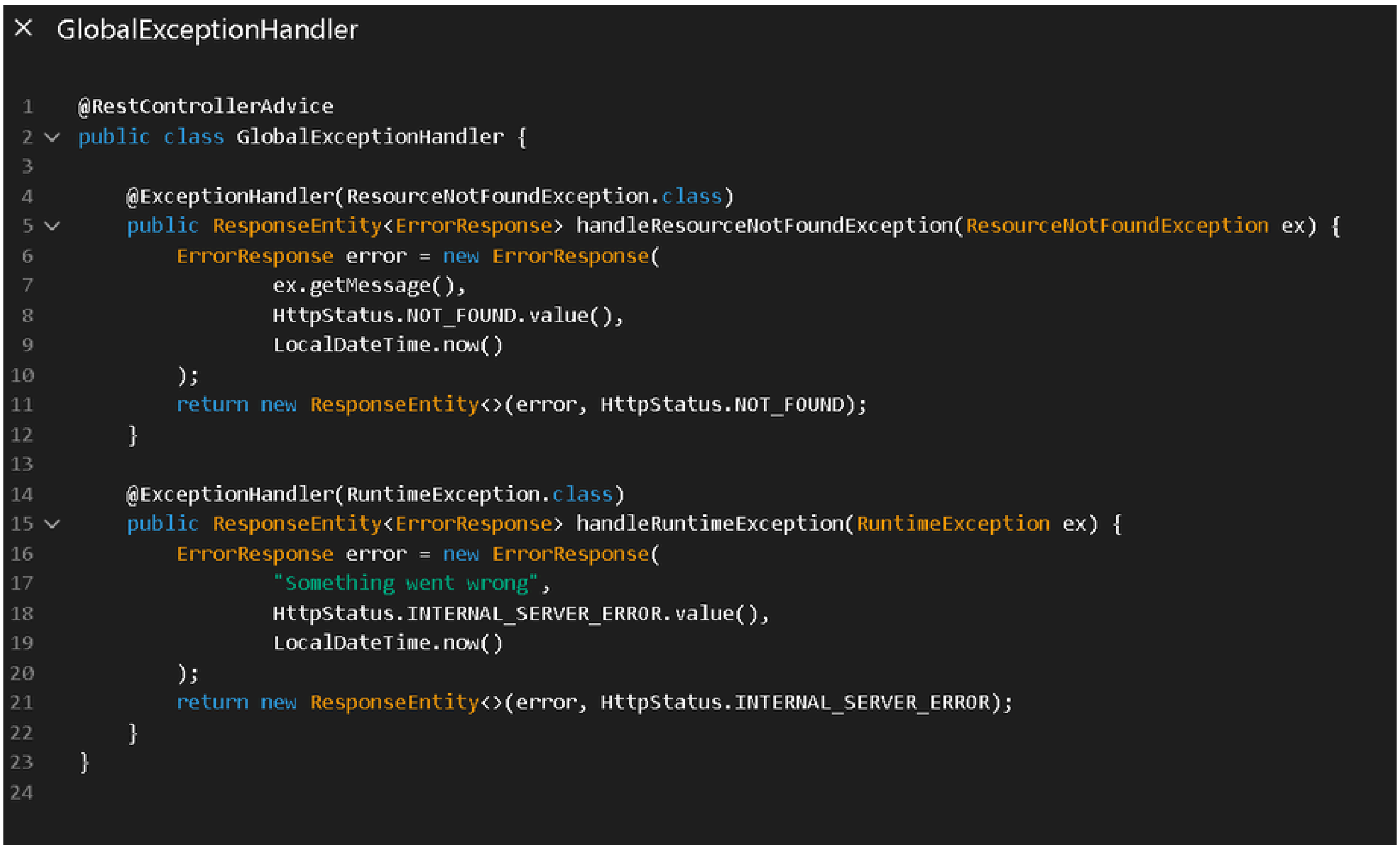
This class intercepts that exception and returns:

```
HTTP 404
(empty response body)
```

🔧 Suggested Improvement (Optional)

You can return a **structured error response** with message and timestamp:

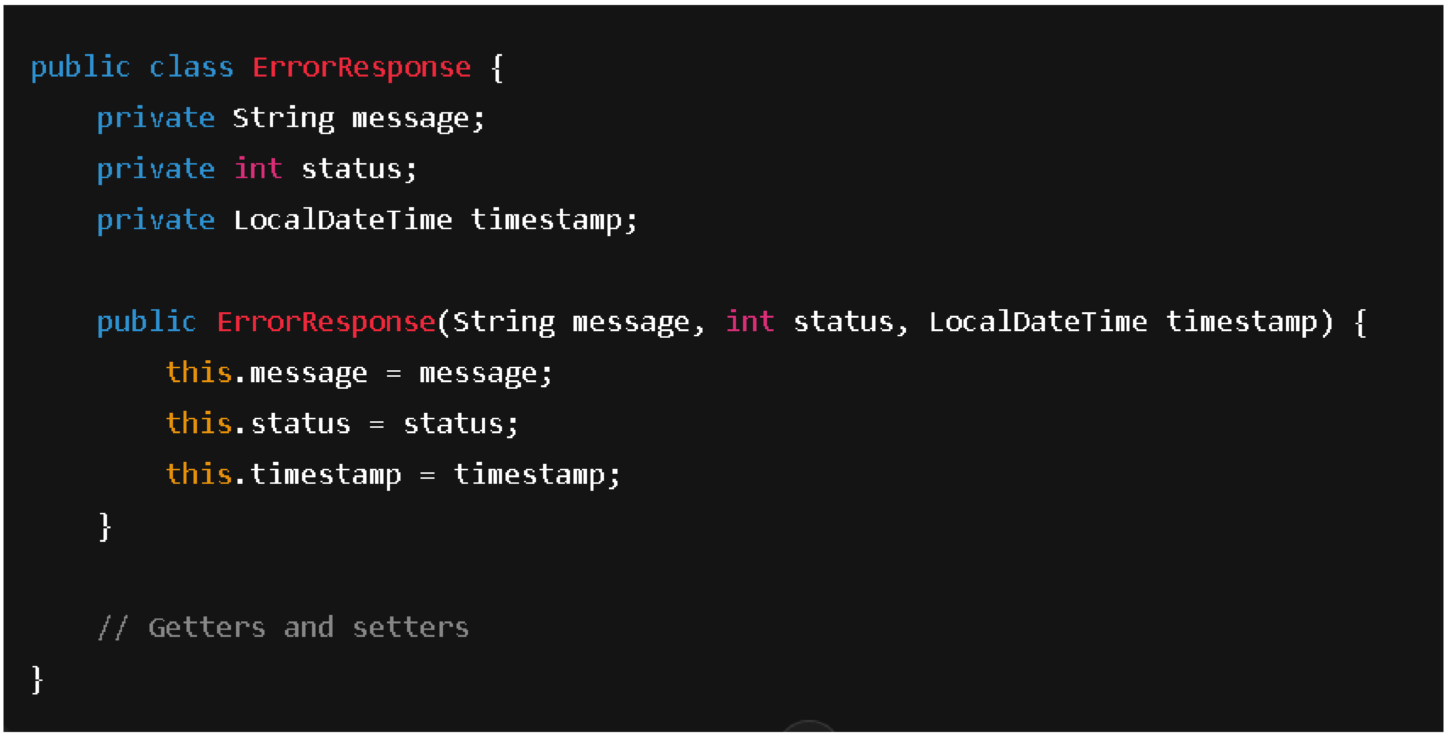
✅ Improved Version:



GlobalExceptionHandler

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(ResourceNotFoundException ex) {
        ErrorResponse error = new ErrorResponse(
                ex.getMessage(),
                HttpStatus.NOT_FOUND.value(),
                LocalDateTime.now()
        );
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<ErrorResponse> handleRuntimeException(RuntimeException ex) {
        ErrorResponse error = new ErrorResponse(
                "Something went wrong",
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                LocalDateTime.now()
        );
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

✅ ErrorResponse class:



```java
public class ErrorResponse {
    private String message;
    private int status;
    private LocalDateTime timestamp;

    public ErrorResponse(String message, int status, LocalDateTime timestamp) {
        this.message = message;
        this.status = status;
        this.timestamp = timestamp;
    }

    // Getters and setters
}
```

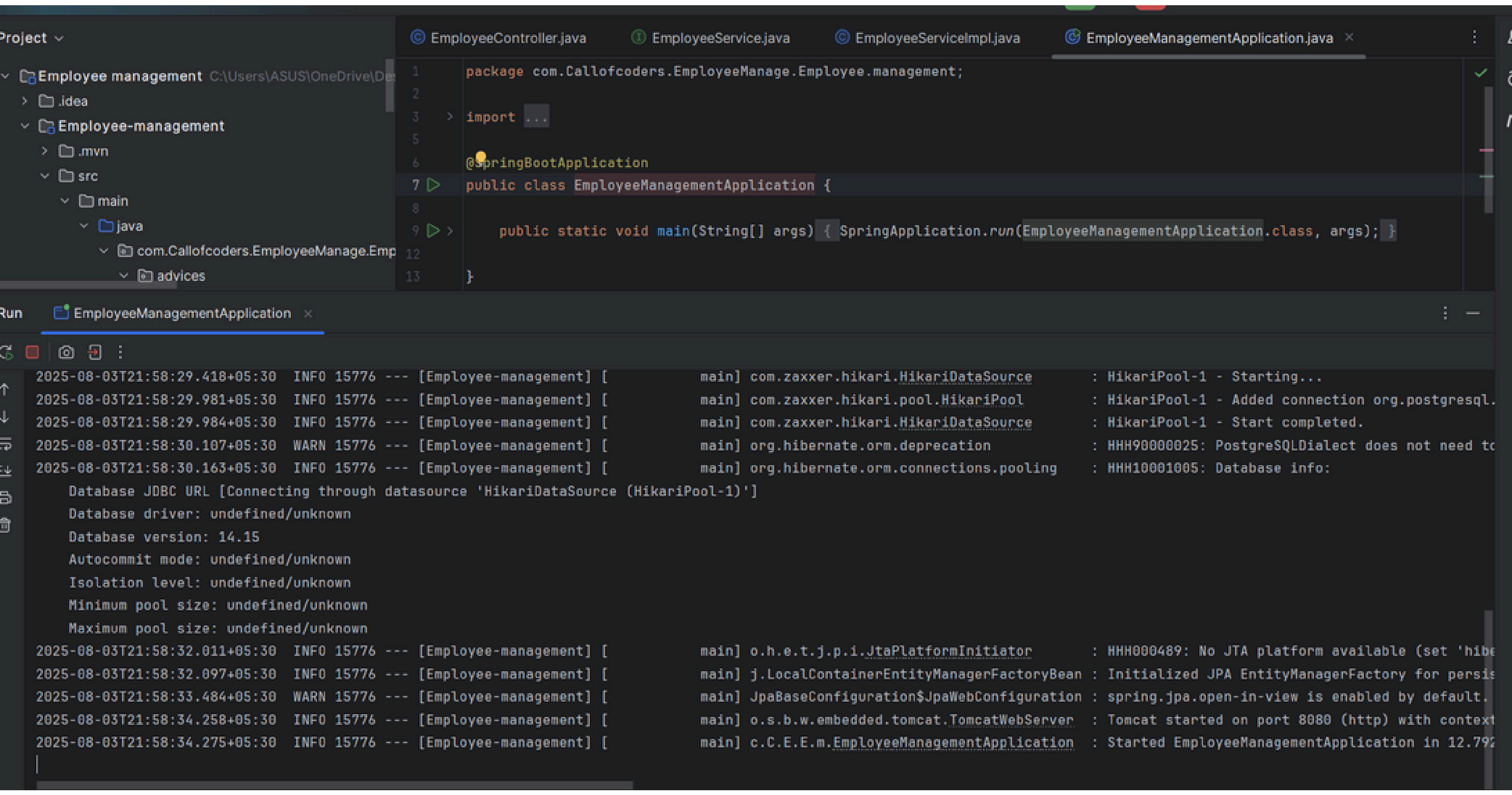
🔽 Sample Response:

```
{
  "message": "Employee not found with ID: 5",
  "status": 404,
  "timestamp": "2025-08-02T12:10:30"
}
```

12. Now Open post man to test our apis

1. Create Employee

Run your application, after running you can see in console the hibernate DDL queries to create the employee table.



```java
package com.Callofcoders.EmployeeManage.Employee.management;

import ...

@SpringBootApplication
public class EmployeeManagementApplication {

    public static void main(String[] args) { SpringApplication.run(EmployeeManagementApplication.class, args); }

}
```

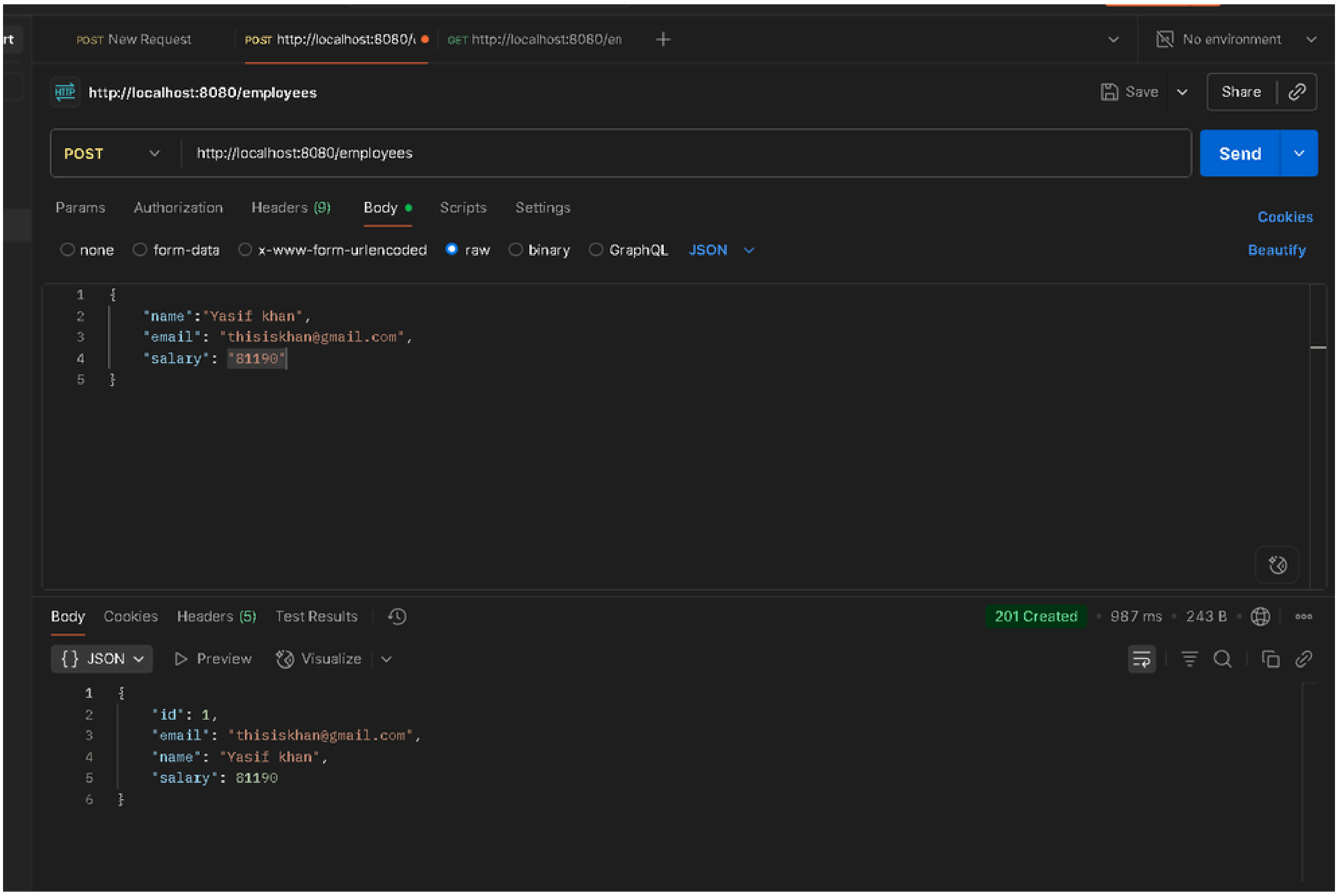
```
2025-08-03T21:58:29.418+05:30  INFO 15776 --- [Employee-management] [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2025-08-03T21:58:29.981+05:30  INFO 15776 --- [Employee-management] [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection org.postgresql.
2025-08-03T21:58:29.984+05:30  INFO 15776 --- [Employee-management] [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2025-08-03T21:58:30.107+05:30  WARN 15776 --- [Employee-management] [           main] org.hibernate.orm.deprecation            : HHH90000025: PostgreSQLDialect does not need to
2025-08-03T21:58:30.163+05:30  INFO 15776 --- [Employee-management] [           main] org.hibernate.orm.connections.pooling    : HHH10001005: Database info:
	Database JDBC URL [Connecting through datasource 'HikariDataSource (HikariPool-1)']
	Database driver: undefined/unknown
	Database version: 14.15
	Autocommit mode: undefined/unknown
	Isolation level: undefined/unknown
	Minimum pool size: undefined/unknown
	Maximum pool size: undefined/unknown
2025-08-03T21:58:32.011+05:30  INFO 15776 --- [Employee-management] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available (set 'hibe
2025-08-03T21:58:32.097+05:30  INFO 15776 --- [Employee-management] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persis
2025-08-03T21:58:33.484+05:30  WARN 15776 --- [Employee-management] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default.
2025-08-03T21:58:34.258+05:30  INFO 15776 --- [Employee-management] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context
2025-08-03T21:58:34.275+05:30  INFO 15776 --- [Employee-management] [           main] c.C.E.E.m.EmployeeManagementApplication  : Started EmployeeManagementApplication in 12.792
```

Project
Employee management C:\Users\ASUS\OneDrive\De
.idea
Employee-management
.mvn
src
main
java
com.Callofcoders.EmployeeManage.Emp
advices
EmployeeController.java
EmployeeService.java
EmployeeServiceImpl.java
EmployeeManagementApplication.java
Run
EmployeeManagementApplication

Employee table DDL query

Now go to post man and hit the create employee endpoint

Create Employee


POST New Request | POST http://localhost:8080/ | GET http://localhost:8080/en

No environment

http://localhost:8080/employees

Save | Share

POST http://localhost:8080/employees

Send

Params | Authorization | Headers (9) | Body | Scripts | Settings

Cookies

none | form-data | x-www-form-urlencoded | [✓ raw] | binary | GraphQL | JSON

Beautify

```
{
    "name":"Yasif khan",
    "email": "thisiskhan@gmail.com",
    "salary": "81190"
}
```

Body | Cookies | Headers (5) | Test Results

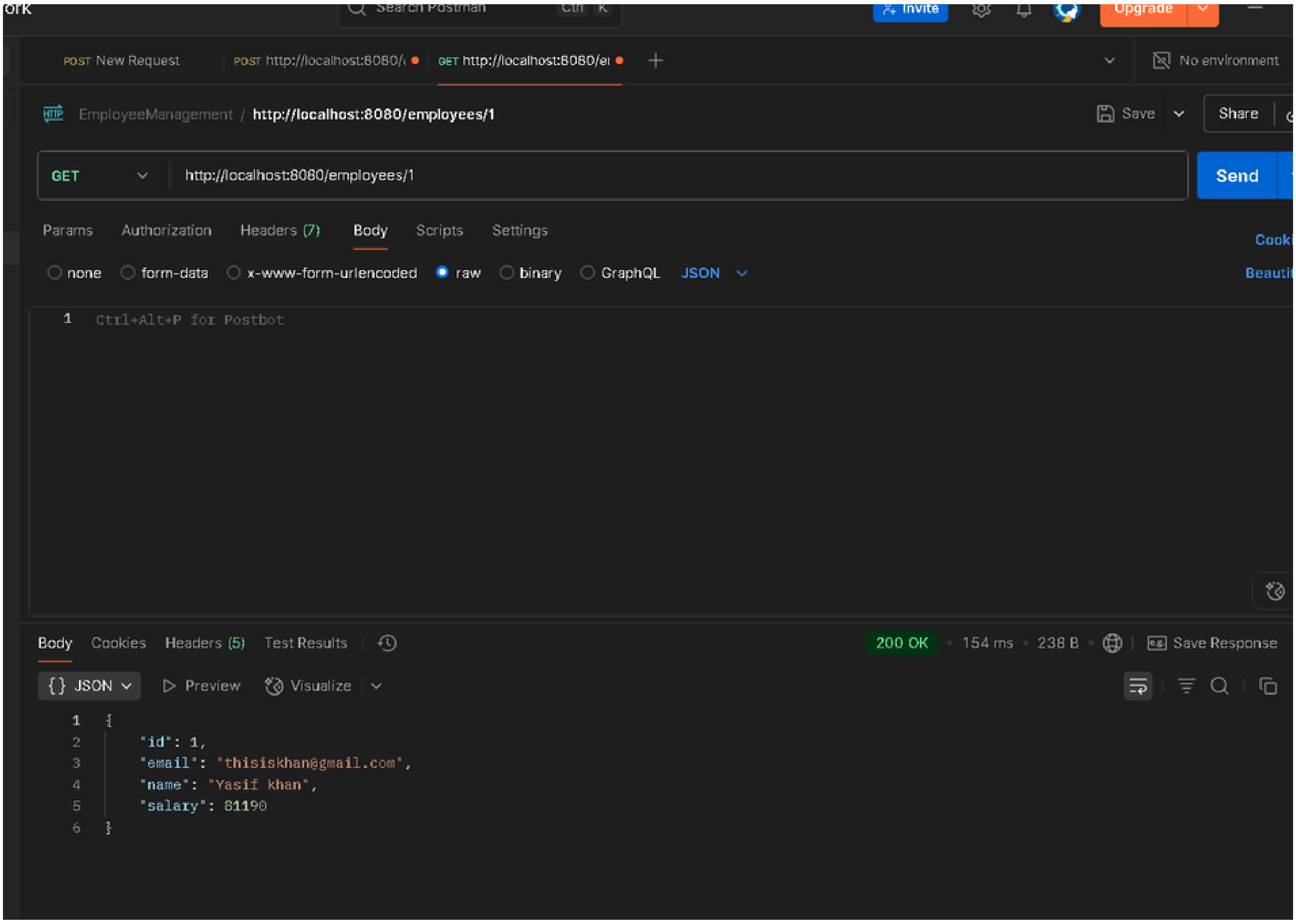
201 Created • 987 ms • 243 B

JSON | Preview | Visualize

```
{
    "id": 1,
    "email": "thisiskhan@gmail.com",
    "name": "Yasif khan",
    "salary": 81190
}
```

2. Get Employee by id

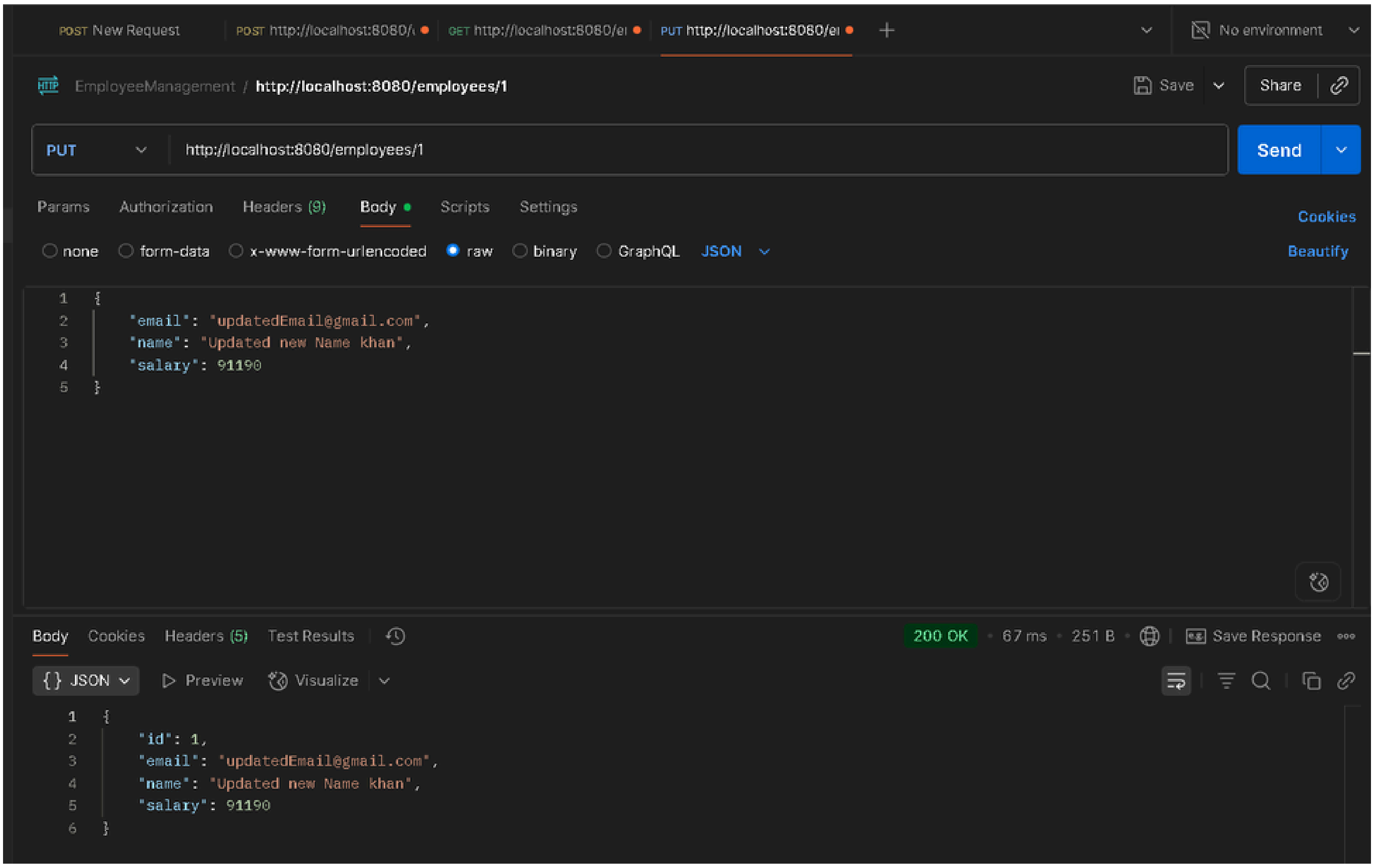
Get Employee by id


EmployeeManagement / http://localhost:8080/employees/1
GET http://localhost:8080/employees/1
Send
Params Authorization Headers (7) Body Scripts Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
200 OK • 154 ms • 238 B

```
{
    "id": 1,
    "email": "thisiskhan@gmail.com",
    "name": "Yasif khan",
    "salary": 81190
}
```

3. Update Employee by id

Update Employee by id



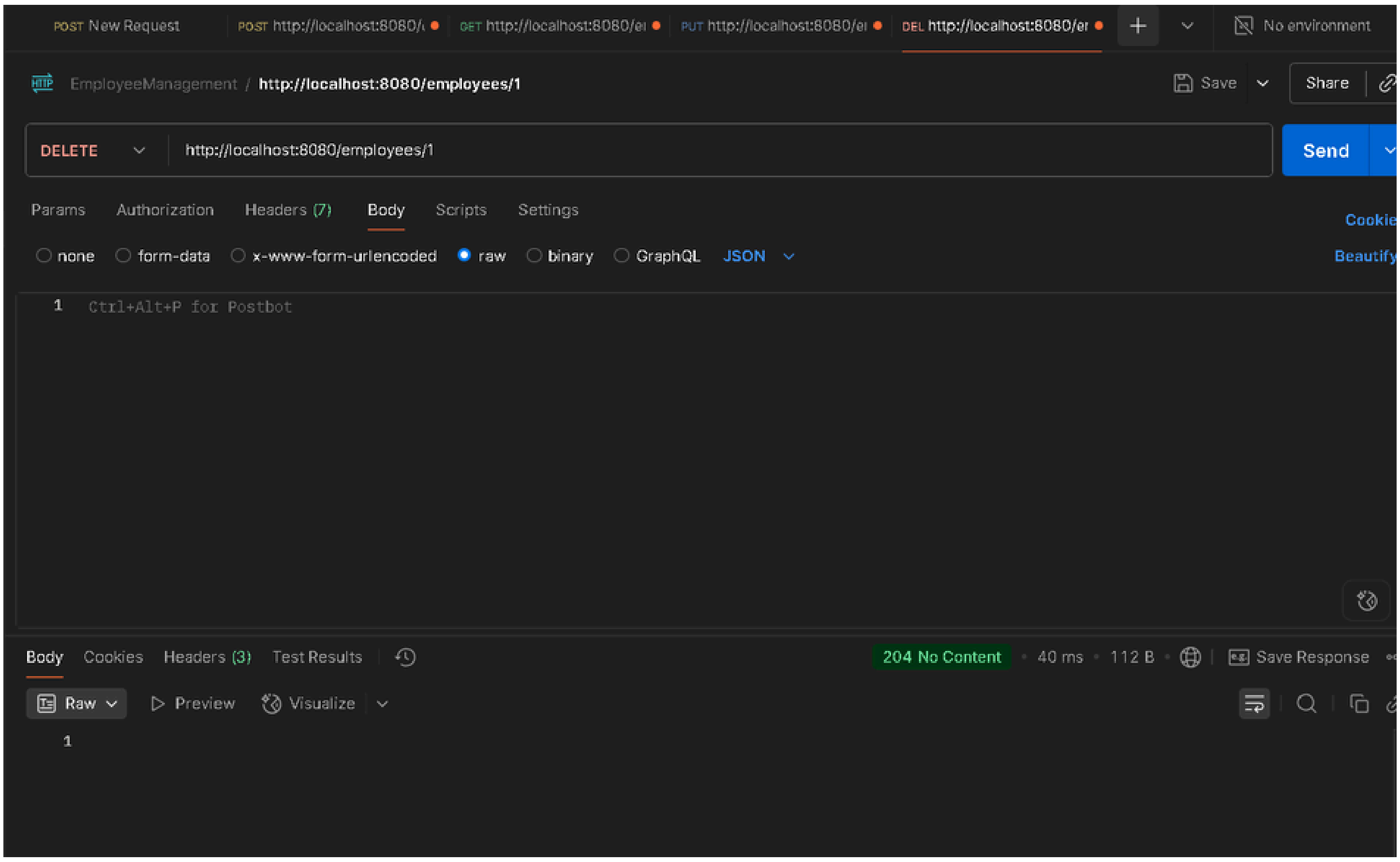
PUT http://localhost:8080/employees/1

```
{
    "email": "updatedEmail@gmail.com",
    "name": "Updated new Name khan",
    "salary": 91190
}
```

```
{
    "id": 1,
    "email": "updatedEmail@gmail.com",
    "name": "Updated new Name khan",
    "salary": 91190
}
```

4. Delete Employee by id

Delete Employee by id



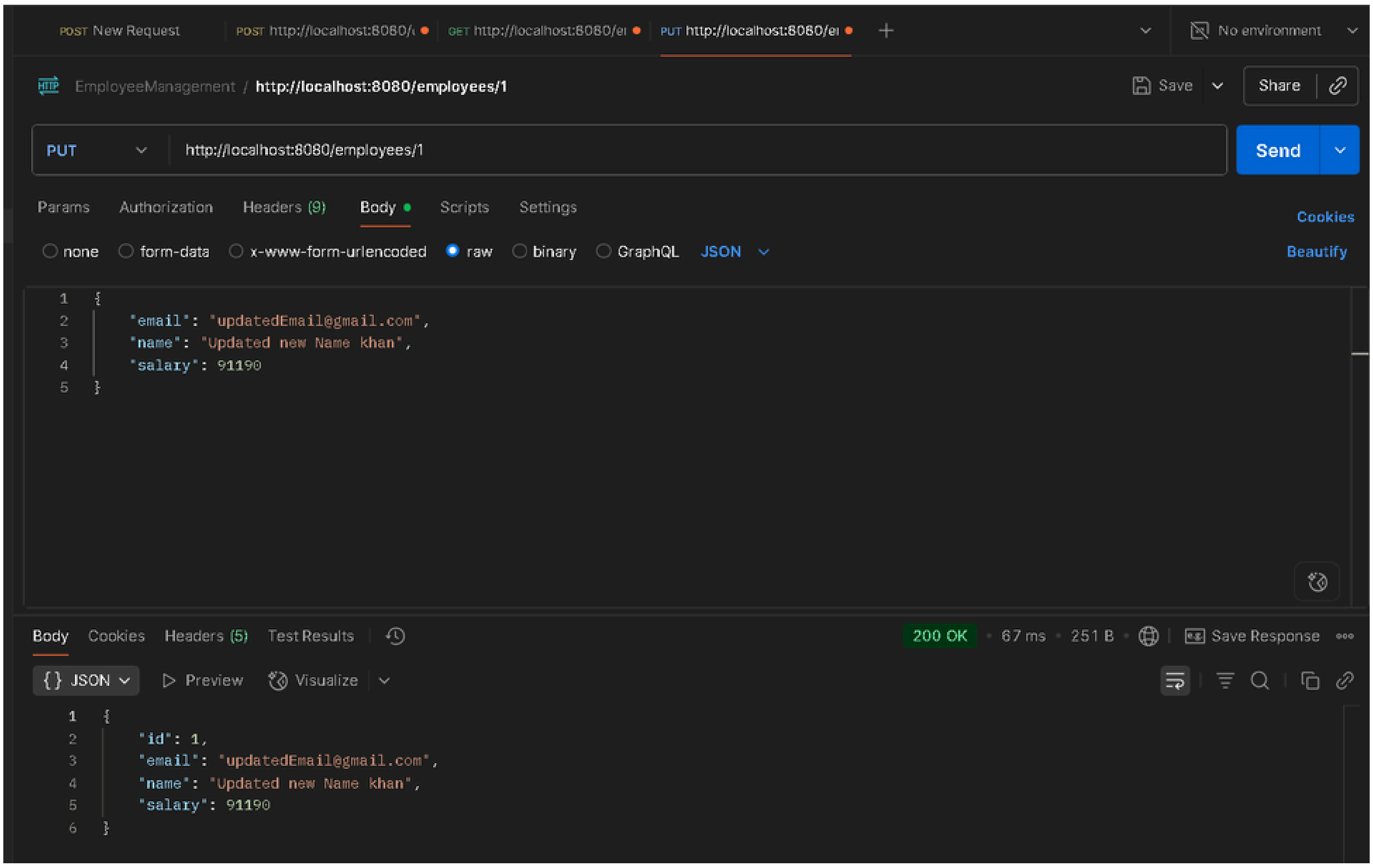
DELETE http://localhost:8080/employees/1

```
Ctrl+Alt+P for Postbot
```

In this article, we covered the essential steps for building a RESTful API in Spring Boot, including setting up a basic project, configuring a database, creating entity classes, and implementing CRUD operations through controllers and services. We also explored exception handling to make APIs more robust and user-friendly. By leveraging Spring Boot's powerful features and annotations, you can rapidly develop scalable and maintainable RESTful services.

problem I got when building this simple project

In The Update Method


EmployeeManagement / http://localhost:8080/employees/1

PUT http://localhost:8080/employees/1

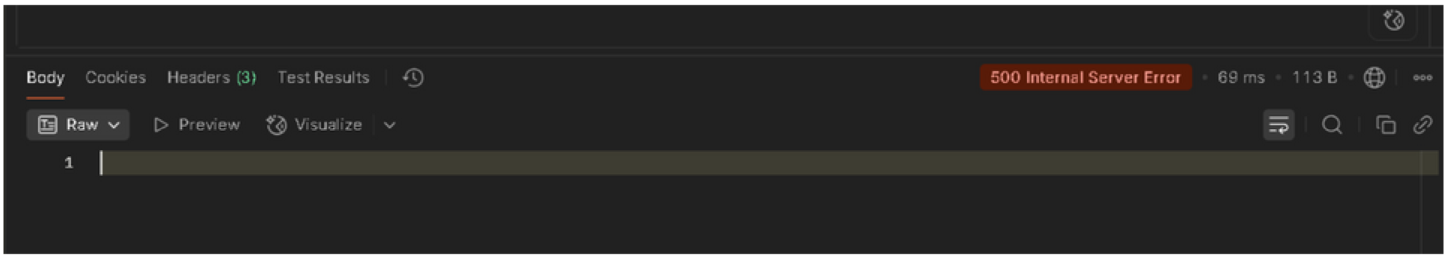
```
{
    "email": "updatedEmail@gmail.com",
    "name": "Updated new Name khan",
    "salary": 91190
}
```

200 OK · 67 ms · 251 B

```
{
    "id": 1,
    "email": "updatedEmail@gmail.com",
    "name": "Updated new Name khan",
    "salary": 91190
}
```

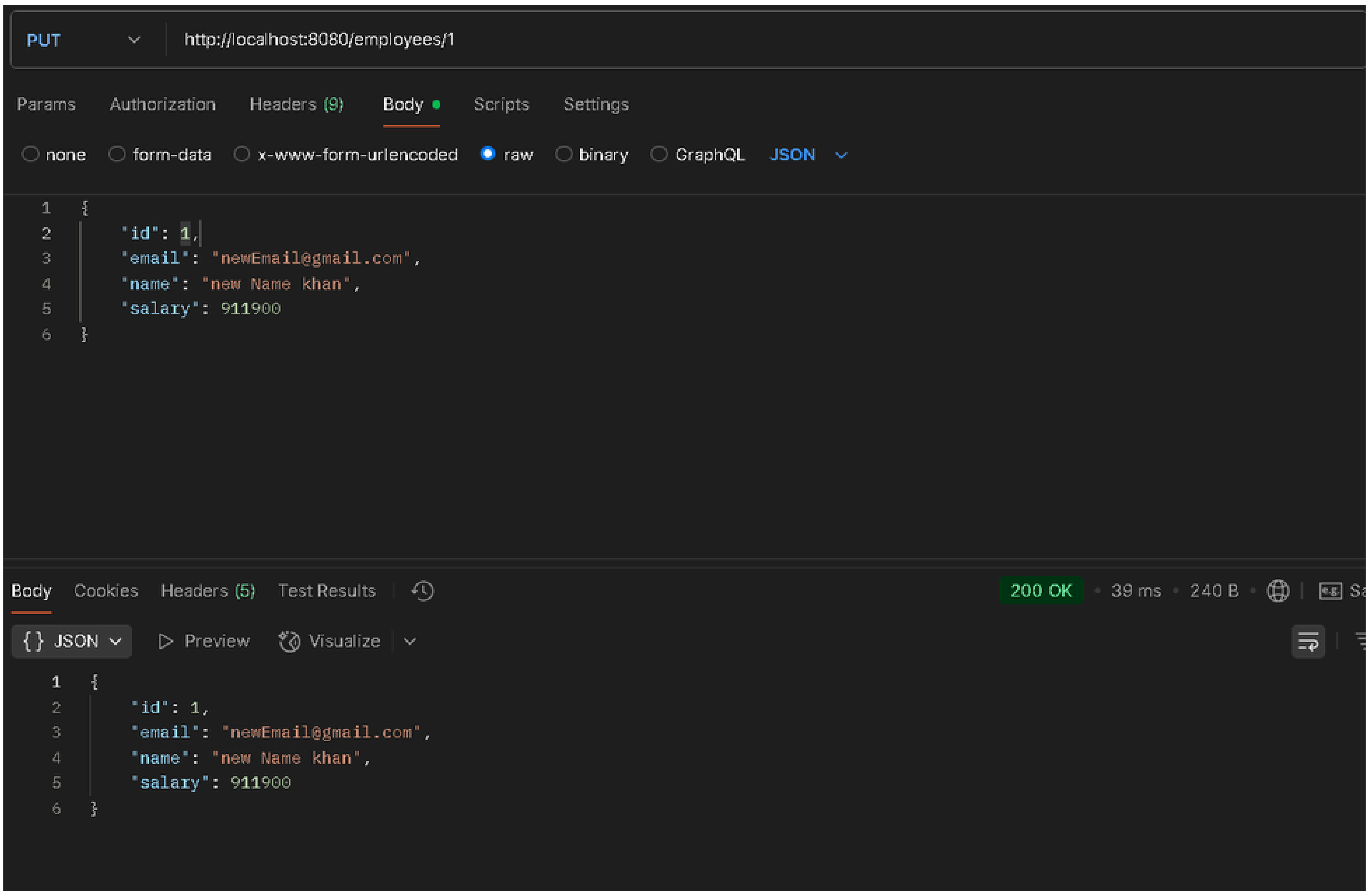
i Want like this above. Where i do not need to put id field in the json request body, and in response i want to get the output with the particular id and updated fields.

but i got like this


500 Internal Server Error · 69 ms · 113 B

if i did not give the id field in the json request body.

but if i add the id filed like this int the request body.



```
PUT http://localhost:8080/employees/1
```

```json
{
    "id": 1,
    "email": "newEmail@gmail.com",
    "name": "new Name khan",
    "salary": 911900
}
```

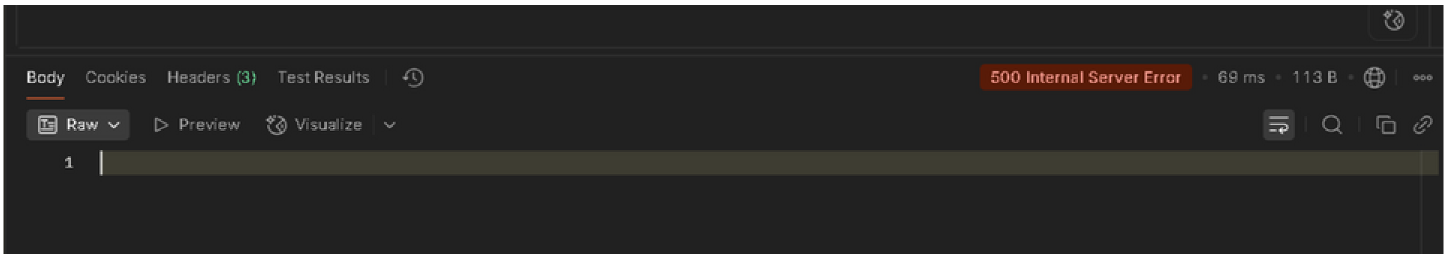
```
200 OK • 39 ms • 240 B
```

```json
{
    "id": 1,
    "email": "newEmail@gmail.com",
    "name": "new Name khan",
    "salary": 911900
}
```

it runs successfully. and update my fields with that id.

but here is to notice that i need to provide id in the json body along with url path also.

but if i already provide the id in the url path. why do need to add id field in the json body also



```
500 Internal Server Error • 69 ms • 113 B
```

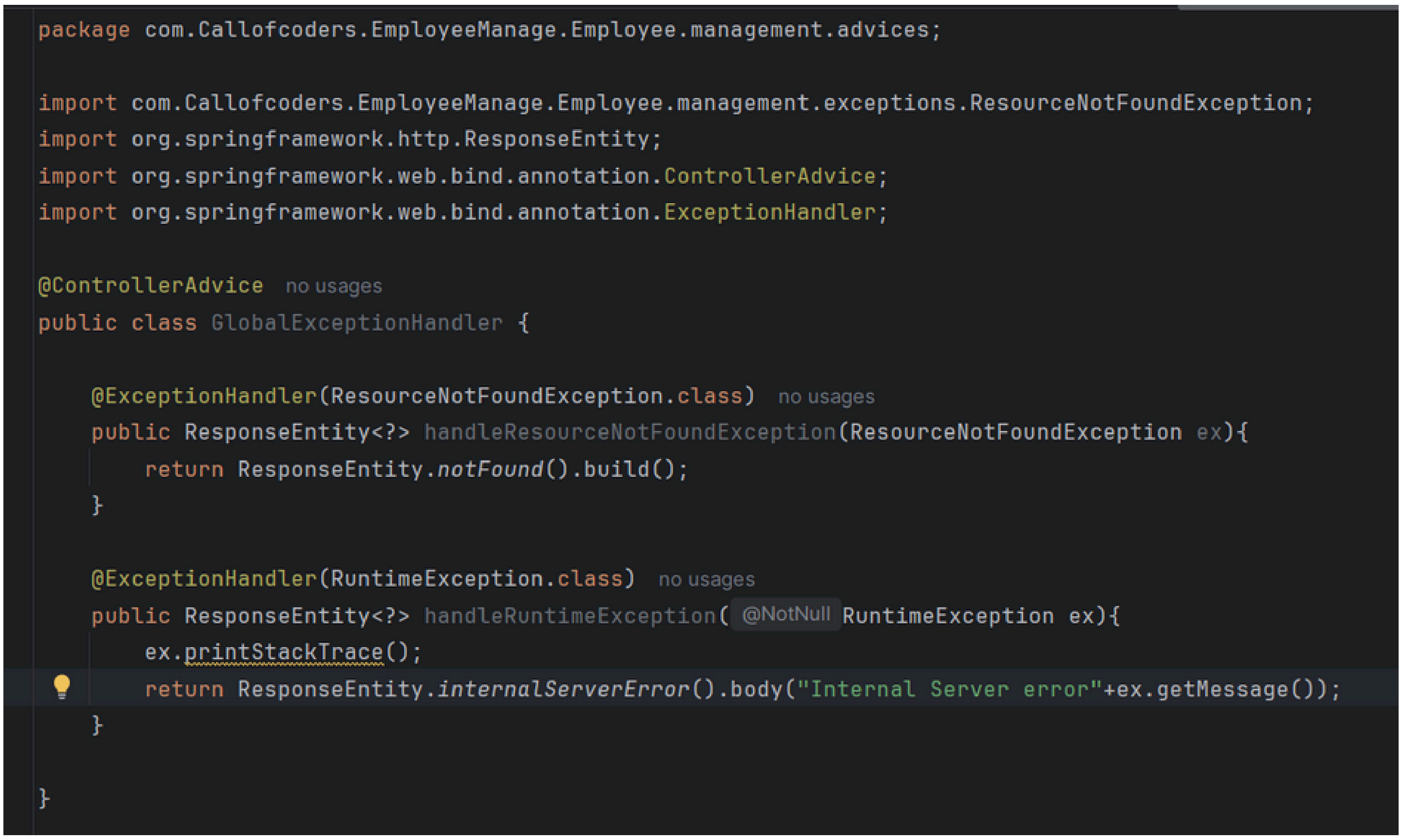
also i did not see any error message.

if you want to see the message. and know the what exact is the problem. update the GlobalExceptionHandler class like this.

Older Code

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException ex) {
        return ResponseEntity.notFound().build();
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<?> handleRuntimeException(RuntimeException ex) {
        return ResponseEntity.internalServerError().build();
    }
}
```

Updated Code



```
package com.Callofcoders.EmployeeManage.Employee.management.advices;

import com.Callofcoders.EmployeeManage.Employee.management.exceptions.ResourceNotFoundException;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice  no usages
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)  no usages
    public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException ex){
        return ResponseEntity.notFound().build();
    }

    @ExceptionHandler(RuntimeException.class)  no usages
    public ResponseEntity<?> handleRuntimeException( @NotNull RuntimeException ex){
        ex.printStackTrace();
        return ResponseEntity.internalServerError().body("Internal Server error"+ex.getMessage());
    }

}
```

What i do here.

```
ex.printStackTrace(); // ✅ Logs full error in console
```

Logs the **full stack trace** in the server console.

Useful for debugging — you can see **what line and what file** caused the exception.

✅ Only do this in development or debugging mode. For production, use logging frameworks like SLF4J or Logback.

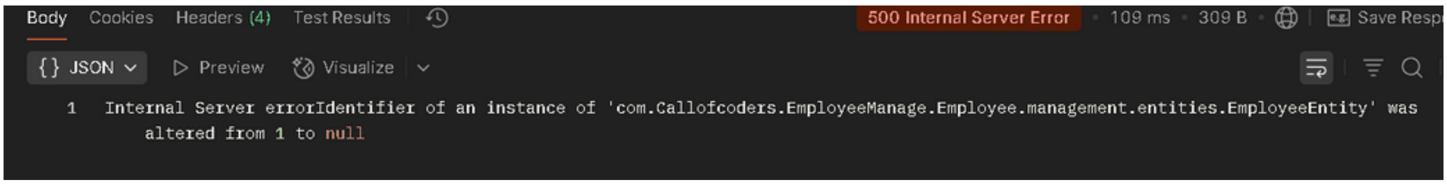
```java
return ResponseEntity.internalServerError().body("Internal Server Error: " + ex.getMessage());
```

Sends an HTTP **500 Internal Server Error** to the client.

.body(...) sets the message in the response body.

ex.getMessage() gives a short description of the error (e.g., "Employee not found").

Now i got the exact error message and what is the root cause of it. let's see



```
Internal Server errorIdentifier of an instance of 'com.Callofcoders.EmployeeManage.Employee.management.entities.EmployeeEntity' was
    altered from 1 to null
```

it says

❌ Identifier of an instance of 'EmployeeEntity' was altered from 3 to null

What does this mean?

✅ Meaning:

This error means your @Entity's **ID field was set to null** during update, and Hibernate doesn't allow changing/removing primary keys.

✅ Option 1: Use **ModelMapper with Strict Matching + Skip id**

You can **tell ModelMapper to ignore the id field** when mapping.

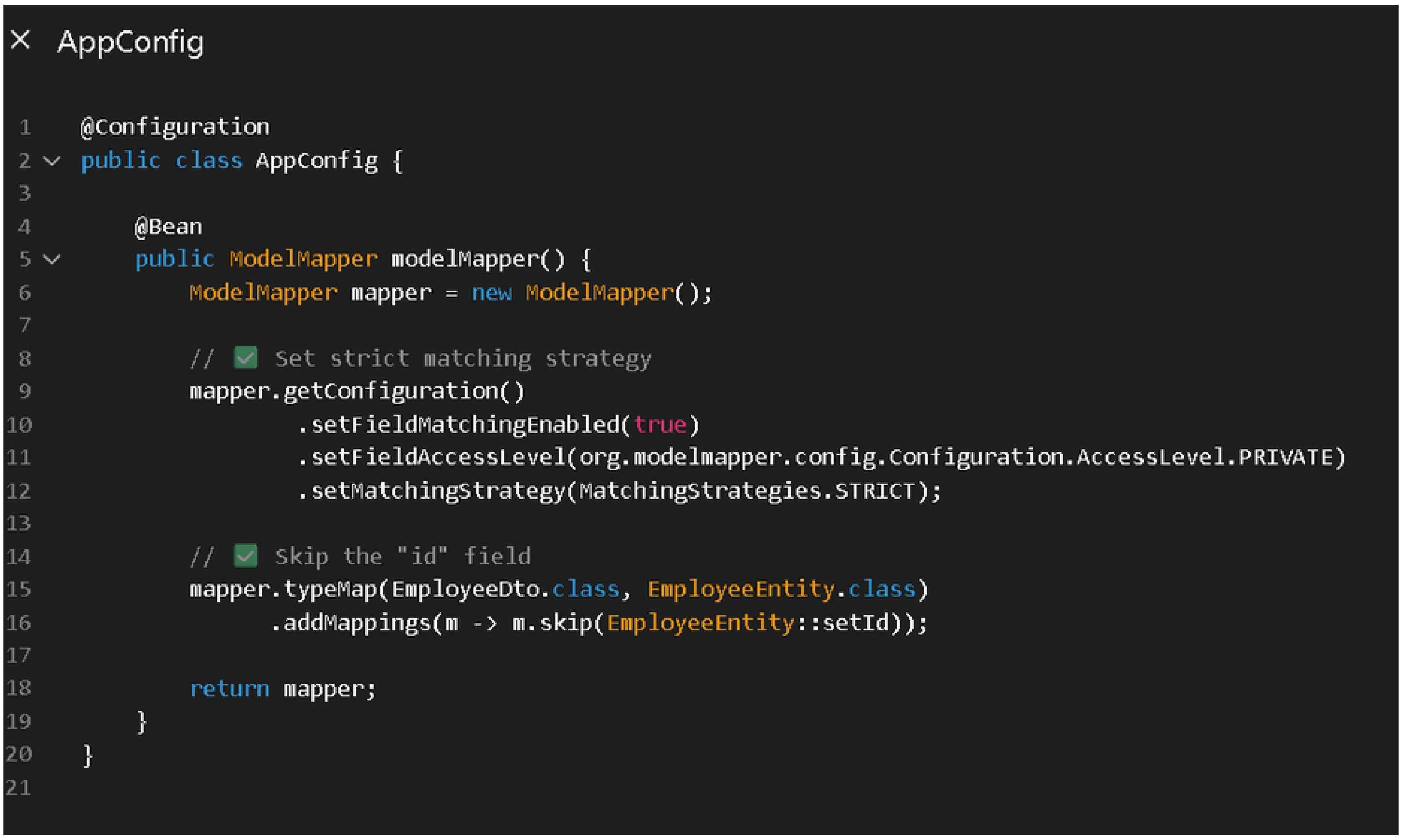
✅ Step-by-Step Fix

1. **Update your AppConfig to configure ModelMapper**

Older Code

```java
@Configuration
public class AppConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

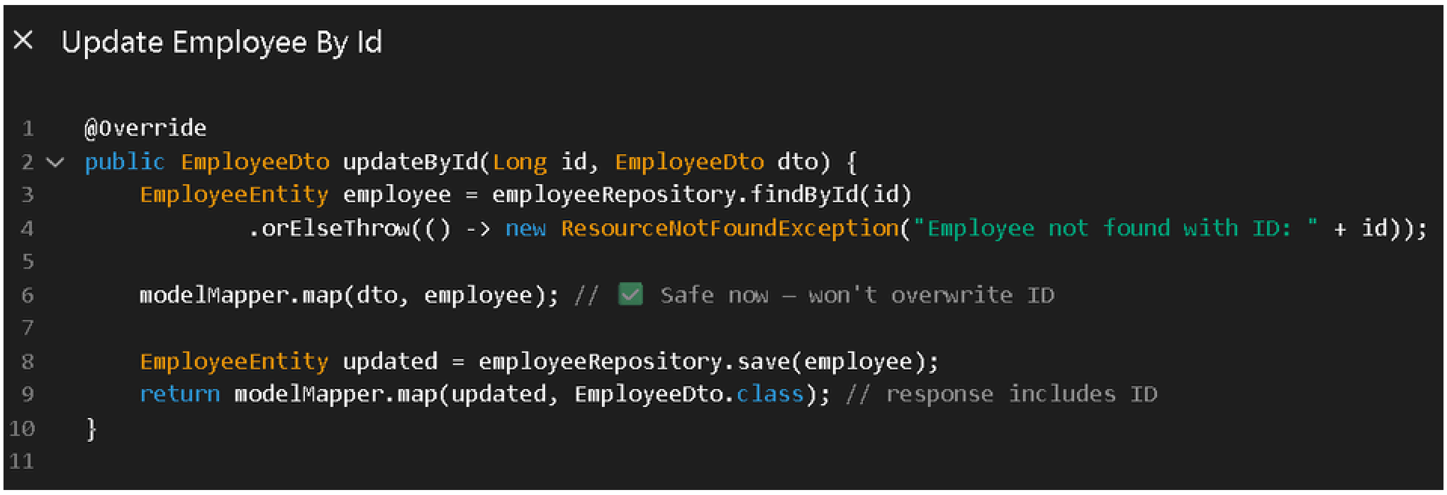
Updated Code



AppConfig

```java
@Configuration
public class AppConfig {

    @Bean
    public ModelMapper modelMapper() {
        ModelMapper mapper = new ModelMapper();

        // ✅ Set strict matching strategy
        mapper.getConfiguration()
                .setFieldMatchingEnabled(true)
                .setFieldAccessLevel(org.modelmapper.config.Configuration.AccessLevel.PRIVATE)
                .setMatchingStrategy(MatchingStrategies.STRICT);

        // ✅ Skip the "id" field
        mapper.typeMap(EmployeeDto.class, EmployeeEntity.class)
              .addMappings(m -> m.skip(EmployeeEntity::setId));

        return mapper;
    }
}
```

2. Keep your service method like this:

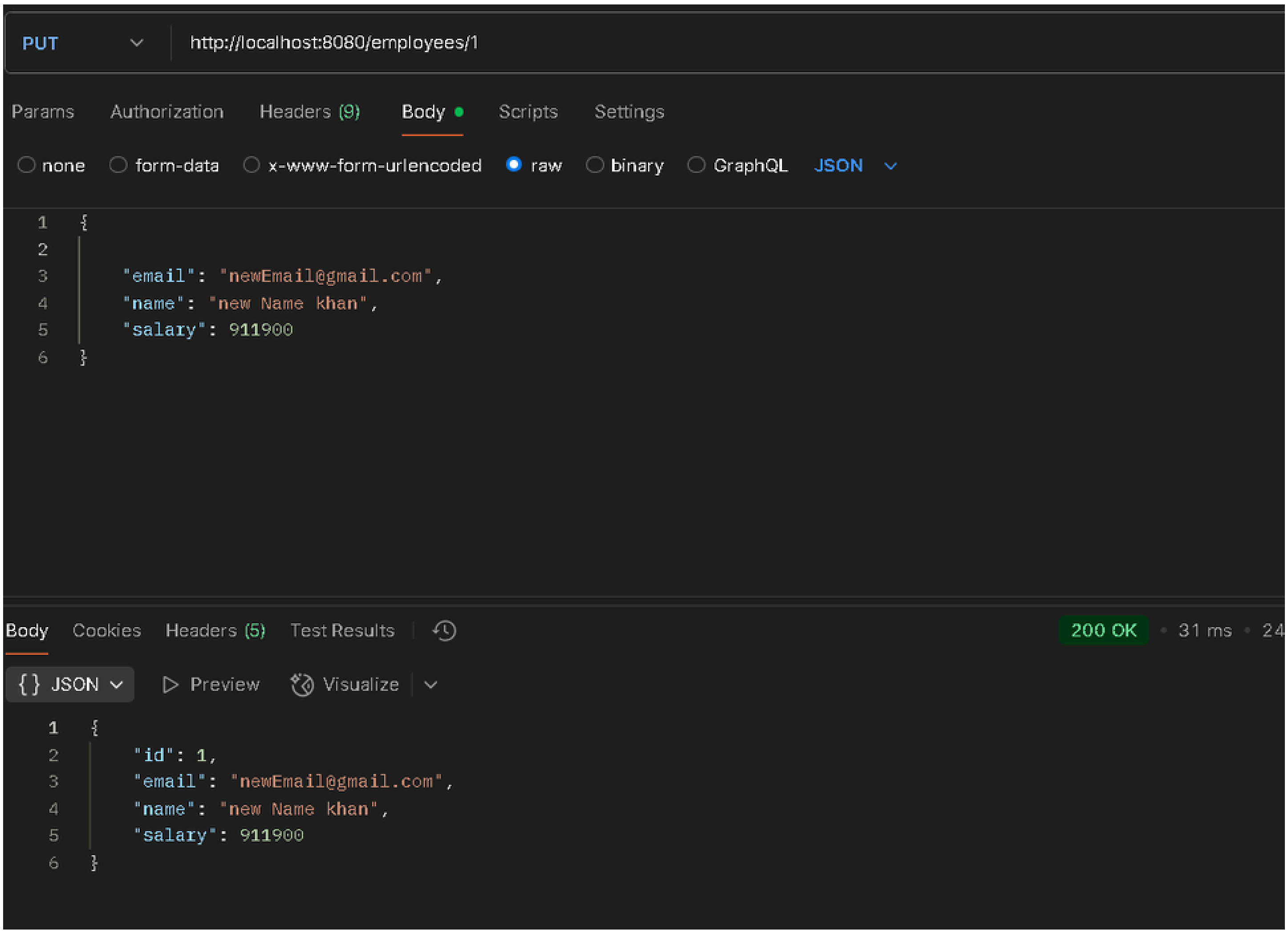

✕ Update Employee By Id

```java
@Override
public EmployeeDto updateById(Long id, EmployeeDto dto) {
    EmployeeEntity employee = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with ID: " + id));

    modelMapper.map(dto, employee); // ✅ Safe now — won't overwrite ID

    EmployeeEntity updated = employeeRepository.save(employee);
    return modelMapper.map(updated, EmployeeDto.class); // response includes ID
}
```

✅ Why this Works

- You **still use ModelMapper**, so less manual code ✅
- You **prevent it from modifying the ID**, which caused the error ✅

Now i re-run the application and test the put request api


PUT http://localhost:8080/employees/1

```json
{

    "email": "newEmail@gmail.com",
    "name": "new Name khan",
    "salary": 911900
}
```

200 OK • 31 ms • 24

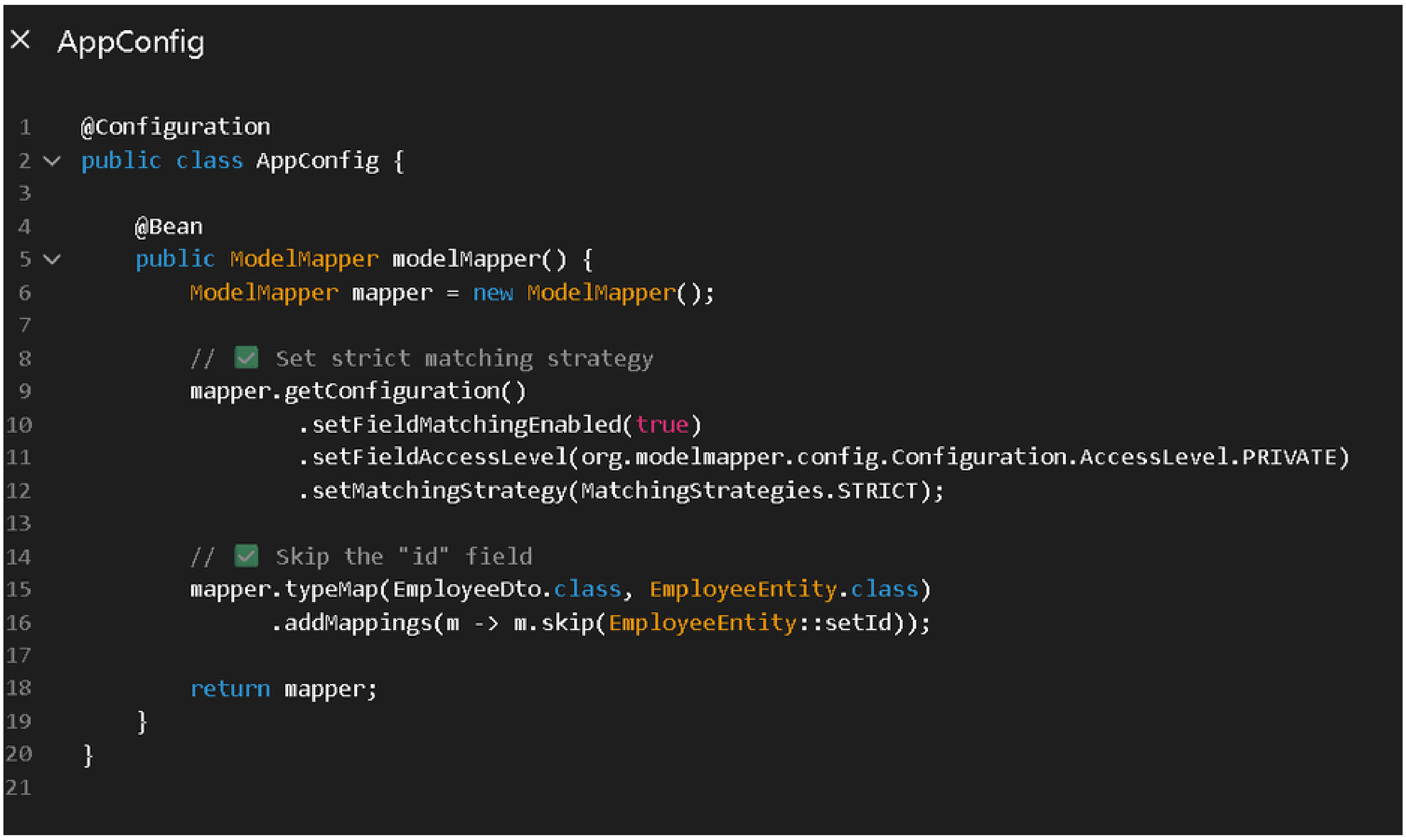
```json
{
    "id": 1,
    "email": "newEmail@gmail.com",
    "name": "new Name khan",
    "salary": 911900
}
```

Now i successfully passed the put request api. without giving the id field in the json request body. but give it to in the url path

What i am doing here let’s explore this

You're configuring a ModelMapper bean to **map between your EmployeeDto and EmployeeEntity classes**, and **explicitly skipping the id field** during that mapping.

Here's a breakdown:



```java
@Configuration
public class AppConfig {

    @Bean
    public ModelMapper modelMapper() {
        ModelMapper mapper = new ModelMapper();

        // ✅ Set strict matching strategy
        mapper.getConfiguration()
                .setFieldMatchingEnabled(true)
                .setFieldAccessLevel(org.modelmapper.config.Configuration.AccessLevel.PRIVATE)
                .setMatchingStrategy(MatchingStrategies.STRICT);

        // ✅ Skip the "id" field
        mapper.typeMap(EmployeeDto.class, EmployeeEntity.class)
              .addMappings(m -> m.skip(EmployeeEntity::setId));

        return mapper;
    }
}
```

✅ What You're Doing Here:

Section	Purpose
@Configuration	Declares a config class for Spring
@Bean public ModelMapper	Registers a single ModelMapper bean into Spring context
setFieldMatchingEnabled(true)	Enables mapping even on private fields
setFieldAccessLevel(PRIVATE)	Allows ModelMapper to access private fields
setMatchingStrategy(STRICT)	Ensures fields match exactly by name and type
m.skip(EmployeeEntity::setId)	🚫 Tells the mapper to **not set id** when mapping DTO → Entity

❓ Why skip id?

You're doing this so that during an **update operation**, the existing entity's ID is **not overwritten with null**, which would trigger errors like:

```
Identifier of an instance ... was altered from 3 to null
```

So you're saying:

"Map everything from DTO to Entity, but leave the ID alone — don't touch it."

✅ Example:

Input DTO (no ID):

```
{
  "name": "Zayn Malik",
  "email": "zayn@gmail.com",
  "salary": 59000
}
```

Mapped Entity:

```
EmployeeEntity {
    id = 3            // stays unchanged
    name = "Zayn Malik"
    email = "zayn@gmail.com"
    salary = 59000
}
```

So why i am telling you about this? is this a company Standars ?

Great question. Let's answer that clearly and practically:

✅ **Is this approach company standard?**

Yes — skipping the id field during DTO → Entity mapping is a common and accepted practice in real-world projects.
In fact, it's considered **a good standard** in many companies.

💼 **Why companies prefer this approach:**

Reason	Explanation
✅ **Avoid accidental ID override**	Prevents bugs where id becomes null or wrong, especially during updates.
✅ **Cleaner DTOs**	Clients don't need to send id in JSON bodies — it's already in the URL.
✅ **Respects REST principles**	In REST APIs, the resource id is ideally passed via the **URL path**, not the body.
✅ **Keeps update logic safe**	You fetch the existing entity from DB using id, then update fields from DTO (excluding id).
✅ **Works well with tools like ModelMapper or MapStruct**	These tools support this approach out-of-the-box with configurations like .skip(...).

✅ Example from companies:

In a typical Spring Boot + REST project:

```java
@PutMapping("/employees/{id}")
public ResponseEntity<EmployeeDto> update(@PathVariable Long id, @RequestBody EmployeeDto dto)
    EmployeeEntity entity = repository.findById(id).orElseThrow(...);

    modelMapper.map(dto, entity); // safely updates fields (id skipped)

    repository.save(entity);

    return ResponseEntity.ok(modelMapper.map(entity, EmployeeDto.class));
}
```

🔒 Without skipping ID — a **real bug** could happen:

Imagine this DTO comes in:

```
{
  "id": null,
  "name": "Zayn"
}
```

If you allow ModelMapper to set id, it will overwrite your database record's id with null or some wrong value — 💥 breaking persistence logic.

✅ Summary

Skipping id while mapping from DTO → Entity is not just acceptable — it's **recommended** in most enterprise-level Spring Boot applications.