

# Java

Friday, November 21, 2025 11:54 AM

Jframe class

Jtext

Javax -> functionalities beyond JDK -> namespace -> javax.swing ,  
javax.servlet

We are saying spring pls manage and create an instance of this class - **@Component**

**@Component** -> Hey Spring, pls make a instance of this class and manage it

How will Spring find these components -> **@ComponentScan** -> Now Spring found the components we annotated -> Now, Spring will register them as beans in the ApplicationContext.

Dependencies are **injected automatically**

**@Component**

```
public class Engine {  
    public String start() {  
        return "Engine started!";  
    }
```

Here, Engine is registered as a Bean

Now we will do Constructor Injection

Constructor injection: Spring sees the constructor needs an Engine and injects the Engine bean it already created.

A service with constructor injection: CarService

**@Service**

```
public class CarService {  
    private final Engine engine;  
    public CarService(Engine engine) {  
        this.engine = engine;  
    }  
  
    public String drive() {  
        return engine.start() + " Car is driving...";  
    }  
}
```

Object

```
class OrderService {  
    private PaymentService paymentService = new PaymentService(); // tightly coupled  
}
```

Apring web dependency for now on spring initialisr

## Beans

A Spring Bean is simply a Java object that is:

- Instantiated by Spring
- Configured by Spring
- Managed throughout its lifecycle by Spring

Beans form the backbone of a Spring application and are the core building blocks that are wired together to create the application.

### What is IoC?

- **Inversion of Control** means that instead of your code controlling object creation and dependency management, the framework (Spring) takes over that responsibility.
- This is achieved through **Dependency Injection (DI)** — objects declare their dependencies, and Spring injects them at runtime.

### BeanFactory vs ApplicationContext

- **BeanFactory**:
  - The simplest container in Spring.
  - Provides basic DI functionality.
  - Lazily initializes beans (creates them only when requested).
- **ApplicationContext**:
  - A superset of BeanFactory.
  - Adds enterprise-level features like:
    - Internationalization (i18n)
    - Event propagation
    - Bean lifecycle management
    - Automatic BeanPostProcessor registration
  - Eagerly initializes singleton beans by default.

### Why ApplicationContext is central?

- It acts as the **IoC container** that:
  - Loads bean definitions from configuration (XML, annotations, or Java config).
  - Manages the lifecycle of beans.

- Resolves dependencies automatically.

### Common Implementations of ApplicationContext

- ClassPathXmlApplicationContext – loads context from XML in classpath.
- AnnotationConfigApplicationContext – loads context from Java-based configuration.
- WebApplicationContext – specialized for web applications.

IoC means **your code does not create and manage dependencies directly**.

Instead, the **framework (Spring)** controls object creation and wiring. This is usually implemented via **Dependency Injection (DI)**.

Instead of doing:

Java

```
Engine engine = new Engine();
Car car = new Car(engine);
Show more lines
```

We let **Spring inject the dependency**:

XML

```
<!-- beans.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
<bean id="engine" class="Engine"/>
<bean id="car" class="Car">
<constructor-arg ref="engine"/>
</beans>
```

Show more lines



Absolutely! In Spring, you can create and manage beans in two primary ways:

1. **Using @Component and stereotype annotations** (automatically discovered via component scanning)
2. **Using @Bean methods inside a @Configuration class** (explicitly declared, great for fine-grained control or third-party classes)

Below are lucid explanations with runnable-style code samples for both.

# 1) Creating Beans with @Component (and Stereotype Annotations)

Spring will **auto-detect** classes annotated with @Component (or its specializations) during **component scanning** and register them as beans in the ApplicationContext.

## Common Annotations

- @Component → generic Spring-managed bean
- @Service → business/service layer semantics
- @Repository → persistence/DAO semantics (adds exception translation)
- @Controller → MVC web controller (returns views)
- @RestController → REST controller (@Controller + @ResponseBody)  
All of these are beans; the stereotypes add intent and extra behaviors (e.g., @Repository enables @Repository exception translation if @EnableTransactionManagement / persistence exception translation is configured).

## Minimal Example (Component Scanning)

### Package structure

```
com.example.app
  └── config
      └── AppConfig.java
  └── domain
      ├── Engine.java
      ├── CarService.java
      ├── CarRepository.java
      ├── CarController.java
      └── CarRestController.java
```

### Config enabling component scan

```
// com/example/app/config/AppConfig.java
package com.example.app.config;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan(basePackages = "com.example.app") // scan this package tree
public class AppConfig {}
```

### Generic component

```
// com/example/app/domain/Engine.java
package com.example.app.domain;
import org.springframework.stereotype.Component;
@Component
public class Engine {
    public String start() {
        return "Engine started!";
    }
}
```

### Service layer

```
// com/example/app/domain/CarService.java
package com.example.app.domain;
import org.springframework.stereotype.Service;
@Service
public class CarService {
    private final Engine engine;
    // Prefer constructor injection (testable, immutable)
    public CarService(Engine engine) {
        this.engine = engine;
    }
    public String drive() {
        return engine.start() + " Car is driving...";
    }
}
```

## **Repository layer**

```
// com/example/app/domain/CarRepository.java
package com.example.app.domain;
import org.springframework.stereotype.Repository;
@Repository
public class CarRepository {
    public String find.byId(Long id) {
        // pretend to hit DB
        return "Car#" + id;
    }
}
```

## **Spring MVC Controller (returns view names)**

```
// com/example/app/domain/CarController.java
package com.example.app.domain;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
@Controller
public class CarController {
    private final CarService carService;
    public CarController(CarService carService) {
        this.carService = carService;
    }
    @GetMapping("/car/view")
    public String viewCar(Model model) {
        model.addAttribute("status", carService.drive());
        return "carView"; // resolves to a template (e.g., Thymeleaf)
    }
}
```

## **REST Controller (returns JSON/text directly)**

```
// com/example/app/domain/CarRestController.java
package com.example.app.domain;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class CarRestController {
    private final CarService carService;
    private final CarRepository carRepository;
    public CarRestController(CarService carService, CarRepository carRepository) {
        this.carService = carService;
        this.carRepository = carRepository;
    }
    @GetMapping("/api/car/drive")
    public String drive() {
        return carService.drive();
    }
}
```

## **Bootstrapping (non-Boot app for demo)**

```
// Main.java
import com.example.app.config.AppConfig;
import com.example.app.domain.CarService;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Main {
    public static void main(String[] args) {
        try (var ctx = new AnnotationConfigApplicationContext(AppConfig.class)) {
            CarService carService = ctx.getBean(CarService.class);
            System.out.println(carService.drive());
        }
    }
}
```

## Notes & Best Practices

- **Constructor injection** is recommended; Spring 4.3+ auto-wires single constructors even without @Autowired.
- Use @Qualifier if there are multiple beans of the same type:  
public CarService(@Qualifier("sportEngine") Engine engine) { ... }
- Customize bean names:  
@Component("engineV1")  
public class Engine { ... }
- Control scope:  
@Service  
@Scope("prototype") // or "singleton" (default), "request", "session" in web contexts  
public class CarService { ... }

## 2) Creating Beans with @Bean in a @Configuration Class

Use this when:

- You need **precise control** over construction, initialization, or destruction
- You are wiring **third-party classes** (not annotated with Spring stereotypes)
- You want to **conditionally** define beans or choose profiles
- You need to **override** component-scanned beans

### Example: Pure Java Config with @Bean

```
// com/example/app/config/AppConfig.java
package com.example.app.config;
import com.example.app.domain.Engine;
import com.example.app.domain.CarService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    // Simple bean
    @Bean
    public Engine engine() {
        return new Engine();
    }
    // Bean method DI via parameters (Spring resolves by type)
    @Bean
    public CarService carService(Engine engine) {
        return new CarService(engine);
    }
}
Main

import com.example.app.config.AppConfig;
import com.example.app.domain.CarService;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Main {
    public static void main(String[] args) {
        try (var ctx = new AnnotationConfigApplicationContext(AppConfig.class)) {
            CarService carService = ctx.getBean(CarService.class);
            System.out.println(carService.drive());
        }
    }
}
```

### Using @Bean for Third-Party Classes

```
// Suppose this is a third-party DataSource class not annotated with Spring
import javax.sql.DataSource;
import org.apache.commons.dbcp2.BasicDataSource;
@Configuration
```

```

public class DataConfig {
    @Bean
    public DataSource dataSource() {
        var ds = new BasicDataSource();
        ds.setUrl("jdbc:h2:mem:testdb");
        ds.setUsername("sa");
        ds.setPassword("");
        ds.setInitialSize(2);
        return ds;
    }
}

```

## Initialization / Destruction Hooks

```

@Configuration
public class LifecycleConfig {
    @Bean(initMethod = "init", destroyMethod = "shutdown")
    public CacheManager cacheManager() {
        return new CacheManager();
    }
}

// Hypothetical class
class CacheManager {
    public void init() { System.out.println("Cache initialized"); }
    public void shutdown() { System.out.println("Cache shutdown"); }
}

```

## Choosing Primary / Qualifiers / Profiles

```

@Configuration
public class EnginesConfig {
    @Bean
    @Primary
    public Engine defaultEngine() { return new Engine(); }

    @Bean(name = "sportEngine")
    public Engine sportEngine() { return new Engine(); }
}

// Usage:
public class CarService {
    private final Engine engine;
    public CarService(@Qualifier("sportEngine") Engine engine) {
        this.engine = engine;
    }
}

```

```

@Configuration
@Profile("prod")
public class ProdConfig {
    @Bean
    public DataSource prodDataSource() { /* prod settings */ }
}

@Configuration
@Profile("dev")
public class DevConfig {
    @Bean
    public DataSource devDataSource() { /* dev settings */ }
}

```

Activate a profile via properties or programmatically:

```

var ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment(). setActiveProfiles("dev");
ctx.register(DevConfig.class, ProdConfig.class);

```

## When to Use Which?

- **@Component / Stereotypes:** Best for your **own application classes** (services, repositories, controllers). Cleaner, less boilerplate, and aligns with layered architecture semantics.

- **@Bean in @Configuration:** Best for **third-party objects, explicit construction** logic, **conditional beans, profiles**, or when you need **fine-grained control** over lifecycle and wiring. In real projects, you'll often combine both: component scan for your layers + @Bean methods for infra and third-party integrations.

If you'd like, I can share a small **Spring Boot** version of the same examples (with `@SpringBootApplication`) to show how Boot auto-config + component scanning works out of the box.

You're right—let me walk you through the code step by step and **explain what each piece is doing** and **why** you would use it. I'll keep it crisp and lucid.

## 1) Beans via @Component & Stereotypes

### What is happening conceptually?

- When you annotate classes with `@Component` (or `@Service`, `@Repository`, `@Controller`, `@RestController`), you're telling Spring: "**Please create and manage an instance of this class.**"
- Spring finds these classes using **component scanning** (enabled via `@ComponentScan`) and registers them as beans in the `ApplicationContext`.
- Dependencies are **injected automatically** (via constructor injection in our examples).

### Code and Explanation

#### Configuration

```
@Configuration
@ComponentScan(basePackages = "com.example.app")
    • @Configuration: Marks this class as a source of bean definitions.
    • @ComponentScan(...): Tells Spring where to look for annotated components. Any
      @Component, @Service, etc., under com.example.app will be discovered and registered.
```

#### A generic component: Engine

```
@Component
public class Engine {
    public String start() {
        return "Engine started!";
    }
    • @Component: Registers Engine as a bean.
    • start(): A simple method to prove the bean works.
```

#### A service with constructor injection: CarService

```
@Service
public class CarService {
```

```

private final Engine engine;
public CarService(Engine engine) {
    this.engine = engine;
}
public String drive() {
    return engine.start() + " Car is driving...";
}


- @Service: Same as @Component but semantically indicates business logic layer.
- Constructor injection: Spring sees the constructor needs an Engine and injects the Engine bean it already created.
- drive(): Uses the injected dependency.



Why constructor injection? It makes the class immutable, easier to test, and ensures dependencies are available at creation time. Spring (4.3+) will autowire the only constructor without needing @Autowired.


```

## Repository

```

@Repository
public class CarRepository {
    public String findById(Long id) {
        return "Car#" + id;
    }
}


- @Repository: Marks persistence/DAO components. It can enable exception translation (converting low-level persistence exceptions into Spring's DataAccessException).

```

## MVC Controller

```

@Controller
public class CarController {
    private final CarService carService;
    public CarController(CarService carService) {
        this.carService = carService;
    }
    @GetMapping("/car/view")
    public String viewCar(Model model) {
        model.addAttribute("status", carService.drive());
        return "carView";
    }
}


- @Controller: A web controller that returns view names (e.g., Thymeleaf templates).
- @GetMapping("/car/view"): Maps HTTP GET /car/view to this method.
- Model: Adds attributes to be used in the rendered view.
- return "carView";: The name of the template to render.

```

## REST Controller

```

@RestController
public class CarRestController {
    private final CarService carService;
    private final CarRepository carRepository;
    public CarRestController(CarService carService, CarRepository carRepository) {
        this.carService = carService;
        this.carRepository = carRepository;
    }
    @GetMapping("/api/car/drive")
    public String drive() {
        return carService.drive();
    }
}


- @RestController: Equivalent to @Controller + @ResponseBody. Returns the response directly (JSON/text) rather than rendering a view.
- Perfect for APIs returning data for clients.

```

## Bootstrapping (non-Boot)

```
try (var ctx = new AnnotationConfigApplicationContext(AppConfig.class)) {
```

```

CarService carService = ctx.getBean(CarService.class);
System.out.println(carService.drive());
}


- Creates an ApplicationContext using your configuration.
- Retrieves the CarService bean from the container.
- Calls drive() to see the wiring in action.
- Try-with-resources ensures the context is closed properly (calls destroy methods).

```

## 2) Beans via @Bean in a @Configuration Class

### What is happening conceptually?

- You explicitly define bean creation **in Java methods** inside a @Configuration class.
- Useful for **third-party classes** or when you need **fine-grained control** (e.g., init/destroy methods, conditional beans, profiles).

### Code and Explanation

#### Explicit bean definitions

```

@Configuration
public class AppConfig {
    @Bean
    public Engine engine() {
        return new Engine();
    }
    @Bean
    public CarService carService(Engine engine) {
        return new CarService(engine);
    }
}


- @Bean: Registers the returned object as a bean.
- carService(Engine engine): Spring resolves the parameter by type from the container and injects the engine() bean.
- This approach gives complete control over construction logic (you can pass custom parameters, conditionals, etc.).

```

#### Third-party class example (DataSource)

```

@Configuration
public class DataConfig {
    @Bean
    public DataSource dataSource() {
        var ds = new BasicDataSource();
        ds.setUrl("jdbc:h2:mem:testdb");
        ds.setUsername("sa");
        ds.setPassword("");
        ds.setInitialSize(2);
        return ds;
    }
}


- This is perfect when the class (e.g., BasicDataSource) is not annotated with Spring stereotypes and you still want it managed by the container.
- You configure properties in code before returning the bean.

```

#### Lifecycle hooks

```

@Configuration
public class LifecycleConfig {
    @Bean(initMethod = "init", destroyMethod = "shutdown")
    public CacheManager cacheManager() {
        return new CacheManager();
    }
}
class CacheManager {
    public void init() { System.out.println("Cache initialized"); }
}

```

```
public void shutdown() { System.out.println("Cache shutdown"); }
• initMethod runs after the bean is constructed and dependencies injected.
• destroyMethod runs when the context is closing (e.g., graceful shutdown).
• Great for resources like caches, thread pools, or connections.
```

### Choosing which bean to inject (@Primary, @Qualifier)

```
@Configuration
public class EnginesConfig {
    @Bean
    @Primary
    public Engine defaultEngine() { return new Engine(); }
    @Bean(name = "sportEngine")
    public Engine sportEngine() { return new Engine(); }
}

• @Primary: If there are multiple beans of the same type, this one wins by default.
• @Bean(name = "sportEngine"): Gives a specific name for use with @Qualifier("sportEngine") in injection points.
```

### Environment-specific beans with @Profile

```
@Configuration
@Profile("prod")
public class ProdConfig {
    @Bean
    public DataSource prodDataSource() { /* prod settings */ }
}

@Configuration
@Profile("dev")
public class DevConfig {
    @Bean
    public DataSource devDataSource() { /* dev settings */ }
}

• @Profile: Only registers these beans when the profile is active (e.g., "dev" vs "prod").
• Activate programmatically:
    var ctx = new AnnotationConfigApplicationContext();
    ctx.getEnvironment().setActiveProfiles("dev");
    ctx.register(DevConfig.class, ProdConfig.class);
```

## When to Use Which (Summary)

- **@Component / Stereotypes:** Best for your **own application classes** (services, repos, controllers). Clean and automatic via scanning.
  - **@Bean in @Configuration:** Best for **third-party objects**, complex construction, lifecycle control, conditional registration, or environment-specific configs.
- In real-world apps, you **combine both**: use component scanning for your layers and @Bean methods for infrastructure.

## Quick Visual Mental Model

- **Component Scanning:** "Find all annotated classes in these packages and create beans automatically."
- **@Bean methods:** "Create exactly these beans, with this construction logic."

If you want, I can convert these examples into a **Spring Boot** project structure and show how `@SpringBootApplication` automatically enables component scanning and simplifies bootstrapping. Or we can add **@Autowired, scopes, and qualifiers** examples inside controllers/services to see different DI flavors. What would you like next?

# IOC Container aka Application Context

IOC Container - The Bean Factory

At the center of everything is the ApplicationContext – Spring's IoC (Inversion of Control) container.

## How to create Beans?

### 1. Using @Component (and Stereotype annotations)

- @Component // generic bean
- @Service // business logic
- @Repository // database access
- @Controller // web layer
- @RestController

### 2. Using @Bean in a @Configuration Class

## Lifecycle of a Bean



The image is explaining the **lifecycle of a Bean** in the context of **Spring Framework** (or similar dependency injection frameworks in Java).

Here's what each stage means:

- 1. Bean Created**
  - The container (like Spring IoC container) creates an instance of the bean class.
- 2. Dependency Injected**
  - The container injects all required dependencies into the bean (using constructor injection, setter injection, etc.).
- 3. Bean Initialized**
  - Any initialization logic runs here (e.g., methods annotated with @PostConstruct or custom init methods).
- 4. Bean is Used**

- The bean is now ready for use by the application. It serves its purpose during the application lifecycle.

## 5. Bean is Destroyed

- When the application context is closed, the bean is cleaned up (e.g., methods annotated with @PreDestroy or custom destroy methods).

This lifecycle is fundamental in **Spring's Inversion of Control (IoC)** and **Dependency Injection (DI)** concepts.

Would you like me to **draw a detailed diagram showing how this lifecycle works in Spring with annotations and interfaces?** Or should I **explain the role of each phase with code examples?**

@Configuration -> within this file we gonna make a lotta beans

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'Project'. It includes a 'module1Introduction' folder containing '.idea', '.mvn', 'src' (with 'main' and 'java' subfolders), 'target', and various configuration files like '.gitattributes', '.gitignore', 'HELP.md', 'mvnw', 'mvnw.cmd', and 'pom.xml'. The right panel shows two open files: 'Module1IntroductionApplication.java' and 'PaymentService.java'. 'Module1IntroductionApplication.java' contains the main method and imports. 'PaymentService.java' contains a single method 'pay()' that prints 'Paying...'. The code editor has syntax highlighting and some yellow markers.

```

package com.codingshuttle.anuj.module1Introduction;

public class PaymentService { no usages

    public void pay() { no usages
        System.out.println("Paying..."); }
}

```

The screenshot shows the IntelliJ IDEA interface after modifications. The 'src/main/java/com.codingshuttle.anuj.module1Introduction' package now contains both 'Module1IntroductionApplication.java' and 'PaymentService.java'. The 'PaymentService' class is still present with its 'pay()' method. The 'Module1IntroductionApplication' class now includes annotations: '@SpringBootApplication' and 'public static void main(String[] args) { SpringApplication.run(Module1IntroductionApplication.class, args); }'. The code editor shows the updated code with syntax highlighting.

```

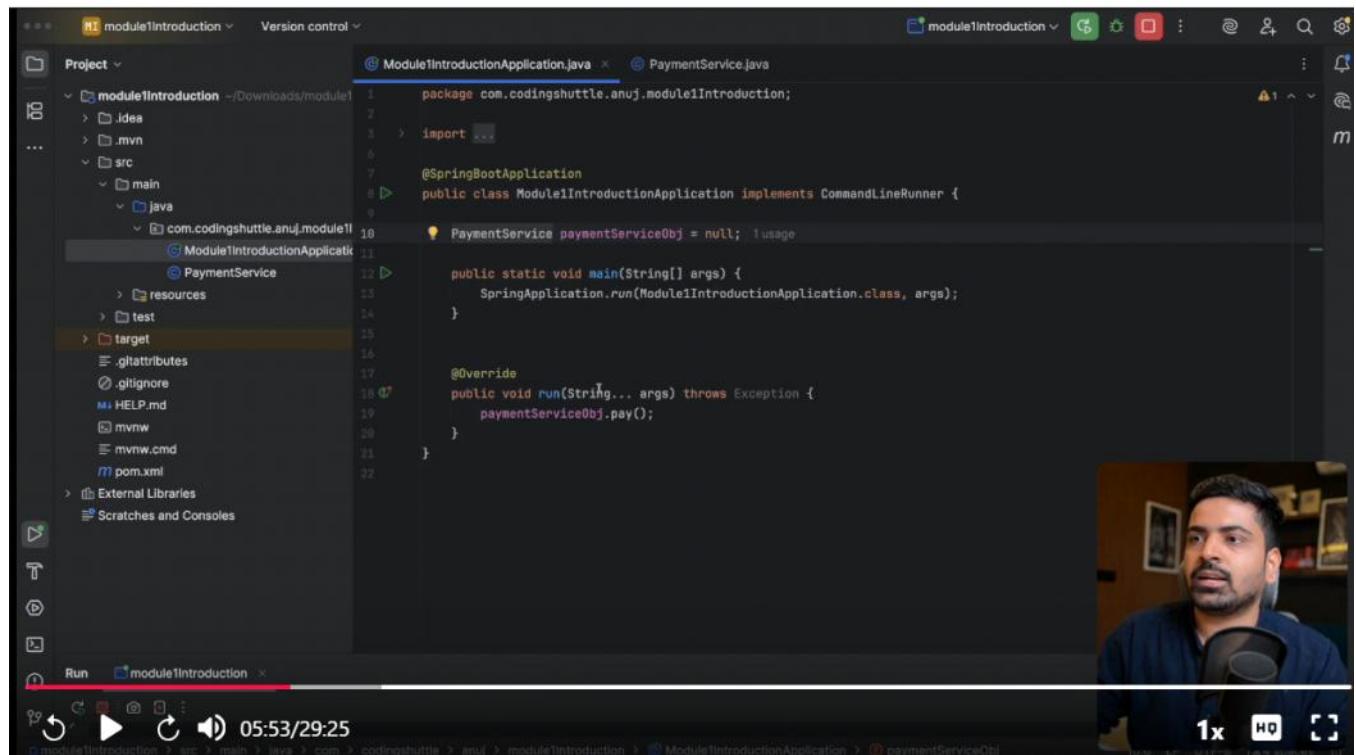
package com.codingshuttle.anuj.module1Introduction;
import ...;
@SpringBootApplication
public class Module1IntroductionApplication {

    public static void main(String[] args) {
        SpringApplication.run(Module1IntroductionApplication.class, args);
    }
}

```

Output: Paying...

Now we are just trying to get into some Spring magic



The screenshot shows a Java project named "module1Introduction" in an IDE. The project structure is visible on the left, and the code editor on the right displays `Module1IntroductionApplication.java`. The code is as follows:

```
package com.codingshuttle.anuj.module1Introduction;
import ...;
@SpringBootApplication
public class Module1IntroductionApplication implements CommandLineRunner {
    PaymentService paymentServiceObj = null; //usage
    public static void main(String[] args) {
        SpringApplication.run(Module1IntroductionApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        paymentServiceObj.pay();
    }
}
```

The video player interface at the bottom indicates the video is at 05:53/29:25 and has a resolution of 1x HQ.

After using `@Autowired`,

The screenshot shows the IntelliJ IDEA interface. The code editor displays `ModuleIntroductionApplication.java` with the following content:

```
package com.codingshuttle.anuj.module1Introduction;
import ...;
@SpringBootApplication
public class Module1IntroductionApplication implements CommandLineRunner {
    @Autowired
    PaymentService paymentServiceObj;
    public static void main(String[] args) {
        SpringApplication.run(Module1IntroductionApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        paymentServiceObj.pay();
    }
}
```

The run console shows application logs:

```
2025-11-14T17:22:36.984+05:30 INFO 61748 --- [module1Introduction] [main] c.c.a.m.Module1IntroductionApplication : Starting
2025-11-14T17:22:36.985+05:30 INFO 61748 --- [module1Introduction] [main] c.c.a.m.Module1IntroductionApplication : No active profile is defined. Falling back to auto-configuration.
2025-11-14T17:22:37.238+05:30 INFO 61748 --- [module1Introduction] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2025-11-14T17:22:37.245+05:30 INFO 61748 --- [module1Introduction] [main] o.apache.catalina.core.StandardService : Starting
2025-11-14T17:22:37.260+05:30 INFO 61748 --- [module1Introduction] [main] o.apache.catalina.core.StandardEngine : Starting
2025-11-14T17:22:37.269+05:30 INFO 61748 --- [module1Introduction] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initial
2025-11-14T17:22:37.363+05:30 INFO 61748 --- [module1Introduction] [main] w.s.c.ServletWebServerApplicationContext : Root Web
2025-11-14T17:22:37.364+05:30 INFO 61748 --- [module1Introduction] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat s
```

The video feed in the bottom right corner shows a man speaking.

Spring IOC container will manage the objects

Now, we can create `@Beans` instead of `@Component` as in `PaymentService` class earlier.

The screenshot shows the IntelliJ IDEA interface. The code editor displays `AppConfig.java` with the following content:

```
package com.codingshuttle.anuj.module1Introduction;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean
    PaymentService paymentService() {
        return new PaymentService();
    }
}
```

The video feed in the bottom right corner shows a man speaking.

The screenshot shows a Java development environment with a code editor, a terminal, and a video player.

**Code Editor:** The active file is `PaymentService.java`. The code contains annotations like `@Service`, `@Controller`, `@Repository`, and `@RestController`. It includes methods `pay()` and `beforeDestroy()`.

```

11  // @Service
12  // @Controller
13  // @Repository
14  // @RestController
15
16  public class PaymentService {
17
18      public void pay() { System.out.println("Paying..."); }
19
20      @PostConstruct no usages
21      public void afterInit() {
22          System.out.println("Before paying.");
23      }
24
25      public void beforeDestroy() { no usages
26          System.out.println("After payment is done");
27      }
28
29  }

```

**Terminal:** Shows log output from the application's main method and shutdown hook.

```

2025-11-14T17:38:16.151+05:30 INFO 62055 --- [module1Introduction] [main] w.s.c.ServletWebServerApplicationContext : Root Web
2025-11-14T17:38:16.255+05:30 INFO 62055 --- [module1Introduction] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat s
2025-11-14T17:38:16.258+05:30 INFO 62055 --- [module1Introduction] [main] c.c.a.m.Module1IntroductionApplication : Started
2025-11-14T17:45:51.615+05:30 INFO 62055 --- [ionShutdownHook] o.s.b.w.e.tomcat.GracefulShutdown : Commenc
2025-11-14T17:45:51.622+05:30 INFO 62055 --- [tomcat-shutdown] o.s.b.w.e.tomcat.GracefulShutdow

```

**Video Player:** A video of a man speaking is overlaid on the right side of the screen.

Got it, Arka. Let's use **your exact code** and explain **what @Bean does, how it differs from @Component, and why you see new in the @Bean method but not with @Component**—step by step and super lucid.

## Your code (annotated)

```

@Component
public class PaymentService {
    public void pay() {
        System.out.println("Paying...");
    }
}
// Bootstraps the Spring context
public static void main(String[] args) {
    SpringApplication.run(Module1IntroductionApplication.class, args);
}
@Configuration
public class AppConfig {
    @Bean
    @Scope("request")
    public PaymentService paymentService() {
        // more logic
        return new PaymentService(); // <-- you wrote "new" here
    }
}

```

You are defining **two ways** to register `PaymentService` as a Spring bean:

1. `@Component` on the class
2. `@Bean` method inside `@Configuration`

Both produce a bean, but **they do it differently**. Here's how.

## How @Component works (with your class)

- @Component means: "**Spring, during component scanning, detect this class and create a bean from it.**"
- Spring finds PaymentService when the app starts (thanks to @SpringBootApplication enabling component scan).
- Spring creates an instance **without you writing new**. It does this internally using reflection:
  - It calls the class's constructor (default no-args constructor here).
  - It registers that instance in the Application Context.
- **Default scope** is singleton (one instance for the whole app), unless you add @Scope on the class.

**Why no new in your code?** Because **you're not the factory** here—**Spring is**. You just annotate the class; Spring takes care of constructing the object.

## How @Bean works (with your AppConfig)

- @Bean means: "**Spring, call this method and register its return value as a bean.**"
- Spring loads AppConfig (because it's @Configuration).
- Spring **invokes the paymentService() method and whatever object you return becomes the bean**.
- In your method, you wrote:

```
return new PaymentService();
```

That's **you** explicitly constructing the object (i.e., **you are the factory** here).
- You also set @Scope("request"), which tells Spring: "create a fresh instance **per HTTP request** (via a proxy)." This only works in a web context.

**Why do you use new here?** Because **@Bean is a factory method**—you are telling Spring exactly **how to build** the object, so you write new (or call builders, set properties, etc.).

## Short, crisp contrast (using your example)

Aspect	@Component on PaymentService	@Bean in AppConfig.paymentService()
Who constructs the object?	Spring (via component scan & reflection)	You, inside the method
Where defined?	On the class itself	Inside a configuration class
Bean name (default)	paymentService (from class name)	paymentService (from method name)
Scope	Singleton by default (unless you put @Scope on class)	You set it per method (here: request)
Use cases	Simple, your own classes	Custom construction, variants, third-party classes, special scopes

## Important: Your code has two beans of the same type/name

Since both @Component and @Bean produce a bean named paymentService by default:

- Spring Boot (modern versions) will **fail** with a bean definition collision (or injection ambiguity) unless you:
  - Give **different names**:

```
@Component("paymentServiceComponent")
public class PaymentService { ... }
@Bean(name = "paymentServiceRequestScoped")
@Scope("request")
public PaymentService paymentService() { return new PaymentService(); }
```
  - Or **remove one** of the definitions (recommended).
  - Or use @Primary on the one you want injected by default.

## One more subtlety: @Scope("request") needs a web context

- If you run only main() without a web server/controller, a **request-scoped** bean won't be meaningful.
- In a Spring Boot web app, @Scope("request") works by injecting a **proxy** that gives a fresh PaymentService instance on each HTTP request.

If you just want to print "Paying..." on startup, don't use request scope; use singleton and call it via a runner:

```
@SpringBootApplication
public class Module1IntroductionApplication {
    public static void main(String[] args) {
        SpringApplication.run(Module1IntroductionApplication.class, args);
    }
    @Bean
    public CommandLineRunner runner(PaymentService paymentService) {
        return args -> paymentService.pay(); // prints "Paying..."
    }
}
```

And choose **one** registration:

#### **Option A: via @Component**

```
@Component
public class PaymentService {
    public void pay() { System.out.println("Paying..."); }
}
```

#### **Option B: via @Bean**

```
@Configuration
public class AppConfig {
    @Bean
    public PaymentService paymentService() {
        return new PaymentService(); // we construct it here
    }
}
```

## The core answer to your "why"

- In **@Component**, you don't write new because you are **declaring** that the class is a Spring-managed component, and Spring's **component scanner** creates it for you.
- In **@Bean**, you do write new because you are **defining a factory method** that returns the object; **you control** how it's constructed. Spring simply **registers the return value** as the bean.

Think of it like:

- @Component → **"Spring, find this class and make the bean yourself."**
- @Bean → **"Spring, here's a method; I'll build the object and hand it to you. Use it as a bean."**

If you tell me whether your app is web or CLI, I'll refactor your exact snippet to the cleanest working version (no collisions, correct scope) and show where "Paying..." will print.

Perfect, Arka! Here are **two minimal, runnable examples** that achieve the **same thing**—register PaymentService as a Spring bean and call pay()—using:

1. **@Component** (component scanning)
2. **@Bean** (explicit configuration)

Both examples print Paying... on application startup via a CommandLineRunner.

## ◊ Example A — Using @Component

**What you do:** Annotate the class; let Spring discover and create it.

```
// DemoApplication.java
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    // Runs after the context starts; injects PaymentService and calls pay()
    @Bean
    public CommandLineRunner runner(PaymentService paymentService) {
        return args -> paymentService.pay();
    }
}

// PaymentService.java
import org.springframework.stereotype.Component;
@Component // <-- Spring will discover and instantiate this bean
public class PaymentService {
    public void pay() {
        System.out.println("Paying...");
    }
}
```

#### How it works:

- `@SpringBootApplication` enables component scanning.
- Spring finds `@Component` on `PaymentService` and **creates the bean** (you don't write new).
- `CommandLineRunner` is injected with `PaymentService` and calls `pay()`.

## ◊ Example B — Using `@Bean`

**What you do:** Provide a factory method returning the object.

```
// DemoApplication.java
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    // Runs after the context starts; injects PaymentService and calls pay()
    @Bean
    public CommandLineRunner runner(PaymentService paymentService) {
        return args -> paymentService.pay();
    }
}

// AppConfig.java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean // <-- You construct and return the bean here
    public PaymentService paymentService() {
        return new PaymentService(); // <-- "new" because you are the factory
    }
}

// PaymentService.java
```

```
// No annotations needed on the class in the @Bean approach
public class PaymentService {
    public void pay() {
        System.out.println("Paying... ");
    }
}
```

#### **How it works:**

- Spring loads AppConfig because it's @Configuration.
- It calls the paymentService() method and **registers the returned object** as a bean.
- CommandLineRunner is injected with that bean and calls pay().

## Key difference (in one line)

- @Component: **Spring constructs** the object (no new in your code).
- @Bean: **You construct** the object inside the method (you usually use new, or a factory).

## Important

Don't keep **both** definitions for the same type (PaymentService) at the same time in one app, or you'll have bean conflicts. Choose **either** @Component **or** @Bean.

If you want, I can share a **Maven/Gradle skeleton** or a **REST endpoint version** (instead of CommandLineRunner) that calls pay() on GET /pay.

## Dependency Injection

We have used NotificationServices interface example here to explain us the tightly coupled scenario. Because we created 2 classes - SmsNotification and EmailNotification which implements the interface, and their method overrides it.

But in the main method, we create

```
@Override & Anuj Kumar Sharma *
public void run(String... args) throws Exception {
    NotificationService notificationServiceObj = new SmsNotificationService();

    notificationServiceObj.send("hello");
}
```

Here, to change implementation of NotificationServices to either email, or phone, we have to make changes in the object creation of interface line. So the Main Class is tightly coupled with the NotificationService we use.

So, for loose coupling we have to create a Bean. But we can't create bean of interface, because we can't create object of an interface. SO give component annotation in email or smsService classes, and give @Autowired annotation in main class where we define NotificationService interface object (dependency injection, But field dependency is happening here and not recommended, Constructor dependency is more preferred than this, will be discussed later)



The screenshot shows a Java development environment with the following details:

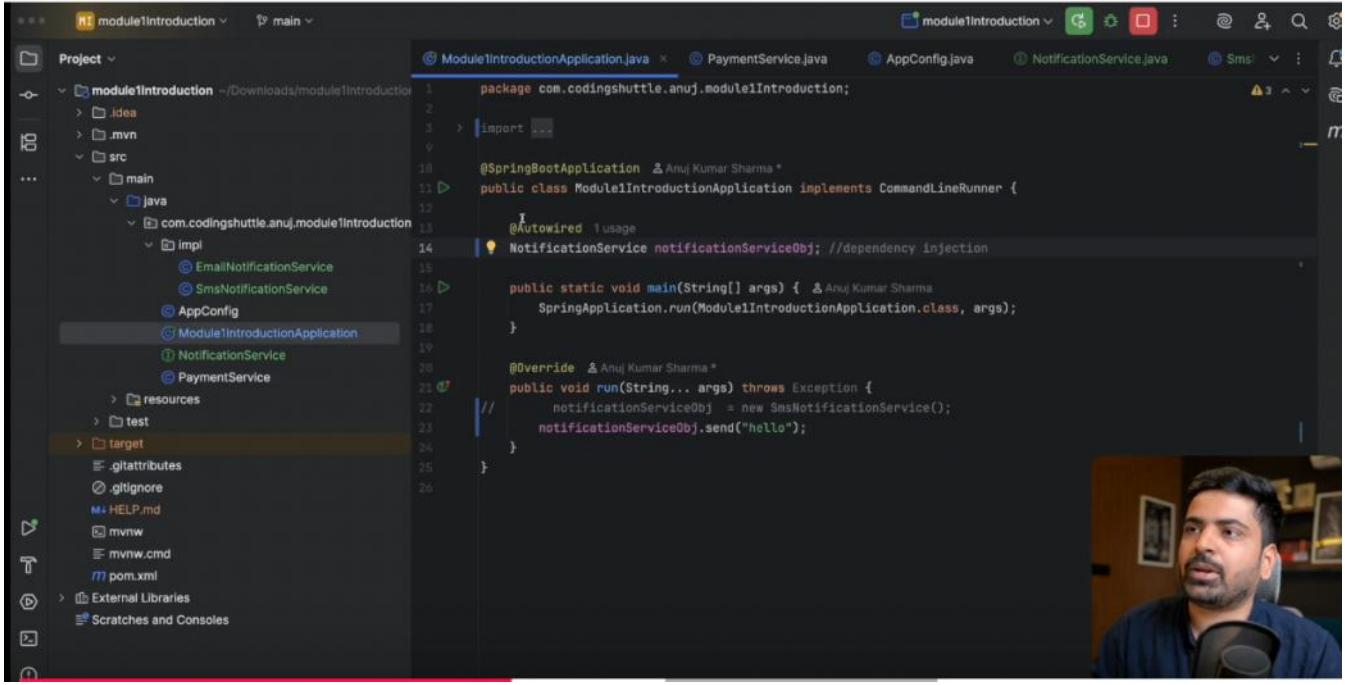
- Project Structure:** The project is named "module1Introduction". The "main" directory contains ".idea", ".mvn", and "src". The "src/main/java" package contains "com.codingshuttle.anuj.module1Introduction" which includes "EmailNotificationService", "SmsNotificationService", "AppConfig", "Module1IntroductionApplication", "NotificationService", and "PaymentService".
- Code Editor:** The code editor displays "Module1IntroductionApplication.java". It includes annotations like @SpringBootApplication, @Override, and @Autowired. It defines a main method and a run method that calls a notification service.
- Terminal:** The bottom pane shows the command-line output of the application's execution, displaying logs from INFO level messages.

If u give component in both classes - email and sms - ambiguity

Then make one of em as @Primary

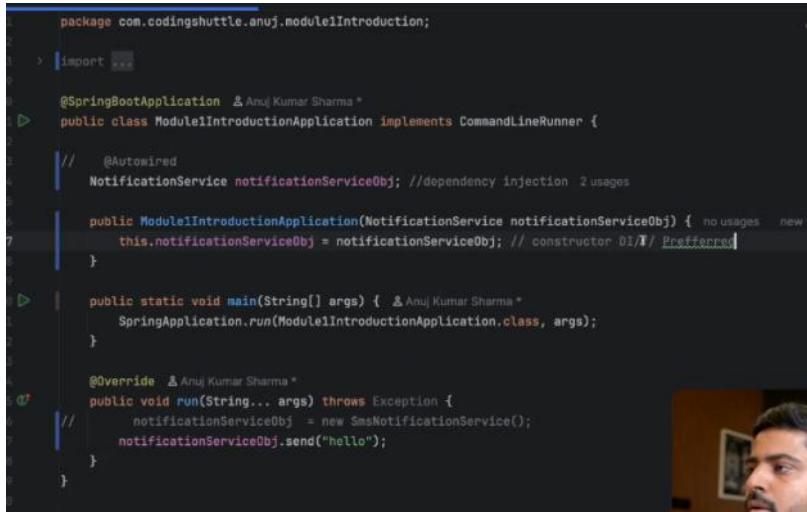
We should use constructor dependencies.

Now, we can see Dependency injection is not happening in constructor, instead happening in field (field dependency injection)



```
1 package com.codingshuttle.anuj.module1Introduction;
2
3 > import ...
4
5 @SpringBootApplication & Anuj Kumar Sharma *
6 public class Module1IntroductionApplication implements CommandLineRunner {
7
8     // @Autowired
9     NotificationService notificationServiceObj; //dependency injection 2 usages
10
11     public Module1IntroductionApplication(NotificationService notificationServiceObj) { no usages new
12         this.notificationServiceObj = notificationServiceObj; // constructor DI / Preferred
13     }
14
15     public static void main(String[] args) { & Anuj Kumar Sharma
16         SpringApplication.run(Module1IntroductionApplication.class, args);
17     }
18
19     @Override & Anuj Kumar Sharma *
20     public void run(String... args) throws Exception {
21         notificationServiceObj = new SmsNotificationService();
22         notificationServiceObj.send("hello");
23     }
24
25 }
```

So, we are doing constructor dependency now :



```
1 package com.codingshuttle.anuj.module1Introduction;
2
3 > import ...
4
5 @SpringBootApplication & Anuj Kumar Sharma *
6 public class Module1IntroductionApplication implements CommandLineRunner {
7
8     // @Autowired
9     NotificationService notificationServiceObj; //dependency injection 2 usages
10
11     public Module1IntroductionApplication(NotificationService notificationServiceObj) { no usages new
12         this.notificationServiceObj = notificationServiceObj; // constructor DI / Preferred
13     }
14
15     public static void main(String[] args) { & Anuj Kumar Sharma
16         SpringApplication.run(Module1IntroductionApplication.class, args);
17     }
18
19     @Override & Anuj Kumar Sharma *
20     public void run(String... args) throws Exception {
21         notificationServiceObj = new SmsNotificationService();
22         notificationServiceObj.send("hello");
23     }
24
25 }
```

It is preferred because now u can make the notificationService object as final (cant change after initialization)

So we are making objects immutable

# How Spring Resolves Dependencies

1. By Type → EmailService
2. If multiple beans → use name or @Qualifier
3. If still ambiguous → use @Primary

```

@Bean
@Qualifier("smtp")
public EmailService smtpEmailService() { ... }

@Bean
@Qualifier("sendgrid")
public EmailService sendgridEmailService() { ... }

public OrderService(@Qualifier("sendgrid") EmailService emailService) { ... }

```



We can use qualifiers like these, instead of using @Primary beans.



```

.java  AppConfigurer.java  NotificationService.java  SmsNotificationService.java  EmailNotificationService.java ×

1 package com.codingshuttle.anuj.module1Introduction.impl;
2
3 import com.codingshuttle.anuj.module1Introduction.NotificationService;
4 import org.springframework.beans.factory.annotation.Qualifier;
5 import org.springframework.context.annotation.Primary;
6 import org.springframework.stereotype.Component;
7
8 // @Primary
9 @Component
10 @Qualifier("emailNotif")
11
12 public class EmailNotificationService implements NotificationService {
13     @Override
14     public void send(String message) {
15         System.out.println("Email sending... "+message);
16     }
17 }
18
19

```

Also, instead of qualifiers also,

# Dependency Injection Misc.

Get all bean instances:

```
@Autowired  
private Map<String, PaymentGateway> gateways; // bean name -> instance
```

919874780718

rka8125@gmail.com

Optional Beans

```
public OrderService(Optional<AnalyticsService> analytics) { ... }
```

```
@Autowired(required = false)  
private BackupService backupService; // null if not present
```

We can provide conditional statements to Beans also in application.properties, like notification.type=email.

Then use @ConditionalOnProperties on EmailNotificationService class then.

```
//@Primary  
@Component usage new *  
@Qualifier("emailNotif")  
@ConditionalOnProperty(name = "notification.type", value = "email")  
public class EmailNotificationService implements NotificationService {  
  
    @Override usage new *  
    public void send(String message) {  
        System.out.println("Email sending... "+message);  
    }  
}
```

# Dependency Injection Misc.

Get all bean instances:

```
@Autowired  
private Map<String, PaymentGateway> gateways; // bean name -> instance
```

919874780718

rka8125@gmail.com

Optional Beans

```
public OrderService(Optional<AnalyticsService> analytics) { ... }
```

```
@Autowired(required = false)  
private BackupService backupService; // null if not present
```

Now consider I wanna send notification to both email and sms, then? I wanna get hold of all the implementations. Then in Main Class,

```
@Autowired 1 usage
Map<String, NotificationService> notificationServiceMap = new HashMap<>();

public static void main(String[] args) { & Anuj Kumar Sharma *
    SpringApplication.run(Module1IntroductionApplication.class, args);
}

@Override & Anuj Kumar Sharma *
public void run(String... args) throws Exception {
//    notificationServiceObj = new SmsNotificationService();
//    notificationServiceObj.send("hello");

    for (var notificationService: notificationServiceMap.entrySet()) {
        System.out.println(notificationService.getKey());
        notificationService.getValue().send("Hello");
    }
}
```

Let me break this down in a **lucid and easy-to-grasp way**:

## What is Spring MVC?

Spring MVC is a **web framework** in the Spring ecosystem that follows the **Model-View-Controller (MVC)** pattern.\ This pattern helps **separate concerns**:

- **Model** → Data + Business Logic
- **View** → UI (HTML, JSP, Thymeleaf)
- **Controller** → Handles requests and connects Model & View

## Core Idea

The **DispatcherServlet** is the **front controller**.\\ Think of it as the **traffic cop**:

- It receives all HTTP requests.
- Decides which controller should handle them.
- Collects the response (Model + View) and sends it back.

## Spring MVC Flow (Step-by-Step)

1. Client sends a request → e.g., /home
2. DispatcherServlet intercepts it
3. HandlerMapping finds the right controller method
4. Controller executes logic → returns ModelAndView
5. ViewResolver picks the correct view (e.g., home.jsp)
6. Response sent back to client

## Example Controller

```
@Controller
public class HomeController {
    @RequestMapping("/home")
    public ModelAndView home() {
        ModelAndView mav = new ModelAndView();
```

```

        mav.setViewName("home"); // View name
        mav.addObject("message", "Welcome to Spring MVC!"); // Model data
        return mav;
    }
}

```

- URL /home → handled by home() method.
- Returns **ModelAndView** → combines **data + view name**.

## Key Components

- **DispatcherServlet** → Front Controller
- **HandlerMapping** → Maps URL to controller
- **Controller** → Processes request
- **ModelAndView** → Holds data + view name
- **ViewResolver** → Finds the actual view file

### In short:

Spring MVC = **DispatcherServlet + MVC pattern + flexible mappings**\ It makes web apps **organized, scalable, and easy to maintain**.

☞ Do you want me to create a **diagram of the Spring MVC flow** (with arrows showing how request moves through DispatcherServlet, Controller, Model, View)? Or a **one-page cheat sheet** summarizing all components?

JPA dependency and H2 dependency added for JPA.

You will notice that we are repeating the code for Employee Entity and Employee DTO, but DTO is for security purposes.

`@PathVariable` is used for mandatory values that are a part of the URL path, typically to identify a specific resource (e.g., `/users/{id}`). `@RequestParam` is used for optional [ ] values from the query string, which are appended after a `?` and used for filtering or other parameters (e.g., `/products?category=electronics`). The choice between them depends [ ] on whether the parameter is an identifier within the resource's hierarchy or a filter for [ ] that resource. [ ]

## The Presentation Layer, DTO and Controllers

Here's the Spring Boot code for the two files you requested:

## 1. EmployeeDTO.java

This is the Data Transfer Object (DTO) with the specified attributes, getters, setters, and constructors.

```
package com.example.demo.dto;
import java.time.LocalDate;
public class EmployeeDTO {
    private Long id;
    private String name;
    private String email;
    private int age;
    private LocalDate dateOfJoining;
    private boolean isActive;
    // Default constructor
    public EmployeeDTO() {
    }
    // Parameterized constructor
    public EmployeeDTO(Long id, String name, String email, int age, LocalDate dateOfJoining, boolean
isActive) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.age = age;
        this.dateOfJoining = dateOfJoining;
        this.isActive = isActive;
    }
    // Getters and Setters
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public LocalDate getDateOfJoining() {
        return dateOfJoining;
    }
    public void setDateOfJoining(LocalDate dateOfJoining) {
        this.dateOfJoining = dateOfJoining;
    }
    public boolean isActive() {
        return isActive;
    }
    public void setActive(boolean active) {
```

```
    isActive = active;
}
}
```

## 2. EmployeeController.java

This controller defines a @GetMapping endpoint that returns a dummy EmployeeDTO object.

```
package com.example.demo.controller;
importController;
import java.time.LocalDate;
@RestController
public class EmployeeController {
    @GetMapping("/employee")
    public EmployeeDTO getEmployee() {
        // Returning dummy employee details
        return new EmployeeDTO(
            1L,
            "John Doe",
            "john.doe@example.com",
            30,
            LocalDate.of(2020, 5, 15),
            true
        );
    }
}
```

This will create a simple Spring Boot REST API with one endpoint: **GET /employee** → returns a JSON representation of the dummy EmployeeDTO.

Do you want me to also provide:

- **pom.xml dependencies** for Spring Boot setup?
- A **complete project structure** (with Application.java)?
- Or **add Swagger/OpenAPI documentation** for the endpoint?

## The Persistence Layer and JPA Repository

Absolutely! Here are the three files you asked for—clean, ready to drop into a Spring Boot project.

### 1) EmployeeEntity.java

JPA entity with fields: name, id, email, age, dateOfJoining, isActive. The id uses GenerationType.AUTO.

```
package com.example.demo.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
```

```

import java.time.LocalDate;
@Entity
@Table(name = "employees")
public class EmployeeEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private int age;
    private LocalDate dateOfJoining;
    private boolean isActive;
    // Default constructor
    public EmployeeEntity() {
    }
    // Parameterized constructor (optional)
    public EmployeeEntity(Long id, String name, String email, int age, LocalDate dateOfJoining,
    boolean isActive) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.age = age;
        this.dateOfJoining = dateOfJoining;
        this.isActive = isActive;
    }
    // Getters and setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public LocalDate getDateOfJoining() { return dateOfJoining; }
    public void setDateOfJoining(LocalDate dateOfJoining) { this.dateOfJoining = dateOfJoining; }
    public boolean isActive() { return isActive; }
    public void setActive(boolean active) { isActive = active; }
}

```

## 2) EmployeeRepository.java

Spring Data JPA repository interface.

```

package com.example.demo.repository;
import com.example.demo.entity.EmployeeEntity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {
}

```

## 3) EmployeeController.java

Constructor-based dependency injection; GET /employees/{employeeId} returns EmployeeEntity by ID (or null if not found).

```

package com.example.demo.controller;
import com.example.demo.entity.EmployeeEntity;
import com.example.demo.repository.EmployeeRepository;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

```

```

import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/employees")
public class EmployeeController {
    private final EmployeeRepository employeeRepository;
    // Constructor Dependency Injection
    public EmployeeController(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }
    @GetMapping("/{employeeId}")
    public EmployeeEntity getEmployeeById(@PathVariable(name = "employeeId") Long id) {
        return employeeRepository.findById(id).orElse(null);
    }
}

```

## Optional tips (if you're wiring this up quickly)

- **Dependencies** (in pom.xml):

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

```

- **H2 quick config** (in application.properties):

```

spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true

```

- If you return LocalDate in JSON, Spring Boot/Jackson handles it out-of-the-box. If you need a custom format, add:

```

// @JsonFormat(pattern = "yyyy-MM-dd")
private LocalDate dateOfJoining;

```

Want me to add a **POST endpoint to create an employee**, or map EmployeeEntity to your EmployeeDTO for cleaner API responses?

That single line packs a lot of meaning in Spring Data JPA. Let's break it down:

## @Repository

- Marks this interface as a **Spring Bean** of type repository.
- Spring will **detect it during component scanning** and create an implementation automatically.
- Also enables **exception translation** (converts JPA exceptions into Spring's DataAccessException hierarchy).

## **public interface EmployeeRepository extends**

### **JpaRepository<EmployeeEntity, Long>**

- This declares an **interface**, not a class. You don't write any implementation.
- JpaRepository<T, ID> is a **generic interface** provided by Spring Data JPA:
  - T → The **entity type** you want to manage (EmployeeEntity).
  - ID → The **type of the primary key** in that entity (Long because your @Id field is Long).

## **What happens under the hood?**

- Spring Data JPA sees this interface and **creates a proxy implementation at runtime**.
- That implementation uses the **JPA EntityManager** to perform CRUD operations.
- All standard methods are ready for you:
  - findById(Long id)
  - findAll()
  - save(EmployeeEntity entity)
  - deleteById(Long id)
  - and many more.

## **How is it mapped to EmployeeEntity?**

- Because you passed EmployeeEntity as the first generic type, Spring knows:
  - Which **table** to query → from @Entity and @Table annotations.
  - Which **columns** to map → from the fields in EmployeeEntity.
- Hibernate (the JPA provider) translates repository calls into SQL queries for the employees table.

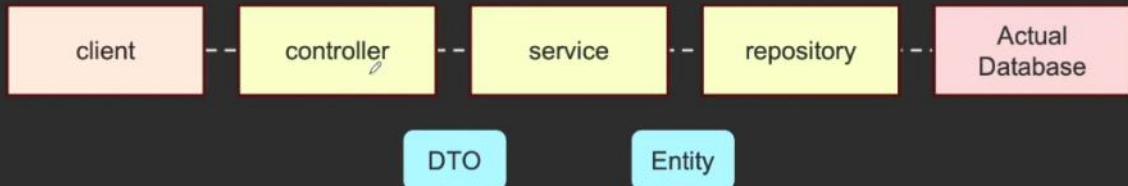
## **In short:**

This line means:

"Create a Spring-managed repository bean that provides all CRUD operations for the EmployeeEntity class, whose primary key is of type Long, without me writing any SQL or implementation."

Would you like me to **show all the methods you automatically get from JpaRepository** and also **how you can add custom query methods like findByEmail without writing SQL?**

# Spring Boot Web Project Structure



## Key roles of Service Layer

- The service layer acts as a bridge between the persistence layer (responsible for data access) and the presentation layer (handling user interaction)
- It encapsulates the business logic of the application, orchestrates interactions between different components, and provides a clean interface for external clients to interact with the system.

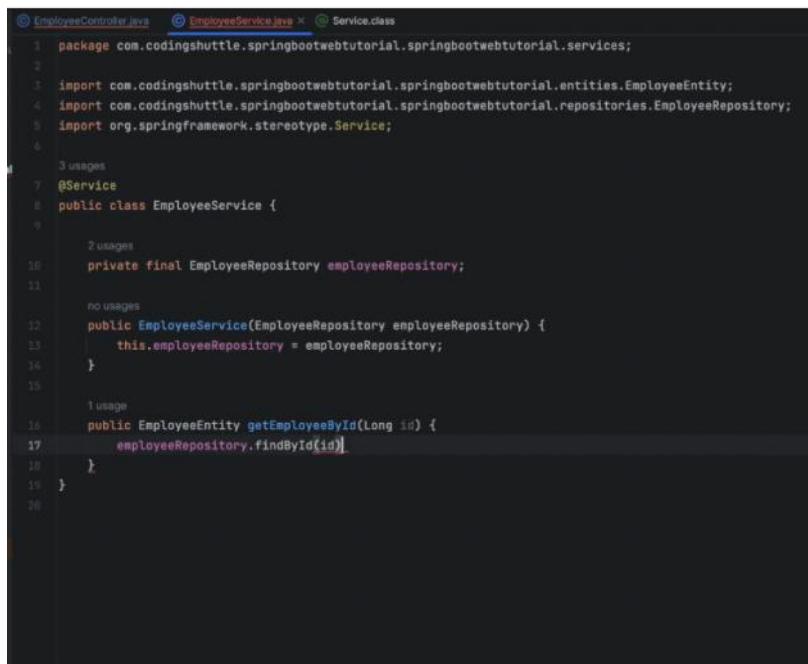


## The Service Layer

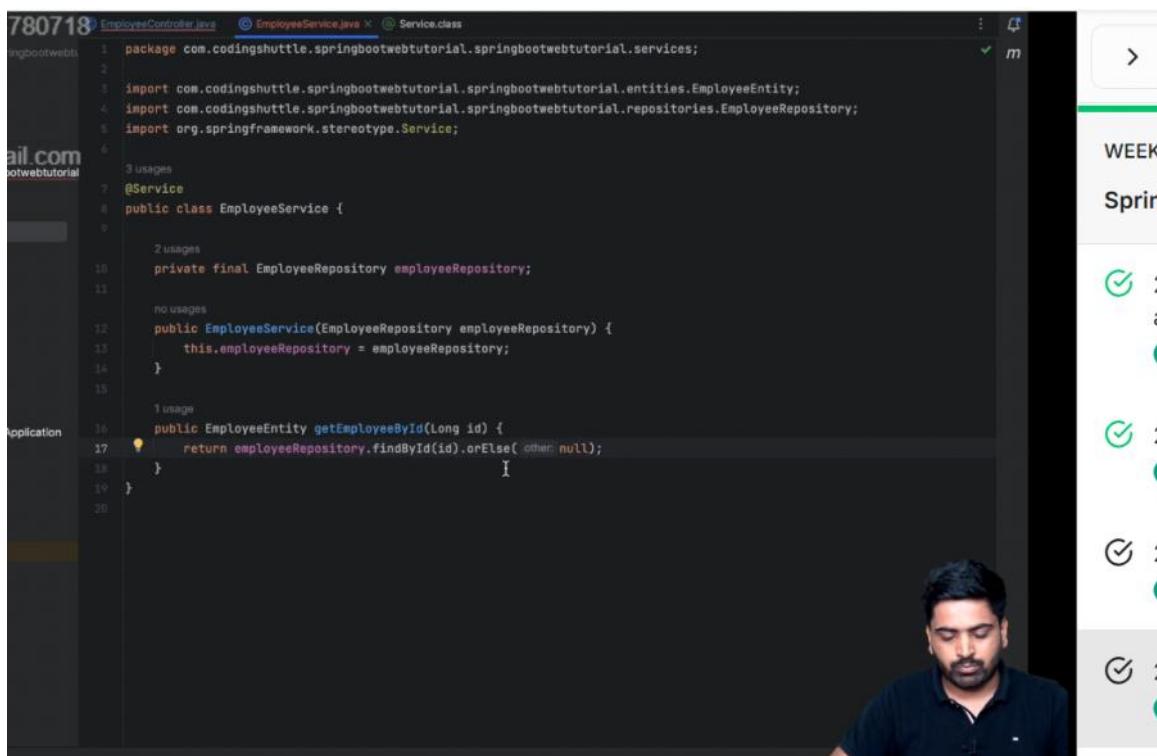
we would be removing EmployeeRepository object from EMployeeController as it is not a good practice.  
Rather add EmployeeService object there.

```
@GetMapping(path = "/{employeeId}")
public EmployeeEntity getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeService.getEmployee(id);
}
```

Use repository in service/



```
EmployeeController.java EmployeeService.java Service.class
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.services;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories.EmployeeRepository;
5 import org.springframework.stereotype.Service;
6
7 3 usages
8 @Service
9 public class EmployeeService {
10
11     2 usages
12     private final EmployeeRepository employeeRepository;
13
14     no usages
15     public EmployeeService(EmployeeRepository employeeRepository) {
16         this.employeeRepository = employeeRepository;
17     }
18
19     1 usage
20     public EmployeeEntity getEmployeeById(long id) {
21         employeeRepository.findById(id);
22     }
23 }
```



```
780718 EmployeeController.java EmployeeService.java Service.class
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.services;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories.EmployeeRepository;
5 import org.springframework.stereotype.Service;
6
7 3 usages
8 @Service
9 public class EmployeeService {
10
11     2 usages
12     private final EmployeeRepository employeeRepository;
13
14     no usages
15     public EmployeeService(EmployeeRepository employeeRepository) {
16         this.employeeRepository = employeeRepository;
17     }
18
19     1 usage
20     public EmployeeEntity getEmployeeById(Long id) {
21         return employeeRepository.findById(id).orElse(null);
22     }
23 }
```

Creating a new employee

EmployeeController.java



```
no usages - Anil Kumar Sharma
@PostMapping
public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity inputEmployee) {
    return employeeService.createNewEmployee(inputEmployee);
}
```

EmployeeService.java

```
1 usage
public EmployeeEntity createNewEmployee(EmployeeEntity inputEmployee) {
    return employeeRepository.save(inputEmployee);
}
```

So we can see controller is completely stripped of the repository and we are usig service in controller. And the service is using repository.

But in the controller, we are stil using Entity. But we shudn't, rather we should use DTO.

Add model Mapper dependency.So, in employeeService, use:

```
1 usage [related item]
public EmployeeDTO getEmployeeById(Long id) {
    EmployeeEntity employeeEntity = employeeRepository.findById(id).orElse( other null);
    ModelMapper mapper = new ModelMapper();
    return mapper.map(employeeEntity, EmployeeDTO.class);
}
```

### EmployeeController

```
no usages - new
@.GetMapping(path = "/{employeeId}")
public EmployeeDTO getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeService.getEmployeeById(id);
}
```

### MapperConfig

```
EmployeeController.java EmployeeService.java MapperConfig.java X EmployeeEntity.java pom.xml (springbootwebtutorial)
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.configs;
2
3
4 import org.modelmapper.ModelMapper;
5 import org.springframework.context.annotation.Bean;
6
7 no usages
8 public class MapperConfig {
9
10     no usages
11     @Bean
12     public ModelMapper getModelMapper() {
13         return new ModelMapper();
14     }
15 }
```

Now constructor inject modelmapper object in EmployeeService

Nice work, Arka — the base setup is clean. ☺ You're absolutely right: it's better to **introduce a service layer** and **return DTOs** from the controller rather than exposing your JPA entities directly.

Below is a drop-in refactor with:

- EmployeeService + EmployeeServiceImpl
  - EmployeeDTO (with validation)
  - A simple EmployeeMapper (manual, no extra dependency)
  - Controller refactored to use the service and return DTOs
  - A POST /employees endpoint that returns 201 Created and sets a Location header
  - Minimal exception handling (404 for not found)
- If you'd prefer automatic mapping, I can swap in **ModelMapper** or **MapStruct**—just say the word.

## 1) DTO

### src/main/java/com/example/demo/dto/EmployeeDTO.java

```
package com.example.demo.dto;
import com.fasterxml.jackson.annotation.JsonFormat;
import jakarta.validation.constraints.*;
import java.time.LocalDate;
public class EmployeeDTO {
    private Long id;
    @NotBlank(message = "Name is required")
    private String name;
    @NotBlank(message = "Email is required")
    @Email(message = "Email must be valid")
    private String email;
    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 70, message = "Age must be at most 70")
    private Integer age;
    @PastOrPresent(message = "Date of joining cannot be in the future")
    @JsonFormat(pattern = "yyyy-MM-dd")
    private LocalDate dateOfJoining;
    @NotNull(message = "Active flag is required")
    private Boolean active;
    public EmployeeDTO() {}
    public EmployeeDTO(Long id, String name, String email, Integer age, LocalDate dateOfJoining,
Boolean active) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.age = age;
        this.dateOfJoining = dateOfJoining;
        this.active = active;
    }
    // Getters and setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public Integer getAge() { return age; }
```

```

public void setAge(Integer age) { this.age = age; }
public LocalDate getDateOfJoining() { return dateOfJoining; }
public void setDateOfJoining(LocalDate dateOfJoining) { this.dateOfJoining = dateOfJoining; }
public Boolean getActive() { return active; }
public void setActive(Boolean active) { this.active = active; }
}

```

## 2) Mapper (manual)

### **src/main/java/com/example/demo/mapper/EmployeeMapper.java**

```

package com.example.demo.mapper;
import com.example.demo.dto.EmployeeDTO;
import com.example.demo.entity.EmployeeEntity;
public final class EmployeeMapper {
    private EmployeeMapper() {}
    public static EmployeeDTO toDTO(EmployeeEntity entity) {
        if (entity == null) return null;
        return new EmployeeDTO(
            entity.getId(),
            entity.getName(),
            entity.getEmail(),
            entity.getAge(),
            entity.getDateOfJoining(),
            entity.isActive()
        );
    }
    public static EmployeeEntity toEntity(EmployeeDTO dto) {
        if (dto == null) return null;
        EmployeeEntity entity = new EmployeeEntity();
        entity.setId(dto.getId()); // usually null on create
        entity.setName(dto.getName());
        entity.setEmail(dto.getEmail());
        entity.setAge(dto.getAge() != null ? dto.getAge() : 0);
        entity.setDateOfJoining(dto.getDateOfJoining());
        entity.setActive(dto.getActive() != null && dto.getActive());
        return entity;
    }
    public static void copyToEntity(EmployeeDTO dto, EmployeeEntity entity) {
        if (dto.getName() != null) entity.setName(dto.getName());
        if (dto.getEmail() != null) entity.setEmail(dto.getEmail());
        if (dto.getAge() != null) entity.setAge(dto.getAge());
        if (dto.getDateOfJoining() != null) entity.setDateOfJoining(dto.getDateOfJoining());
        if (dto.getActive() != null) entity.setActive(dto.getActive());
    }
}

```

## 3) Service Layer

### **src/main/java/com/example/demo/service/EmployeeService.java**

```

package com.example.demo.service;
import com.example.demo.dto.EmployeeDTO;
import java.util.List;
public interface EmployeeService {
    EmployeeDTO getById(Long id);
    EmployeeDTO create(EmployeeDTO dto);
    List<EmployeeDTO> getAll();
    EmployeeDTO update(Long id, EmployeeDTO dto);
}

```

```

    void delete(Long id);
}

src/main/java/com/example/demo/service/impl/EmployeeServiceimpl.java

package com.example.demo.service.impl;
import com.example.demo.dto.EmployeeDTO;
import com.example.demo.entity.EmployeeEntity;
import com.example.demo.exception.ResourceNotFoundException;
import com.example.demo.mapper.EmployeeMapper;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.stream.Collectors;
@Service
public class EmployeeServiceimpl implements EmployeeService {
    private final EmployeeRepository employeeRepository;
    public EmployeeServiceimpl(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }
    @Override
    public EmployeeDTO getById(Long id) {
        EmployeeEntity entity = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with id " + id));
        return EmployeeMapper.toDTO(entity);
    }
    @Override
    public EmployeeDTO create(EmployeeDTO dto) {
        EmployeeEntity toSave = EmployeeMapper.toEntity(dto);
        toSave.setId(null); // ensure auto-generated
        EmployeeEntity saved = employeeRepository.save(toSave);
        return EmployeeMapper.toDTO(saved);
    }
    @Override
    public List<EmployeeDTO> getAll() {
        return employeeRepository.findAll().stream()
            .map(EmployeeMapper::toDTO)
            .collect(Collectors.toList());
    }
    @Override
    public EmployeeDTO update(Long id, EmployeeDTO dto) {
        EmployeeEntity existing = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with id " + id));
        EmployeeMapper.copyToEntity(dto, existing);
        EmployeeEntity updated = employeeRepository.save(existing);
        return EmployeeMapper.toDTO(updated);
    }
    @Override
    public void delete(Long id) {
        EmployeeEntity existing = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with id " + id));
        employeeRepository.delete(existing);
    }
}

```

## 4) Exceptions

### **src/main/java/com/example/demo/exception/ResourceNotFoundException.java**

```

package com.example.demo.exception;
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) { super(message); }
}
src/main/java/com/example/demo/exception/GlobalExceptionHandler.java

```

```

package com.example.demo.exception;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import java.util.HashMap;
import java.util.Map;
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Map<String, Object>> handleNotFound(ResourceNotFoundException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("error", "Not Found");
        body.put("message", ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
    }
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, Object>>
    handleValidation(MethodArgumentNotValidException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("error", "Validation Failed");
        Map<String, String> fieldErrors = new HashMap<>();
        ex.getBindingResult().getFieldErrors()
            .forEach(err -> fieldErrors.put(err.getField(), err.getDefaultMessage()));
        body.put("details", fieldErrors);
        return ResponseEntity.badRequest().body(body);
    }
}

```

## 5) Controller (refactored)

```

src/main/java/com/example/demo/controller/EmployeeController.java

```

```

package com.example.demo.controller;
import com.example.demo.dto.EmployeeDTO;
import com.example.demo.service.EmployeeService;
import jakarta.validation.Valid;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponentsBuilder;
import java.util.List;
@RestController
@RequestMapping("/employees")
public class EmployeeController {
    private final EmployeeService employeeService;
    // Constructor Dependency Injection
    public EmployeeController(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }
    // GET /employees/{id}
    @GetMapping("/{employeeId}")
    public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable("employeeId") Long id) {

```

```

EmployeeDTO dto = employeeService.getById(id);
return ResponseEntity.ok(dto);
}
// GET /employees
@GetMapping
public ResponseEntity<List<EmployeeDTO>> getAllEmployees() {
    return ResponseEntity.ok(employeeService.getAll());
}
// POST /employees
@PostMapping
public ResponseEntity<EmployeeDTO> createEmployee(
    @Valid @RequestBody EmployeeDTO employeeDTO,
    UriComponentsBuilder uriBuilder) {
    EmployeeDTO created = employeeService.create(employeeDTO);
    return ResponseEntity
        .created(uriBuilder.path("/employees/{id}").buildAndExpand(created.getId()).toUri())
        .body(created);
}
// PUT /employees/{id}
@PutMapping("/{employeeId}")
public ResponseEntity<EmployeeDTO> updateEmployee(
    @PathVariable("employeeId") Long id,
    @Valid @RequestBody EmployeeDTO employeeDTO) {
    return ResponseEntity.ok(employeeService.update(id, employeeDTO));
}
// DELETE /employees/{id}
@DeleteMapping("/{employeeId}")
public ResponseEntity<Void> deleteEmployee(@PathVariable("employeeId") Long id) {
    employeeService.delete(id);
    return ResponseEntity.noContent().build();
}
}

```

## 6) Your existing Entity/Repository are fine

(Keeping your EmployeeEntity and EmployeeRepository as-is.)

**src/main/java/com/example/demo/entity/EmployeeEntity.java**

a

```

package com.example.demo.entity;
import jakarta.persistence.*;
import java.time.LocalDate;
@Entity
@Table(name = "employees")
public class EmployeeEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private int age;
    private LocalDate dateOfJoining;
    private boolean isActive;
    public EmployeeEntity() {}
    public EmployeeEntity(Long id, String name, String email, int age, LocalDate dateOfJoining,
boolean isActive) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.age = age;
        this.dateOfJoining = dateOfJoining;
        this.isActive = isActive;
    }
}

```

```

}
public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }
public int getAge() { return age; }
public void setAge(int age) { this.age = age; }
public LocalDate getDateOfJoining() { return dateOfJoining; }
public void setDateOfJoining(LocalDate dateOfJoining) { this.dateOfJoining = dateOfJoining; }
public boolean isActive() { return isActive; }
public void setActive(boolean active) { isActive = active; }
}

```

## **src/main/java/com/example/demo/repository/EmployeeRepository.java**

```

package com.example.demo.repository;
import com.example.demo.entity.EmployeeEntity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
@Repository

```

## **7) pom.xml dependencies**

Add validation support (Spring Boot 3 needs this starter explicitly):

```

<dependencies>
    <!-- Web & JPA & H2 (as you had) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <!-- Validation -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
</dependencies>

```

If you choose **ModelMapper**:

```

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

```

...and we'll add a @Bean config and swap the mapper implementation.

## **8) application.properties (as you had)**

```

spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=update

```

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

## 9) Quick test with cURL

### Create an employee

```
curl -i -X POST http://localhost:8080/employees \
-H "Content-Type: application/json" \
-d '{
  "name": "Ada Lovelace",
  "email": "ada@example.com",
  "age": 30,
  "dateOfJoining": "2023-12-01",
  "active": true
}'
```

### Get by id

```
curl -i http://localhost:8080/employees/1
```

### List all

### Update

```
curl -i -X PUT http://localhost:8080/employees/1 \
-H "Content-Type: application/json" \
-d '{
  "name": "Ada L.",
  "email": "ada.l@example.com",
  "age": 31,
  "active": false
}'
```

### Delete

```
curl -i -X DELETE http://localhost:8080/employees/1
```

## Why this structure?

- **Service layer** keeps business logic away from web concerns and repositories.
- **DTOs** avoid leaking persistence details and let you shape API contracts independently.
- **Validation** ensures clean inputs at the boundary.
- **Consistent HTTP semantics** (201 Created, 404 Not Found, 204 No Content) make your API predictable.

**Want me to switch the mapper to ModelMapper or MapStruct, or add pagination/sorting to GET /employees? Also, if you prefer more REST-hardened responses (e.g., Problem Details RFC 7807), I can wire that up too.**

## ModelMapper concepts

Awesome — let's wire in **ModelMapper** and remove the manual mapper.\ Below are the exact changes you can drop into your project.

Highlights:

- ModelMapper @Bean with strict matching + skip-null mapping for updates
- Custom property maps so EmployeeEntity.isActive ↔ EmployeeDTO.active map correctly
- Service refactored to use ModelMapper for create/get/list/update

## 1) Add dependency (pom.xml)

```
<dependencies>
    <!-- existing dependencies -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <!-- validation (needed for @Valid and constraint annotations) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <!-- ModelMapper -->
    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>3.2.0</version>
    </dependency>
</dependencies>
```

## 2) Keep your DTO (no change)

src/main/java/com/example/demo/dto/EmployeeDTO.java

```
package com.example.demo.dto;
import com.fasterxml.jackson.annotation.JsonFormat;
import jakarta.validation.constraints.*;
import java.time.LocalDate;
public class EmployeeDTO {
    private Long id;
    @NotBlank(message = "Name is required")
    private String name;
    @NotBlank(message = "Email is required")
    @Email(message = "Email must be valid")
    private String email;
    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 70, message = "Age must be at most 70")
    private Integer age;
    @PastOrPresent(message = "Date of joining cannot be in the future")
    @JsonFormat(pattern = "yyyy-MM-dd")
    private LocalDate dateOfJoining;
    @NotNull(message = "Active flag is required")
    private Boolean active;
    public EmployeeDTO() {}
    public EmployeeDTO(Long id, String name, String email, Integer age, LocalDate dateOfJoining,
Boolean active) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.age = age;
        this.dateOfJoining = dateOfJoining;
        this.active = active;
    }
}
```

```

}
// Getters & setters
public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }
public Integer getAge() { return age; }
public void setAge(Integer age) { this.age = age; }
public LocalDate getDateOfJoining() { return dateOfJoining; }
public void setDateOfJoining(LocalDate dateOfJoining) { this.dateOfJoining = dateOfJoining; }
public Boolean getActive() { return active; }
public void setActive(Boolean active) { this.active = active; }
}

```

### 3) ModelMapper configuration

src/main/java/com/example/demo/config/ModelMapperConfig.java

```

package com.example.demo.config;
import com.example.demo.dto.EmployeeDTO;
import com.example.demo.entity.EmployeeEntity;
import org.modelmapper.Conditions;
import org.modelmapper.ModelMapper;
import org.modelmapper.PropertyMap;
import org.modelmapper.config.Configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration as SpringConfiguration;
@Configuration
public class ModelMapperConfig {
    @Bean
    public ModelMapper modelMapper() {
        ModelMapper mapper = new ModelMapper();
        // Strict matching to avoid accidental mappings
        mapper.getConfiguration()
            .setFieldMatchingEnabled(true)
            .setFieldAccessLevel(Configuration.AccessLevel.PRIVATE)
            .setSkipNullEnabled(true) // helpful for patch/update
            .setMatchingStrategy(org.modelmapper.convention.MatchingStrategies.STRICT);
        // Entity -> DTO: map boolean isActive -> Boolean active
        mapper.addMappings(new PropertyMap<EmployeeEntity, EmployeeDTO>() {
            @Override
            protected void configure() {
                map().setActive(source.isActive());
                // age is primitive in entity; map directly (ModelMapper handles primitives)
            }
        });
        // DTO -> Entity: map Boolean active -> boolean isActive (default false if null)
        mapper.addMappings(new PropertyMap<EmployeeDTO, EmployeeEntity>() {
            @Override
            protected void configure() {
                // If DTO.active is null, leave entity flag unchanged (for update flows).
                when(Conditions.isNotNull()).map().setActive(source.getActive());
            }
        });
    }
    return mapper;
}

```

Notes:

- `setSkipNullEnabled(true)` ensures update operations don't overwrite existing values with null unintentionally.
- We added explicit mapping for `isActive ↔ active` because the field names differ.

## 4) Service layer (refactored to use ModelMapper)

src/main/java/com/example/demo/service/EmployeeService.java

```
package com.example.demo.service;
import com.example.demo.dto.EmployeeDTO;
import java.util.List;
public interface EmployeeService {
    EmployeeDTO getById(Long id);
    EmployeeDTO create(EmployeeDTO dto);
    List<EmployeeDTO> getAll();
    EmployeeDTO update(Long id, EmployeeDTO dto);
    void delete(Long id);
}
```

src/main/java/com/example/demo/service/impl/EmployeeServiceImpl.java

```
package com.example.demo.service.impl;
import com.example.demo.dto.EmployeeDTO;
import com.example.demo.entity.EmployeeEntity;
import com.example.demo.exception.ResourceNotFoundException;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.stream.Collectors;
@Service
public class EmployeeServiceImpl implements EmployeeService {
    private final EmployeeRepository employeeRepository;
    private final ModelMapper modelMapper;
    public EmployeeServiceImpl(EmployeeRepository employeeRepository, ModelMapper
modelMapper) {
        this.employeeRepository = employeeRepository;
        this.modelMapper = modelMapper;
    }
    @Override
    public EmployeeDTO getById(Long id) {
        EmployeeEntity entity = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with id " + id));
        return modelMapper.map(entity, EmployeeDTO.class);
    }
    @Override
    public EmployeeDTO create(EmployeeDTO dto) {
        // Map DTO -> Entity, ensure id is null for auto-generation
        EmployeeEntity toSave = modelMapper.map(dto, EmployeeEntity.class);
        toSave.setId(null);
        EmployeeEntity saved = employeeRepository.save(toSave);
        return modelMapper.map(saved, EmployeeDTO.class);
    }
    @Override
    public List<EmployeeDTO> getAll() {
        return employeeRepository.findAll().stream()
            .map(e -> modelMapper.map(e, EmployeeDTO.class))
            .collect(Collectors.toList());
    }
    @Override
    public EmployeeDTO update(Long id, EmployeeDTO dto) {
        EmployeeEntity existing = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with id " + id));
        // Apply non-null fields from DTO onto existing entity
        modelMapper.map(dto, existing);
        EmployeeEntity updated = employeeRepository.save(existing);
```

```

        return modelMapper.map(updated, EmployeeDTO.class);
    }
    @Override
    public void delete(Long id) {
        EmployeeEntity existing = employeeRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Employee not found with id " + id));
        employeeRepository.delete(existing);
    }
}

```

## 5) Controller (already using service & DTO)

src/main/java/com/example/demo/controller/EmployeeController.java

```

package com.example.demo.controller;
import com.example.demo.dto.EmployeeDTO;
import com.example.demo.service.EmployeeService;
import jakarta.validation.Valid;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponentsBuilder;
import java.util.List;
@RestController
@RequestMapping("/employees")
public class EmployeeController {
    private final EmployeeService employeeService;
    // Constructor Dependency Injection
    public EmployeeController(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }
    // GET /employees/{id}
    @GetMapping("/{employeeId}")
    public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable("employeeId") Long id) {
        return ResponseEntity.ok(employeeService.getById(id));
    }
    // GET /employees
    @GetMapping
    public ResponseEntity<List<EmployeeDTO>> getAllEmployees() {
        return ResponseEntity.ok(employeeService.getAll());
    }
    // POST /employees
    @PostMapping
    public ResponseEntity<EmployeeDTO> createEmployee(
        @Valid @RequestBody EmployeeDTO employeeDTO,
        UriComponentsBuilder uriBuilder) {
        EmployeeDTO created = employeeService.create(employeeDTO);
        return ResponseEntity
            .created(uriBuilder.path("/employees/{id}").buildAndExpand(created.getId()).toUri())
            .body(created);
    }
    // PUT /employees/{id}
    @PutMapping("/{employeeId}")
    public ResponseEntity<EmployeeDTO> updateEmployee(
        @PathVariable("employeeId") Long id,
        @Valid @RequestBody EmployeeDTO employeeDTO) {
        return ResponseEntity.ok(employeeService.update(id, employeeDTO));
    }
    // DELETE /employees/{id}
    @DeleteMapping("/{employeeId}")
    public ResponseEntity<Void> deleteEmployee(@PathVariable("employeeId") Long id) {
        employeeService.delete(id);
        return ResponseEntity.noContent().build();
    }
}

```

## 6) Exceptions (unchanged)

src/main/java/com/example/demo/exception/ResourceNotFoundException.java

```
package com.example.demo.exception;
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) { super(message); }
}
```

src/main/java/com/example/demo/exception/GlobalExceptionHandler.java

```
package com.example.demo.exception;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import java.util.HashMap;
import java.util.Map;
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Map<String, Object>> handleNotFound(ResourceNotFoundException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("error", "Not Found");
        body.put("message", ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
    }
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, Object>>
    handleValidation(MethodArgumentNotValidException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("error", "Validation Failed");
        Map<String, String> fieldErrors = new HashMap<>();
        ex.getBindingResult().getFieldErrors()
            .forEach(err -> fieldErrors.put(err.getField(), err.getDefaultMessage()));
        body.put("details", fieldErrors);
        return ResponseEntity.badRequest().body(body);
    }
}
```

## 7) Entity & Repository (unchanged)

You can keep your existing EmployeeEntity and EmployeeRepository exactly as you shared.

## 8) Optional: a “PATCH-like” partial update

If you want true partial updates (only provided fields applied), keep setSkipNullEnabled(true) in the ModelMapper config (already done). Then you can add:

```
@PatchMapping("/{employeeId}")
public ResponseEntity<EmployeeDTO> patchEmployee(
    @PathVariable("employeeId") Long id,
    @RequestBody EmployeeDTO partialDto) {
    return ResponseEntity.ok(employeeService.update(id, partialDto));
(No @Valid here if you want to allow missing fields.)
```

## 9) Quick test

- Create:

```
curl -i -X POST http://localhost:8080/employees \
-H "Content-Type: application/json" \
-d '{
    "name": "Ada Lovelace",
```

```
"email": "ada@example.com",
"age": 30,
"dateOfJoining": "2023-12-01",
"active": true
• Get:
• Update:
```

```
curl -i -X PUT http://localhost:8080/employees/1 \
-H "Content-Type: application/json" \
-d '{
  "name": "Ada L.",
  "email": "ada.l@example.com",
  "age": 31,
  "active": false
```

**Want me to add pagination & sorting (e.g., GET /employees?  
page=0&size=20&sort=name,asc) or switch to MapStruct for  
compile-time safety? Also happy to add Problem Details (RFC  
7807) responses or OpenAPI/Swagger docs for your API.**

Great question! Let's go **step by step** and make this super clear for you.\ We'll cover:  
 What is ModelMapper?\  Why use it in Spring Boot?\  How to set it up (step-by-step)\  How to use it for **Entity ↔ DTO mapping**\  Bonus: How to handle custom mappings

## 1. What is ModelMapper?

ModelMapper is a Java library that **automatically maps one object to another** based on matching property names.\ Example:

- EmployeeEntity has name, email, age
- EmployeeDTO also has name, email, age\ ModelMapper can copy values from Entity → DTO or DTO → Entity **without writing boilerplate code**.

## 2. Why use it in Spring Boot?

- Avoid writing repetitive dto.setName(entity.getName()) code.
- Keeps your code clean and maintainable.
- Works well with layered architecture (Controller → Service → Repository).

## 3. Step-by-Step Setup

### Step 1: Add dependency in pom.xml

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
```

<version>3.2.0</version>

## Step 2: Create a Config class

This makes ModelMapper a Spring Bean so you can inject it anywhere.

```
package com.example.demo.config;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class ModelMapperConfig {
    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

Now you can use @Autowired ModelMapper mapper; in your service.

## Step 3: Create DTO and Entity

Example:

```
public class EmployeeDTO {
    private Long id;
    private String name;
    private String email;
    private Integer age;
    private Boolean active;
    // getters & setters
}

@Entity
public class EmployeeEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private int age;
    private boolean isActive;
    // getters & setters
}
```

## Step 4: Use ModelMapper in Service

```
@Service
public class EmployeeService {
    private final EmployeeRepository employeeRepository;
    private final ModelMapper modelMapper;
    public EmployeeService(EmployeeRepository employeeRepository, ModelMapper modelMapper) {
        this.employeeRepository = employeeRepository;
        this.modelMapper = modelMapper;
    }
    public EmployeeDTO getEmployee(Long id) {
        EmployeeEntity entity = employeeRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Not found"));
        return modelMapper.map(entity, EmployeeDTO.class);
    }
    public EmployeeDTO createEmployee(EmployeeDTO dto) {
        EmployeeEntity entity = modelMapper.map(dto, EmployeeEntity.class);
        EmployeeEntity saved = employeeRepository.save(entity);
        return modelMapper.map(saved, EmployeeDTO.class);
    }
}
```

```
    }
}
 No manual mapping needed!
```

## 4. Custom Mapping (for different field names)

If your DTO has active and Entity has isActive, you need to configure:

```
@Bean
public ModelMapper modelMapper() {
    ModelMapper mapper = new ModelMapper();
    mapper.typeMap(EmployeeEntity.class, EmployeeDTO.class)
        .addMapping(EmployeeEntity::isActive, EmployeeDTO::setActive);
    mapper.typeMap(EmployeeDTO.class, EmployeeEntity.class)
        .addMapping(EmployeeDTO::getActive, EmployeeEntity:: setActive);
    return mapper;
}
```

## 5. Bonus: Skip Nulls for Partial Updates

```
mapper.getConfiguration().setSkipNullEnabled(true);
This means if DTO has null fields, they won't overwrite existing entity values.
```

### Summary

- Add dependency
- Create @Configuration class with @Bean ModelMapper
- Inject ModelMapper in Service
- Use mapper.map(source, Destination.class)
- Add custom mappings if field names differ
- Enable skipNull for updates

### Next Steps

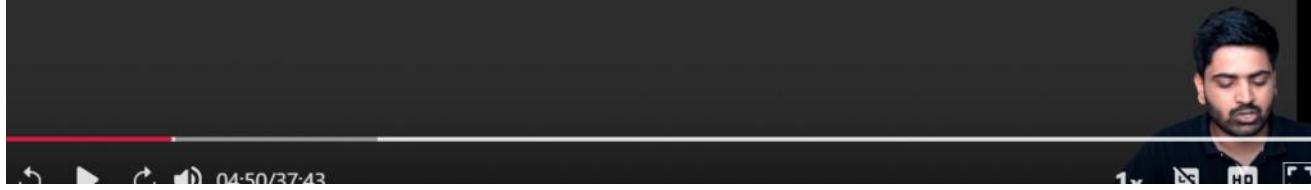
Do you want me to:  
✓ Show full working Spring Boot project structure with ModelMapper?  
✓ Or explain Java Streams & Lambdas (since you asked earlier) in the context of mapping?  
✓ Or give you unit test examples for ModelMapper mappings?

Which one would help you most right now?

## Annotated Controllers

Spring MVC provides an annotation-based programming model where `@Controller` and `@RestController` components use annotations to express request mappings, request input, exception handling, and more.

The `@RestController` annotation is a shorthand for `@Controller` and `@ResponseBody`, meaning all methods in the controller will return JSON/XML directly to the response body.



## Request Mappings

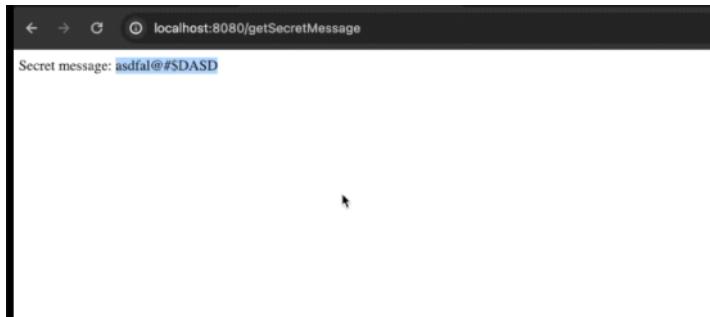
arka8135@gmail.com

You can use the `@RequestMapping` annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.



```
com.codingshuttle.springbootwebtutorial
```

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.controllers;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 no usages
7 @RestController
8 public class EmployeeController {
9
10     no usages
11     @GetMapping(path = "/getSecretMessage")
12     public String getMySuperSecretMessage() {
13         return "Secret message: asdfal@#SDASD";
14     }
15 }
```



```
Spring-Boot-Starter-Web-3.0.pom
```

```
1 package com.codingshuttle.springbootwebtutorial;
2
3 import java.time.LocalDate;
4
5 no usages
6 public class EmployeeDTO {
7
8     2 usages
9     private Long id;
10    2 usages
11    private String name;
12    2 usages
13    private String email;
14    2 usages
15    private Integer age;
16    2 usages
17    private LocalDate dateOfJoining;
18    2 usages
19    private Boolean isActive; I
20
21    no usages
22    public Long getId() {
23        return id;
24    }
25
26    no usages
27    public void setId(Long id) {
28        this.id = id;
29    }
30
31    no usages
32    public String getName() {
33        return name;
34    }
35 }
```



```
EmployeeDTO.java
1 no usages
2 public class EmployeeDTO {
3
4     3 usages
5     private Long id;
6     3 usages
7     private String name;
8     3 usages
9     private String email;
10    3 usages
11    private Integer age;
12    3 usages
13    private LocalDate dateOfJoining;
14    3 usages
15    private Boolean isActive;
16
17    no usages
18    public EmployeeDTO() {
19
20    }
21
22    no usages
23    public EmployeeDTO(Long id, String name, String email, Integer age, LocalDate dateOfJoining, Boolean isActive) {
24        this.id = id;
25        this.name = name;
26        this.email = email;
27        this.age = age;
28        this.dateOfJoining = dateOfJoining;
29        this.isActive = isActive;
30    }
31
32
33
34
35
36
```

How to get data from URL? 2 ways - `@PathVariable` and `@RequestParam`

## Dynamic URLs Paths

### `@PathVariable`

`/employees/123`

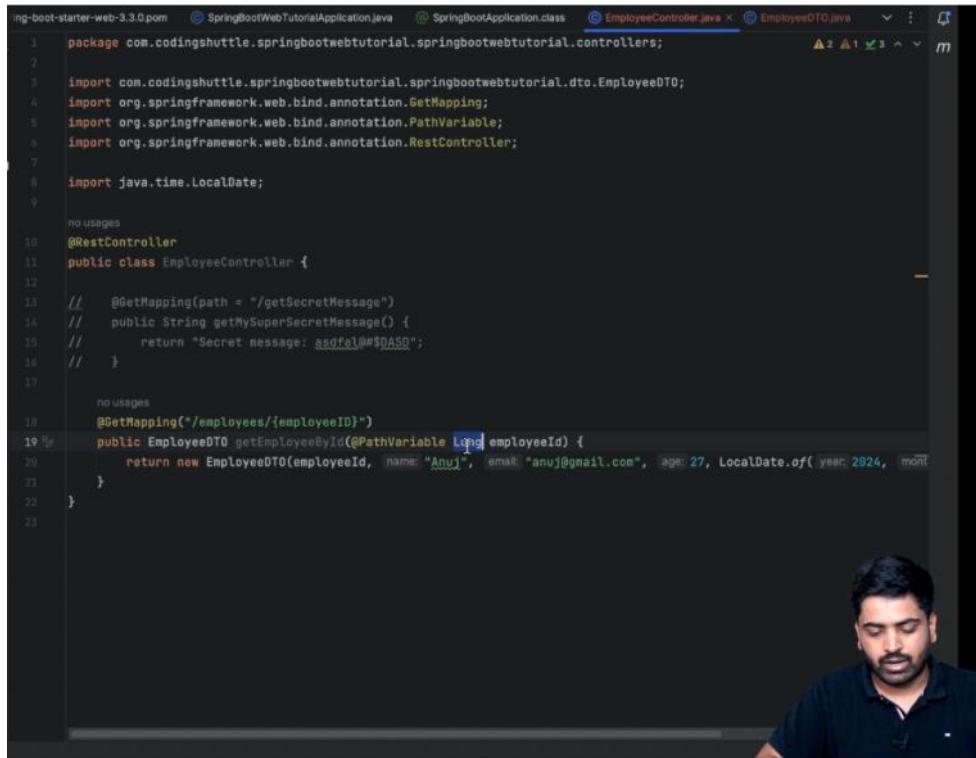
Use path variables when the parameter is an essential part of the URL path that identifies a resource.

### `@RequestParam`

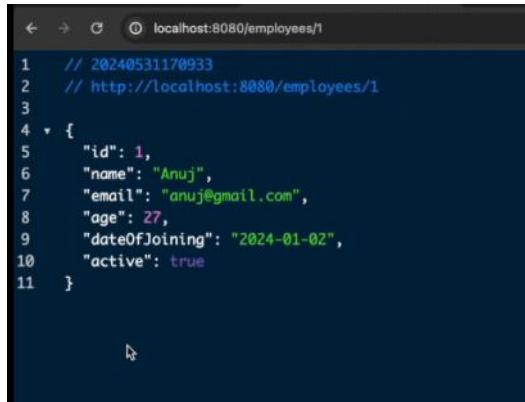
`/employees?id=123`

Use query parameters when the parameter is optional and used for filtering, sorting, or other modifications to the request.

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.controllers;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.dto.EmployeeDTO;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RestController;
7
8 no usages
9 @RestController
10 public class EmployeeController {
11
12     // @GetMapping(path = "/getSecretMessage")
13     // public String getMySuperSecretMessage() {
14     //     return "Secret message: asdfal@#DASD";
15     // }
16
17     no usages
18     @GetMapping("/employees/{employeeID}")
19     public EmployeeDTO getEmployeeById(@PathVariable String employeeId) {
20
21         }
22     }
23
```



```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.controllers;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.dto.EmployeeDTO;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import java.time.LocalDate;
9
10 no usages
11 @RestController
12 public class EmployeeController {
13
14     // @GetMapping(path = "/getSecretMessage")
15     // public String getMySuperSecretMessage() {
16     //     return "Secret message: asdfal@#DASD";
17     // }
18
19     no usages
20     @GetMapping("/employees/{employeeID}")
21     public EmployeeDTO getEmployeeById(@PathVariable Long employeeId) {
22         return new EmployeeDTO(employeeId, name: "Anuj", email: "anuj@gmail.com", age: 27, LocalDate.of( year: 2024, month: 1, day: 2));
23     }
24 }
```



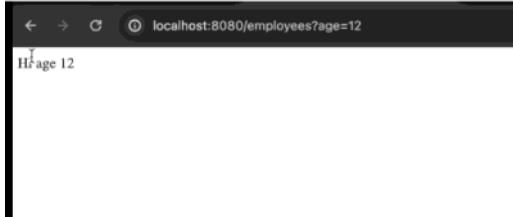
```
← → ⌂ localhost:8080/employees/1
1 // 20240531170933
2 // http://localhost:8080/employees/1
3
4 {
5     "id": 1,
6     "name": "Anuj",
7     "email": "anuj@gmail.com",
8     "age": 27,
9     "dateOfJoining": "2024-01-02",
10    "active": true
11 }
```

See, we didn't convert our response from Java object to JSON, but `@RestController` did it for us. Jackson converted it.

@RequestParam

```
no usages
@GetMapping(path = "/employees")
public String getAllEmployees(@RequestParam Integer age) {
    return "Hi age "+age;
}

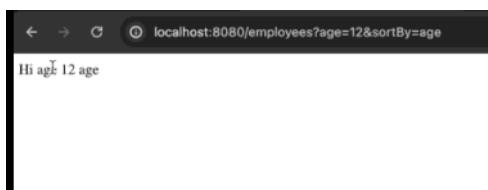
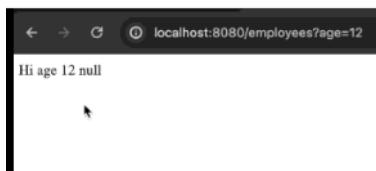
}
```



Making parameters optional:

```
no usages
@GetMapping(path = "/employees")
public String getAllEmployees(@RequestParam(required = false) Integer age,
                             @RequestParam(required = false) String sortBy) {
    return "Hi age "+age+" "+sortBy;
}

}
```



Better representation would be using @RequestMapping

```
no usages
@RestController
@RequestMapping(path = "/employees")
public class EmployeeController {
    /**
     * @GetMapping(path = "/getSecretMessage")
     * public String getMySuperSecretMessage() {
     *     return "Secret message: asdfal@#$ASD0";
     * }
    */

    @GetMapping(path = "/{employeeId}")
    public EmployeeDTO getEmployeeById(@PathVariable Long employeeId) {
        return new EmployeeDTO(employeeId, name: "Anuj", email: "anuj@gmail.com", age: 27, LocalDate.of( year: 2024, month: 10, day: 1));
    }

    no usages
    @GetMapping
    public String getAllEmployees(@RequestParam(required = false) Integer age,
                                @RequestParam(required = false) String sortBy) {
    }
}
```

```
23
24     no usages
25     @GetMapping
26     public String getAllEmployees(@RequestParam(required = false) Integer age,
27         @RequestParam(required = false) String sortBy) {
28         return "Hi age "+age+" "+sortBy;
29     }
30 }
31
32 }
```

We can also change path variable names.

```
5 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.dto.EmployeeDTO;
6 import org.springframework.web.bind.annotation.*;
7
8 import java.time.LocalDate;
9
10 no usages
11 @RestController
12 @RequestMapping(path = "/employees")
13 public class EmployeeController {
14
15     // @GetMapping(path = "/getSecretMessage")
16     // public String getMySuperSecretMessage() {
17     //     return "Secret message: asdfal#@$DASD";
18     // }
19
20     no usages
21     @GetMapping(path = "{employeeId}")
22     public EmployeeDTO getEmployeeById(@PathVariable(name = "employeeId") Long id) {
23         return new EmployeeDTO(id, name: "Anuj", email: "anuj@gmail.com", age: 27, LocalDate.of(year: 2024, month: 1, day: 1));
24     }
25     no usages
26     @GetMapping
27     public String getAllEmployees(@RequestParam(required = false) Integer Age,
28         @RequestParam(required = false) String sortBy) {
29         return "Hi age "+age+" "+sortBy;
30     }
31 }
32 }
```

```
1 no usages
2 @GetMapping
3 public String getAllEmployees(@RequestParam(required = false, name = "inputAge") Integer age,
4     @RequestParam(required = false) String sortBy) {
5     return "Hi age "+age+" "+sortBy;
6 }
7
8 }
```

InputAge in the url will only work

```
no usages
@PostMapping
public String createNewEmployee() {
    return "Hello from POST";
}

no usages
@PutMapping String updateEmployeeById() {
    return "Hello from Put";
}

}
```

PUT http://localhost:8080/employees

http://localhost:8080/employees

Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 91 ms Size: 178 B Save as example

Pretty Raw Preview Visualize Text

```
1 Hello from Put
```

## Creating a new employee

We will provide id by myself.

```
no usages
@PostMapping
public EmployeeDTO createNewEmployee(@RequestBody EmployeeDTO inputEmployee) {
    inputEmployee.setId(100L);
    return inputEmployee;
}
```

In production, we are supposed to save this in db.

In postman, you have to pass employee details in body (raw, JSON)

POST http://localhost:8080/employees

http://localhost:8080/employees

Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Body

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2     "name": "Ram",
3     "email": "ram@codingshuttle.com",
4     "age": 23
5 }
```

arka8135@gmail.com

Body Cookies Headers (5) Test Results Status: 200 OK Time: 2 ms

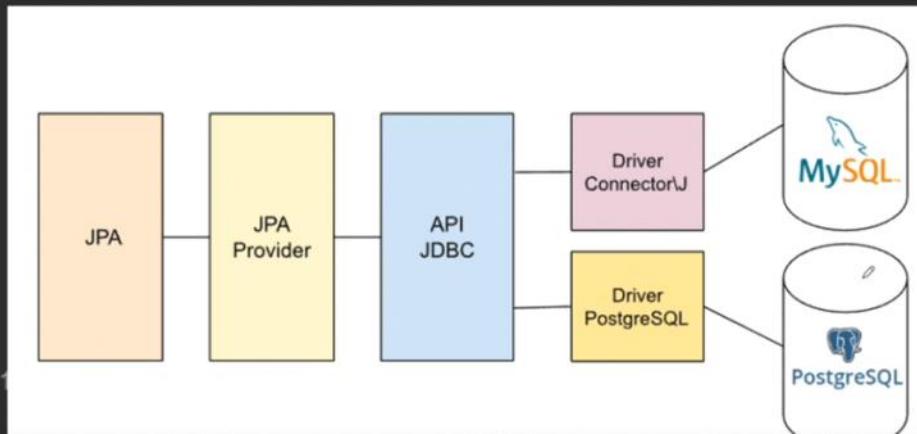
Pretty Raw Preview Visualize JSON

```
1 {
2     "id": 100,
3     "name": "Ram",
4     "email": "ram@codingshuttle.com",
5     "age": 23,
6     "dateOfJoining": null,
7     "active": null
8 }
```

See here, id is provided by us.

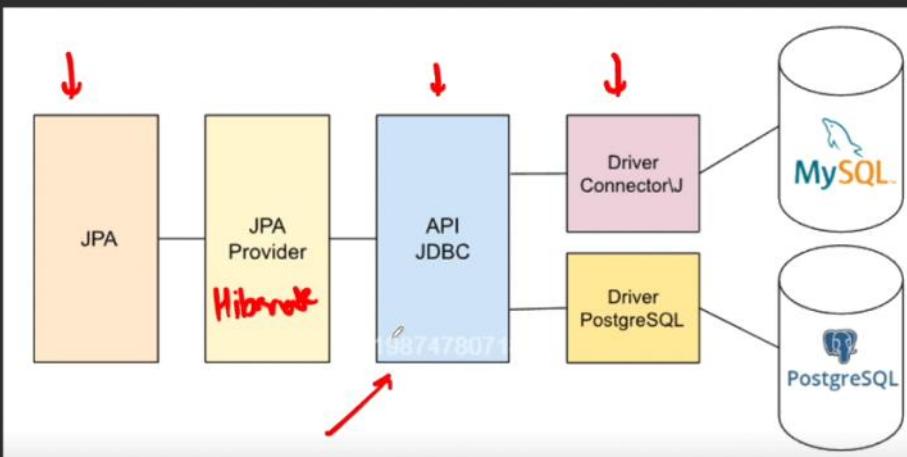
## The Persistence layer and JPA Repository

### Java Persistence API - JPA



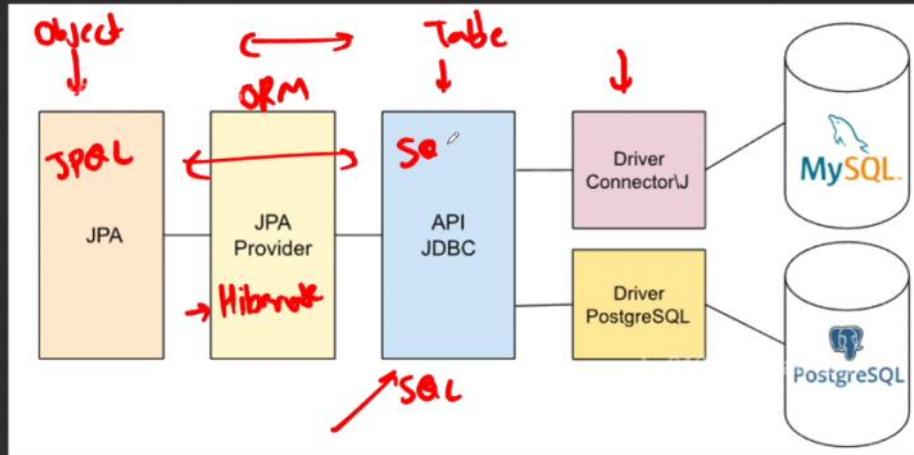
### Java Persistence API - JPA

arka8135@gmail.com



We can perform direct SQL queries in JDBC

# Java Persistence API - JPA



Resource

Notes

All

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Google

h2 maven dependency

X | ⌂ | ⌂ | ⌂ | ⌂

All Images Videos Shopping News More Tools

Maven Repository

<https://mvnrepository.com/artifact/h2>

[com.h2database » h2 - Maven Repository](#)

Dependency Injection · XML Processing · Web ... H2 Database Engine. A fast SQL database that can ... maven mobile module npm osgi plugin resources rlang adk server ...

2.1.210

... h2 --> <dependency> <groupId>com.h2database</groupId>

2.1.214

... h2 --> <dependency> <groupId>com.h2database</groupId>

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.2.224</version>
    <scope>test</scope>
</dependency>
```

No need to give version as it was already given earlier

```
29
30
31
32
33
34
```

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    </dependency>
```

Now we got both jpa and h2 dependency added.

Now we are creating EmployeeEntity which will create Entity Table.

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities;
2
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.Table;
5
6 no usages
7 @Entity
8 @Table(name = "employees")
9 public class EmployeeEntity {
10
11 }
12
```

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities;
2
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.Table;
5
6 import java.time.LocalDate;
7
8 @Entity
9 @Table(name = "employees")
10 public class EmployeeEntity {
11     no usages
12     private Long id;
13     no usages
14     private String name;
15     no usages
16     private String email;
17     no usages
18     private Integer age;
19     no usages
20     private LocalDate dateOfJoining;
21     no usages
22     private Boolean isActive;
23 }
```

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities;
2
3 import jakarta.persistence.*;
4
5 import java.time.LocalDate;
6
7 no usages
8 @Entity
9 @Table(name = "employees")
10 public class EmployeeEntity {
11
12     no usages
13     @Id
14     @GeneratedValue(strategy = GenerationType.AUTO)
15     private Long id;
16     no usages
17     private String name;
18     no usages
19     private String email;
20     no usages
21     private Integer age;
22     no usages
23     private LocalDate dateOfJoining;
24     no usages
25     private Boolean isActive;
26 }
```

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 @Repository
7 public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {}
```

We will need a service layer that will connect to Repository.

Controller shouldn't be directly connected to the Repository.

Controller should be connected to service.

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6
7 @Repository
8 public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {}
```

## Employeecontroller

```
15 //     return "Secret message: ${refId}@${ASD}";
16 // }
17
18 /**
19  * @RequiredArgsConstructor
20  */
21 public EmployeeController(EmployeeRepository employeeRepository) {
22     this.employeeRepository = employeeRepository;
23 }
```

Required type: EmployeeDTO  
Provided: Optional<com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity>  
Make 'getEmployeeById()' return 'java.util.Optional<com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity>' More actions...

If id is null, this is handled by Java, don't worry.

```
no usages + Anuj Kumar Sharma *
@GetMapping(path = "/{employeeId}")
public EmployeeDTO getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeRepository.findById(id).orElse(null);
}

Required type: EmployeeDTO
Provided: EmployeeEntity

no usage: More actions... ↗ ↘
```

```
@GetMapping("getEmployeeById")
public EmployeeEntity getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeRepository.findById(id).orElse(null);
}
```

Here, Repository is giving me an Entity, but return type is DTO.

So, we are using now return type with Entity.

```
// }

2 usages
private final EmployeeRepository employeeRepository;

no usages: new *
public EmployeeController(EmployeeRepository employeeRepository) {
    this.employeeRepository = employeeRepository;
}

no usages: new *
@GetMapping(path = "/{employeeId}")
public EmployeeEntity getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeRepository.findById(id).orElse(null);
}
```

Returning list of all employees

```
no usages: new *
@GetMapping
public List<EmployeeEntity> getAllEmployees(@RequestParam(required = false, name = "inputAge") Integer age,
                                              @RequestParam(required = false) String sortBy) {
    return employeeRepository.findAll();
}
```

```
no usages: new *
@PostMapping
public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity inputEmployee) {
    return employeeRepository.save(inputEmployee);
}
```

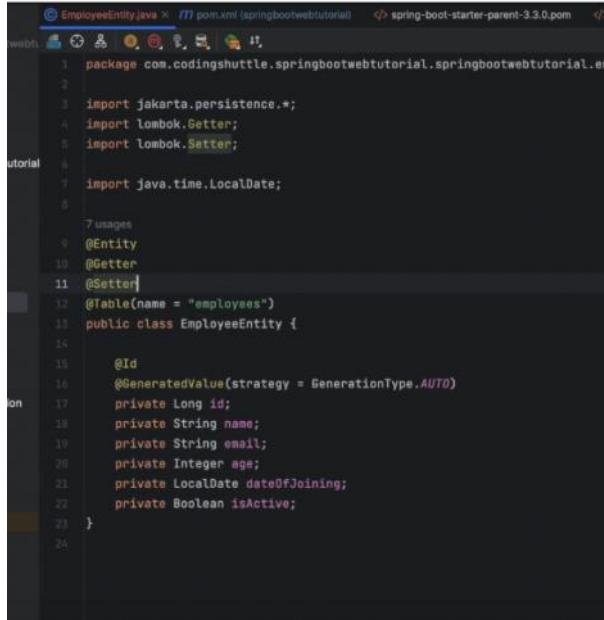
The screenshot shows the Postman interface with a request to `http://localhost:8080/employees`. The request method is `POST`. In the `Body` tab, the raw JSON payload is:

```
1 {  
2     "name": "Ram",  
3     "email": "ram@codingshuttle.com",  
4     "age": 23,  
5     "dateOfJoining": "2024-01-01",  
6     "isActive": true  
7 }
```

The response at the bottom shows a status of `200 OK` with a size of `166 B`.

Now add Lombok dependency

Now we can define getters and setters easily



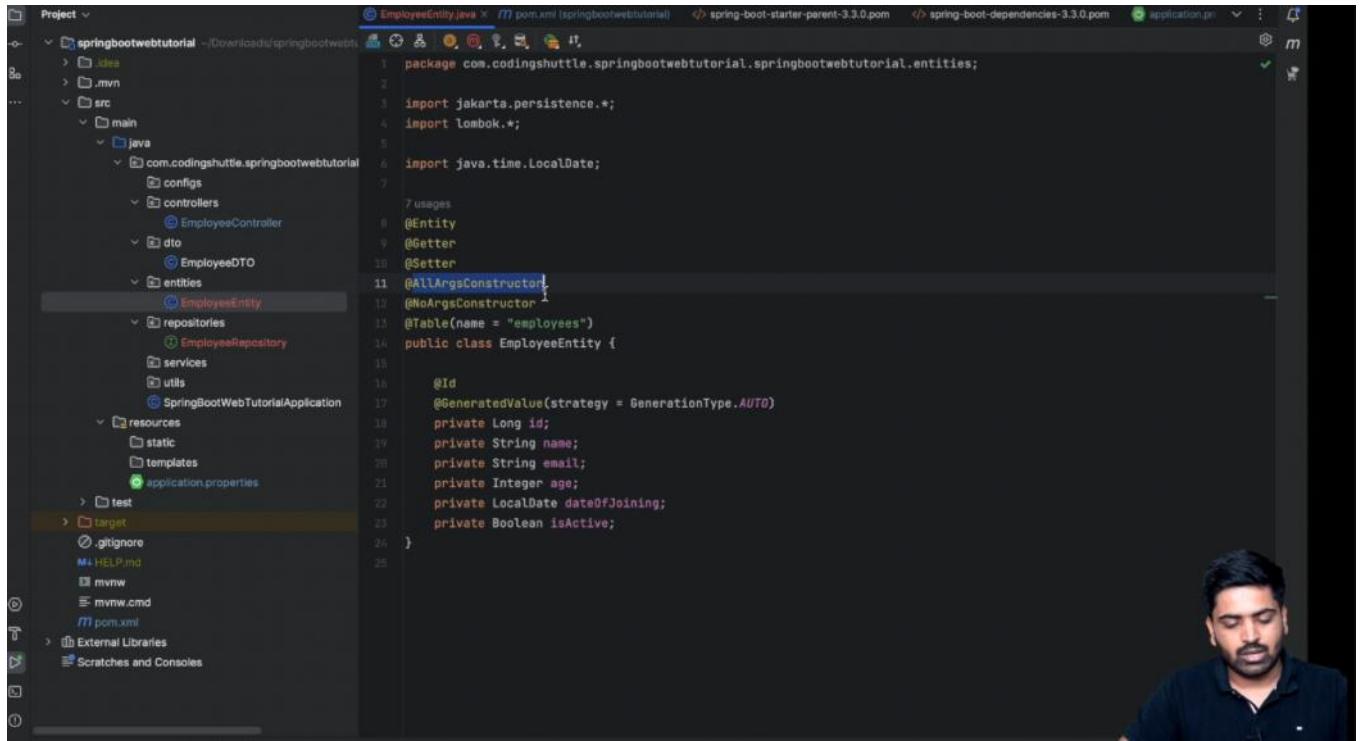
```
EmployeeEntity.java
package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import java.time.LocalDate;

import static jakarta.persistence.GenerationType.AUTO;

@Entity
@Getter
@Setter
@Table(name = "employees")
public class EmployeeEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private Integer age;
    private LocalDate dateOfJoining;
    private Boolean isActive;
}
```



Project

- springbootwebtutorial
  - src
    - main
      - java
        - com.codingshuttle.springbootwebtutorial
          - controllers
          - configs
          - EmployeeController
          - dto
          - EmployeeDTO
          - entities
          - EmployeeEntity
          - repositories
          - EmployeeRepository
          - services
          - utils
        - SpringBootWebTutorialApplication
      - resources
        - static
        - templates
        - application.properties
      - test
      - target
    - Maven
    - pom.xml
    - External Libraries
    - Scratches and Consoles

```
EmployeeEntity.java
package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities;

import jakarta.persistence.*;
import lombok.*;  
import java.time.LocalDate;  
  
import static jakarta.persistence.GenerationType.AUTO;  
  
@Entity  
@Getter  
@Setter  
@Table(name = "employees")  
public class EmployeeEntity {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    private String name;  
    private String email;  
    private Integer age;  
    private LocalDate dateOfJoining;  
    private Boolean isActive;  
}
```

Now post request will work.

The screenshot shows the Postman application interface. A POST request is being made to `http://localhost:8080/employees`. The request body is a JSON object:

```
1 {
2   "name": "Ram",
3   "email": "ram@codingshuttle.com",
4   "age": 23,
5   "dateOfJoining": "2024-01-01",
6   "isActive": true
7 }
```

The response status is 200 OK, with a response body:

```
1 [
2   {
3     "id": 1,
4     "name": "Ram",
5     "email": "ram@codingshuttle.com",
6     "age": 23,
7     "dateOfJoining": "2024-01-01",
8     "isActive": true
9   }
10 ]
```

The screenshot shows the Postman application interface. A GET request is being made to `http://localhost:8080/employees`. The response status is 200 OK, with a response body:

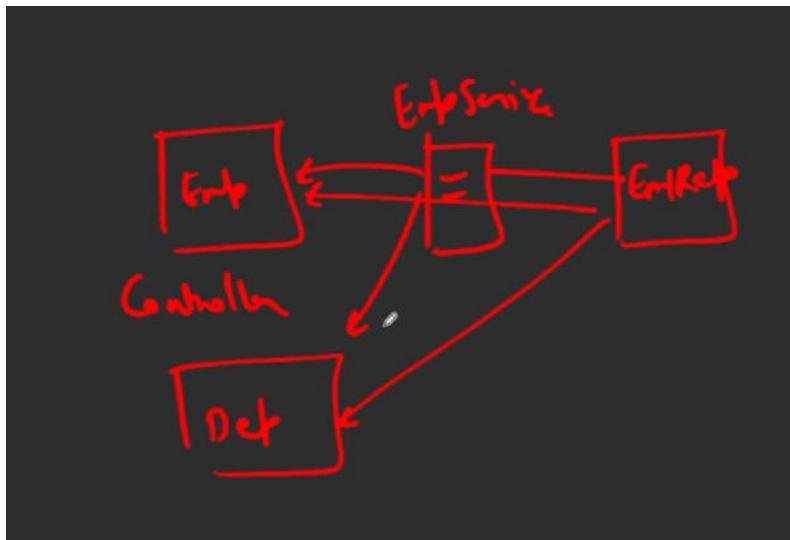
```
1 [
2   {
3     "id": 1,
4     "name": "Ram",
5     "email": "ram@codingshuttle.com",
6     "age": 23,
7     "dateOfJoining": "2024-01-01",
8     "isActive": true
9   },
10   {
11     "id": 2,
12     "name": "Ohan",
13     "email": "ohan@codingshuttle.com",
14     "age": 23,
15     "dateOfJoining": "2024-01-01",
16     "isActive": false
17   }
18 ]
```

Here data saving in in-memory database.

## The Service Layer - writing our business logic

# Key roles of Service Layer

- The service layer acts as a bridge between the persistence layer (responsible for data access) and the presentation layer (handling user interaction)
- It encapsulates the business logic of the application, orchestrates interactions between different components, and provides a clean interface for external clients to interact with the system.



So, previously we performed constructor injection of EmployeeRepository inside EmployeeController, but now we would perform constructor injection of EmployeeService inside EmployeeController.

```
21
22
23
24
25
26
27
28
29
30
31
```

```
2 usages
private final EmployeeService employeeService;

no usages * new *
public EmployeeController(EmployeeService employeeService) {
    this.employeeService = employeeService;
}

no usages * Anuj Kumar Sharma *
@GetMapping(path = "/{employeeId}")
public EmployeeEntity getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeService.getEmployeeById(id);
}
```

Cmd+ Enter and it will create a getEmployeeById() method in EmployeeService class.

Now we will perform constructor injection of EmployeeRepository inside EmployeeService.

```
2
3
4
5
6
7
8
9
```

```
import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories.EmployeeRepository;
import org.springframework.stereotype.Service;

3 usages
```

```
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories.EmployeeRepository;
5 import org.springframework.stereotype.Service;
6
7 3 usages
8 @Service
9 public class EmployeeService {
10
11     2 usages
12     private final EmployeeRepository employeeRepository;
13
14     no usages
15     public EmployeeService(EmployeeRepository employeeRepository) {
16         this.employeeRepository = employeeRepository;
17     }
18
19     1 usage
20     public EmployeeEntity getEmployeeById(Long id) {
21         return employeeRepository.findById(id);
22     }
23
24 }
```

```
EmployeeController.java EmployeeService.java - Service class
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.services;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories.EmployeeRepository;
5 import org.springframework.stereotype.Service;
6
7 3 usages
8 @Service
9 public class EmployeeService {
10
11     2 usages
12     private final EmployeeRepository employeeRepository;
13
14     no usages
15     public EmployeeService(EmployeeRepository employeeRepository) {
16         this.employeeRepository = employeeRepository;
17     }
18
19     1 usage
20     public EmployeeEntity getEmployeeById(Long id) {
21         return employeeRepository.findById(id).orElse(null);
22     }
23
24 }
```

Now again in EMployeeController

```
no usages ▾ Anuj Kumar Sharma *
@GetMapping
public List<EmployeeEntity> getAllEmployees(@RequestParam(required = false, name = "inputAge") Integer age,
                                              @RequestParam(required = false) String sortBy) {
    ②     return employeeService.getAllEmployees();
}
```

```
EmployeeController.java EmployeeService.java Service.class
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.services;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEntity;
4 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.repositories.EmployeeRepository;
5 import org.springframework.stereotype.Service;
6
7 import java.util.List;
8
9 3 usages
10 @Service
11 public class EmployeeService {
12
13     3 usages
14     private final EmployeeRepository employeeRepository;
15
16     no usages
17     public EmployeeService(EmployeeRepository employeeRepository) {
18         this.employeeRepository = employeeRepository;
19     }
20
21     1 usage
22     public EmployeeEntity getEmployeeById(Long id) {
23         return employeeRepository.findById(id).orElse(null);
24     }
25
26 }
```

```
no usages ▾ Anuj Kumar Sharma
@PostMapping
public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity inputEmployee) {
    return employeeService.createNewEmployee(inputEmployee);
}
```

```
1 usage
26     public EmployeeEntity createNewEmployee(EmployeeEntity inputEmployee) {
27         return employeeRepository.save(inputEmployee);
28     }
29 }
```

But we are still using EMployeeEntity in controller, but rather we should use EmployeeDTO.

### EmployeeService

```
1 usage 1 related problem
public EmployeeDTO getEmployeeById(Long id) {
    EmployeeEntity employeeEntity = employeeRepository.findById(id).orElse(null);
    return new EmployeeDTO(employeeEntity.getId(), employeeEntity.getName(), employeeEntity.getEmail());
}
```

In the above example, we changed the return type to EMployeeDTO, but since EmployeeRepository returns an entity, we are converting it into DTO in later step.

But good way would be to use a Model Mapper. Give model mapper dependency in pom.xml

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.8.0</version>
</dependency>
```

Now we can use this model mapper to convert any class to any other object.  
This looks for all fields which are common in both.

EmployeeService:

```
1 usage 1 related problem
public Employee getEmployeeById(Long id) {
    EmployeeEntity employeeEntity = employeeRepository.findById(id).orElse( other: null);
    ModelMapper mapper = new ModelMapper();
    return mapper.map(employeeEntity, EmployeeDTO.class);
}
```

Now since we would be using modelMapper again and again for all methods, why not make a bean of it.

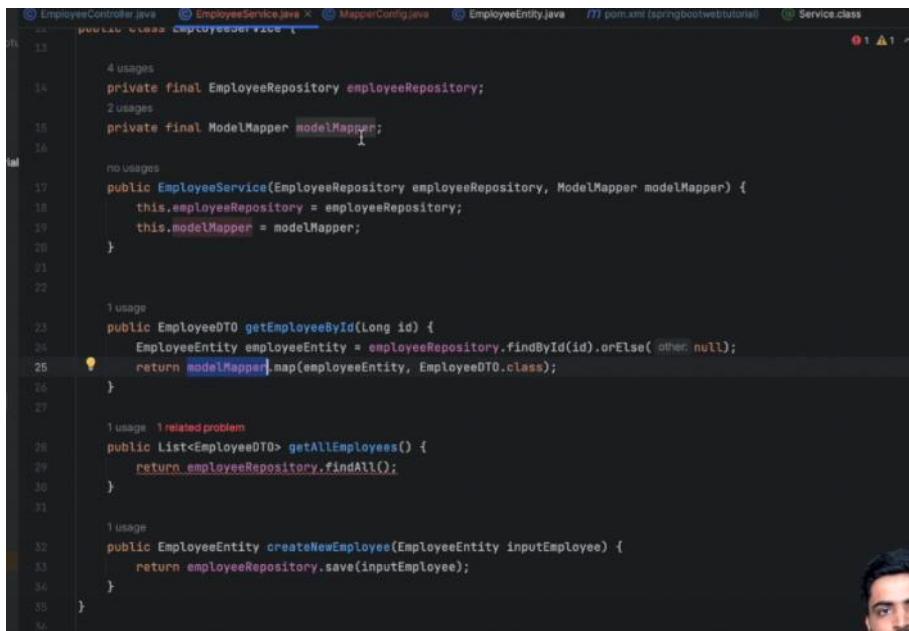
MapperConfig.java

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.configs;
2
3
4 import org.modelmapper.ModelMapper;
5 import org.springframework.context.annotation.Bean;
6
7 no usages
8 public class MapperConfig {
9
10     no usages
11     @Bean
12     public ModelMapper getModelMapper() {
13         return new ModelMapper();
14     }
15
16 }
```

Now constructor injection dependency this model mapper in EmployeeService.

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.services;
2
3 import java.util.List;
4
5 3 usages
6 @Service
7 public class EmployeeService {
8
9     5 usages
10     private final EmployeeRepository employeeRepository;
11     1 usage
12     private final ModelMapper modelMapper;
13
14     no usages
15     public EmployeeService(EmployeeRepository employeeRepository, ModelMapper modelMapper) {
16         this.employeeRepository = employeeRepository;
17         this.modelMapper = modelMapper;
18     }
19
20     |
21 }
```

We are successfully doing dependency injection.



```
public class EmployeeService {
    private final EmployeeRepository employeeRepository;
    private final ModelMapper modelMapper;

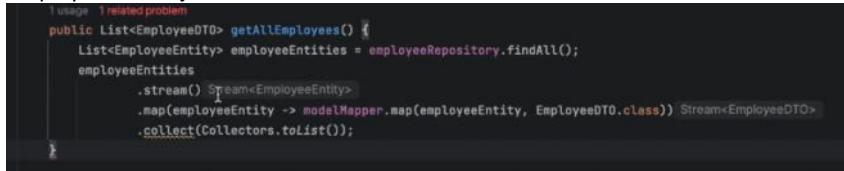
    public EmployeeService(EmployeeRepository employeeRepository, ModelMapper modelMapper) {
        this.employeeRepository = employeeRepository;
        this.modelMapper = modelMapper;
    }

    public EmployeeDTO getEmployeeById(Long id) {
        EmployeeEntity employeeEntity = employeeRepository.findById(id).orElse(null);
        return modelMapper.map(employeeEntity, EmployeeDTO.class);
    }

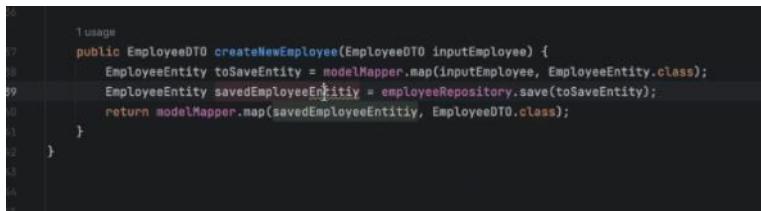
    public List<EmployeeDTO> getAllEmployees() {
        return employeeRepository.findAll();
    }

    public EmployeeEntity createNewEmployee(EmployeeEntity inputEmployee) {
        return employeeRepository.save(inputEmployee);
    }
}
```

### EmployeeService.java

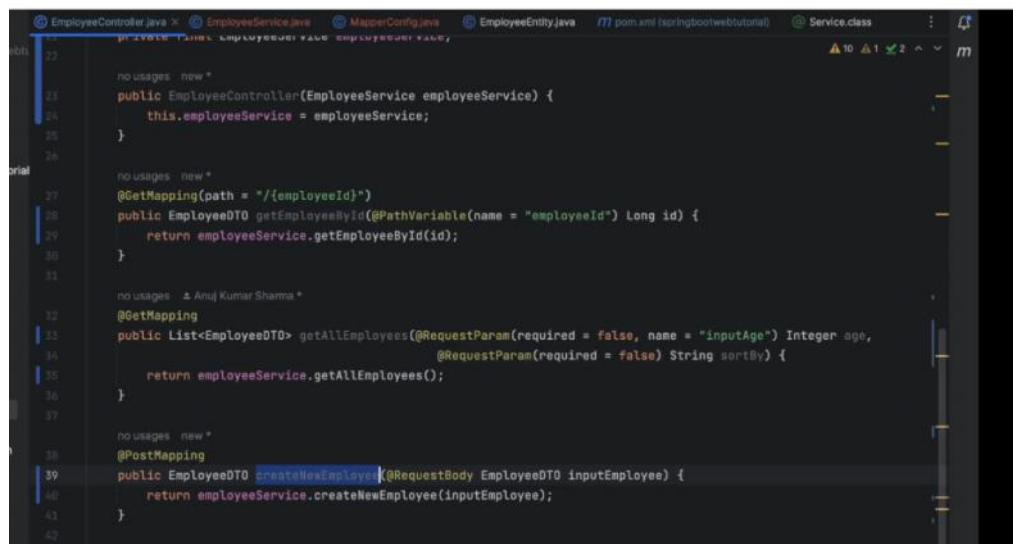


```
public List<EmployeeDTO> getAllEmployees() {
    List<EmployeeEntity> employeeEntities = employeeRepository.findAll();
    employeeEntities
        .stream()
        .map(employeeEntity -> modelMapper.map(employeeEntity, EmployeeDTO.class))
        .collect(Collectors.toList());
}
```



```
public EmployeeDTO createNewEmployee(EmployeeDTO inputEmployee) {
    EmployeeEntity toSaveEntity = modelMapper.map(inputEmployee, EmployeeEntity.class);
    EmployeeEntity savedEmployeeEntity = employeeRepository.save(toSaveEntity);
    return modelMapper.map(savedEmployeeEntity, EmployeeDTO.class);
}
```

### EmployeeController.java



```
public EmployeeController(EmployeeService employeeService) {
    this.employeeService = employeeService;
}

public EmployeeDTO getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeService.getEmployeeById(id);
}

public List<EmployeeDTO> getAllEmployees(@RequestParam(required = false, name = "inputAge") Integer age,
                                           @RequestParam(required = false) String sortBy) {
    return employeeService.getAllEmployees();
}

public EmployeeDTO createNewEmployee(@RequestBody EmployeeDTO inputEmployee) {
    return employeeService.createNewEmployee(inputEmployee);
}
```



```
39     @PostMapping
40     public EmployeeDTO createNewEmployee(@RequestBody EmployeeDTO inputEmployee) {
41         return employeeService.createNewEmployee(inputEmployee);
42     }
43
44     no usages + Anuj Kumar Sharma
45     @PutMapping String updateEmployeeById() { return "Hello from Put"; }
46
47 }
48
```

```
3
4 import org.modelmapper.ModelMapper;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 no usages
9 @Configuration
10 public class MapperConfig {
11
12     no usages
13     @Bean
14     public ModelMapper getModelMapper() {
15         return new ModelMapper();
16     }
17 }
18
```

```
31 --- [Spring Boot Web Tutorial] [           main] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'
31 --- [Spring Boot Web Tutorial] [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000284: Processing PersistenceUnitInfo ...
31 --- [Spring Boot Web Tutorial] [           main] org.hibernate.Version                : HHH000412: Hibernate O...
```

## PUT, PATCH, DELETE MAPPINGS

controller

```
no usages new "
@PutMapping(path = "/{employeeId}")
public EmployeeDTO updateEmployeeById(@RequestBody EmployeeDTO employeeDTO) {
    return employeeService.updateEmployeeById(employeeId);
}
```

```
no usages new "
@PutMapping(path = "/{employeeId}")
public EmployeeDTO updateEmployeeById(@RequestBody EmployeeDTO employeeDTO, @PathVariable Long employeeId) {
    return employeeService.updateEmployeeById(employeeId, employeeDTO);
}
```

```
Service
no usages new "
public EmployeeDTO updateEmployeeById(Long employeeId, EmployeeDTO employeeDTO) {
    EmployeeEntity employeeEntity = modelMapper.map(employeeDTO, EmployeeEntity.class);
    employeeEntity.setId(employeeId);
    EmployeeEntity savedEmployeeEntity = employeeRepository.save(employeeEntity);
    return modelMapper.map(savedEmployeeEntity, EmployeeDTO.class);
}
```

POST http://localhost:8080/employees/1

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

Body (JSON)

```

1 {
2     "name": "Shivam",
3     "email": "Shivam@codingshuttle.com",
4     "age": 20,
5     "dateOfJoining": "2024-01-03",
6     "isActive": false
7 }

```

Response

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

Body (JSON)

```

1 {
2     "id": 1,
3     "name": "Shivam",
4     "email": "Shivam@codingshuttle.com",
5     "age": 20,
6     "dateOfJoining": "2024-01-03",
7     "active": false
8 }

```

Status: 200 OK Time: 84 ms

## Delete Mapping

Controller:

```

no usages new*
@DeleteMapping(path = "/{employeeId}")
public void deleteEmployeeById(@PathVariable Long employeeId) {
    employeeService.updateEmployeeById(employeeId, employeeDTO);
}

```

## Service

```

no usages new*
@DeleteMapping(path = "/{employeeId}")
public void deleteEmployeeById(@PathVariable Long employeeId) {
    employeeService.deleteEmployeeById(employeeId);
}

```

Personal Workspace

New Import POST http://localhost:8080/ PUT http://localhost:8080/e PATCH http://localhost:8080/er DEL http://localhost:8080/er + No environment

http://localhost:8080/employees/54

DELETE http://localhost:8080/employees/54

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Body: none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body.

Body Cookies Headers (4) Test Results

Status: 200 OK Time: 103 ms Size: 123 B Save as example

Text

1

```

    1 usage new
    2 public boolean deleteEmployeeById(Long employeeId) {
    3     boolean exists = employeeRepository.existsById(employeeId);
    4     if(!exists) return false;
    5     employeeRepository.deleteById(employeeId);
    6     return true;
    7 }

```

## PATCH MAPPING

Changing some of the fields of an employee existing

Problems with Put mapping:

POST http://localhost:8080/ PUT http://localhost:8080/e PATCH http://localhost:8080/er DEL http://localhost:8080/er +

http://localhost:8080/employees/1

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

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Body: none form-data x-www-form-urlencoded raw binary JSON

1 {  
2 "name": "XYZ"  
3 }

Body Cookies Headers (5) Test Results

JSON

1 {  
2 "id": 1,  
3 "name": "XYZ",  
4 "email": null,  
5 "age": null,  
6 "dateOfJoining": null,  
7 "active": null  
8 }

You can see every other fields became null here

So

## controller

```
no usages new"
@PatchMapping(path = "/{employeeId}")
public EmployeeDTO updatePartialEmployeeById(@RequestBody Map<String, Object> updates,
                                             @PathVariable Long employeeId) {
    return employeeService.updatePartialEmployeeById(employeeId, updates);
}
```

## Service

```
2 usages new"
public boolean existsByEmployeeId(Long employeeId) {
    return employeeRepository.existsById(employeeId);
}

1 usage new"
public boolean deleteEmployeeById(Long employeeId) {
    boolean exists = existsByEmployeeId(employeeId);
    if(!exists) return false;
    employeeRepository.deleteById(employeeId);
    return true;
}

1 usage new"
public EmployeeDTO updatePartialEmployeeById(Long employeeId, Map<String, Object> updates) {
    boolean exists = existsByEmployeeId(employeeId);
    if(!exists) return null;
    EmployeeEntity employeeEntity = employeeRepository.findById(employeeId).get();
    updates.forEach((field, value) -> {
        Field fieldToUpdate = ReflectionUtils.findRequiredField(EmployeeEntity.class, field);
        fieldToUpdate.setAccessible(true);
    });
}
}
```

```
1 usage new"
public EmployeeDTO updatePartialEmployeeById(Long employeeId, Map<String, Object> updates) {
    boolean exists = existsByEmployeeId(employeeId);
    if(!exists) return null;
    EmployeeEntity employeeEntity = employeeRepository.findById(employeeId).get();
    updates.forEach((field, value) -> {
        Field fieldToUpdate = ReflectionUtils.findRequiredField(EmployeeEntity.class, field);
        fieldToUpdate.setAccessible(true);
        ReflectionUtils.setField(fieldToUpdate, employeeEntity, value);
    });
    return modelMapper.map(employeeRepository.save(employeeEntity), EmployeeDTO.class);
}
}
```

The screenshot shows a Postman interface with the following details:

- Method: PATCH
- URL: <http://localhost:8080/employees/2>
- Body type: JSON
- Body content:

```
1 {
2     "name": "XYZ"
3 }
```
- Response status: Status: 200 OK Time: 265 ms Size: 270 B
- Response content:

```
1 {
2     "id": 2,
3     "name": "XYZ",
4     "email": "Anuj@codingshuttle.com",
5     "age": 23,
6     "dateOfJoining": "2024-01-01",
7     "active": null
8 }
```

isActive null error solution

DTO

```
4
5 import java.time.LocalDate;
6
7 21 usages  ↳ Anuj Kumar Sharma *
8 public class EmployeeDTO {
9
10    3 usages
11    private Long id;
12    3 usages
13    private String name;
14    3 usages
15    private String email;
16    3 usages
17    private Integer age;
18    3 usages
19    private LocalDate dateOfJoining;
20    3 usages
21    @JsonProperty("isActive")
22    private Boolean isActive;
23
24    no usages  ↳ Anuj Kumar Sharma
25    public EmployeeDTO() {
26
27    }
28
29    no usages  ↳ Anuj Kumar Sharma
30    public EmployeeDTO(Long id, String name, String email,
31        this.id = id;
```

## Input Validation Annotations, Creating Custom Annotation for Validation

We need validations so that it can be restricted before user's request reaches the service layers. We have some annotations for that.

Firstly need springboot- starter validation dependency.

## Annotations for Validation

S . No .	Annotation	Description
5	@Max	Ensures that the annotated element is a number with a value no greater than the specified maximum.
6	@Email	Validates that the annotated string is a valid email address.
7	@Pattern	Validates that the annotated string matches the specified regular expression.
8	@Positive	Ensures that the annotated element is a positive number (greater than zero).
9	@PositiveOrZero	Ensures that the annotated element is a positive number or zero.



## EmployeeDTO

```
EmployeeController.java pom.xml (springbootwebtutorial) ResponseEntity.class HttpStatus.class EmployeeDTO.java

1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.dto;
2
3 import com.fasterxml.jackson.annotation.JsonProperty;
4 import jakarta.validation.constraints.NotNull;
5 import lombok.AllArgsConstructor;
6 import lombok.Getter;
7 import lombok.NoArgsConstructor;
8 import lombok.Setter;
9
10 import java.time.LocalDate;
11
12 24 usages
13 @Getter
14 @Setter
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class EmployeeDTO {
18
19     private Long id;
20     @NotNull(message = "Required field in Employee: name")
21     private String name;
22     private String email;
23     private Integer age;
24     private LocalDate dateOfJoining;
25     @JsonProperty("isactive")
26     private Boolean isActive;
27 }
```

## EmployeeController

```
no usages
@PostMapping
public ResponseEntity<EmployeeDTO> createNewEmployee(@RequestBody @Valid EmployeeDTO inputEmployee) {
    EmployeeDTO savedEmployee = employeeService.createNewEmployee(inputEmployee);
    return new ResponseEntity<>(savedEmployee, HttpStatus.CREATED);
}
```

Now lets try sending no name.

The screenshot shows a POST request to `http://127.0.0.1:8080/employees` with the following JSON body:

```
POST http://127.0.0.1:8080/employees
Params Authorization Headers (9) Body Pre-request Script Tests Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
1 ...
2 ...
3 ...
4 ...
5 ...
6 ...
7 ...
```

The response status is 400 Bad Request, with the following JSON body:

```
Pretty Raw Preview Visualize JSON
1 ...
2 ...
3 ...
4 ...
5 ...
6 ...
```

```

import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

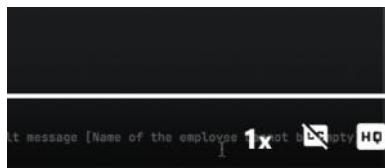
import java.time.LocalDate;

24 usages
@NotNull
@NotNull
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class EmployeeDTO {

    private Long id;
    @NotEmpty(message = "Name of the employee cannot be empty")
    private String name;
    private String email;
    private Integer age;
    private LocalDate dateOfJoining;
    @JsonProperty("isActive")
    private Boolean isActive;
}

```

+919874780718



```

EmployeeController.java  pom.xml  SpringBootTutorials  ResponseEntity.class  HttpStatus.class  EmployeeDTO.java  NotNull.class  EmployeeControllerTest.java
10 import java.time.LocalDate;
11
12 24 usages
13 @Getter
14 @Setter
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class EmployeeDTO {
18
19     private Long id;
20
21     @NotBlank(message = "Name of the employee cannot be blank")
22     @Size(min = 3, max = 10, message = "Number of characters in name should be in the range: [3, 10]")
23     private String name;
24
25     @Email(message = "Email should be a valid email")
26     private String email;
27
28     @Max(value = 80, message = "Age of Employee cannot be greater than 80")
29     @Min(value = 18, message = "Age cannot be less than 18")
30     private Integer age;
31     private LocalDate dateOfJoining;
32     @JsonProperty("isActive")
33     private Boolean isActive;
34 }

```

```

EmployeeController.java  pom.xml  SpringBootTutorials  ResponseEntity.class  HttpStatus.class  EmployeeDTO.java  NotNull.class  EmployeeControllerTest.java
10 import java.time.LocalDate;
11
12 24 usages
13 @Getter
14 @Setter
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class EmployeeDTO {
18
19     private Long id;
20
21     @NotBlank(message = "Name of the employee cannot be blank")
22     @Size(min = 3, max = 10, message = "Number of characters in name should be in the range: [3, 10]")
23     private String name;
24
25     @Email(message = "Email should be a valid email")
26     private String email;
27
28     @Max(value = 80, message = "Age of Employee cannot be greater than 80")
29     @Min(value = 18, message = "Age cannot be less than 18")
30     private Integer age;
31     private LocalDate dateOfJoining;
32     @JsonProperty("isActive")
33     private Boolean isActive;
34 }

```

```
10 import java.time.LocalDate;
11
12     24 usages
13     @Getter
14     @Setter
15     @AllArgsConstructor
16     @NoArgsConstructor
17     public class EmployeeDTO {
18
19         private Long id;
20
21         @NotBlank(message = "Name of the employee cannot be blank")
22         @Size(min = 3, max = 10, message = "Number of characters in name should be in the range: [3, 10]")
23         private String name;
24
25         @Email(message = "Email should be a valid email")
26         private String email;
27
28         @Max(value = 80, message = "Age of Employee cannot be greater than 80")
29         @Min(value = 18, message = "Age of Employee cannot be less than 18")
30         private Integer age;
31
32         @Pattern(regexp = "(ADMIN|USER)$")
33         private String role; //ADMIN, USER
34
35         private LocalDate dateOfJoining;
36         @JsonProperty("isActive")
37         private Boolean isActive;
38     }
39
```

```
EmployeeController.java    /** pointcut (springboot/webtutorial)    ResponseEntity.class    HttpStatus.class    EmployeeDTO.java    NotNull.class    Emplo...
10 import java.time.LocalDate;
11
12     24 usages
13     @Getter
14     @Setter
15     @AllArgsConstructor
16     @NoArgsConstructor
17     public class EmployeeDTO {
18
19         private Long id;
20
21         @NotBlank(message = "Name of the employee cannot be blank")
22         @Size(min = 3, max = 10, message = "Number of characters in name should be in the range: [3, 10]")
23         private String name;
24
25         @Email(message = "Email should be a valid email")
26         private String email;
27
28         @Max(value = 80, message = "Age of Employee cannot be greater than 80")
29         @Min(value = 18, message = "Age of Employee cannot be less than 18")
30         private Integer age;
31
32         @Pattern(regexp = "(ADMIN|USER)$", message = "Role of Employee can be USER or ADMIN")
33         private String role; //ADMIN, USER
34
35         private LocalDate dateOfJoining;
36         @JsonProperty("isActive")
37         private Boolean isActive;
38     }
39
```

```

11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
    @Setter
    @NoArgsConstructor
    @AllArgsConstructor
    public class EmployeeDTO {
        private Long id;
        @NotBlank(message = "Name of the employee cannot be blank")
        @Size(min = 3, max = 10, message = "Number of characters in name should be in the range: [3, 10]")
        private String name;
        @NotBlank(message = "Email of the employee cannot be blank")
        @Email(message = "Email should be a valid email")
        private String email;
        @NotNull(message = "Age of the employee cannot be blank")
        @Max(value = 80, message = "Age of Employee cannot be greater than 80")
        @Min(value = 18, message = "Age of Employee cannot be less than 18")
        private Integer age;
        @NotBlank(message = "Role of the employee cannot be blank")
        @Pattern(regexp = "^(ADMIN|USER)$", message = "Role of Employee can be USER or ADMIN")
        private String role; //ADMIN, USER
        private LocalDate dateOfJoining;
        @JsonProperty("isActive")
        private Boolean isActive;
    }

```

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** http://127.0.0.1:8080/employees
- Body:** Raw JSON (selected)
- JSON Payload:**

```

1
2   "name": "Arul",
3   "age": 20,
4   "dateOfJoining": "1996-08-27",
5   "role": "USERS",
6   "isActive": true,
7   "email": "asldikj@glm.com"
8

```
- Response:**
  - Status: 400 Bad Request
  - Time: 172 ms
  - Size: 244 B
  - Content:

```

1
2   "timestamp": "2024-06-15T13:57:05.482+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "path": "/employees"
6

```

## Annotations for Validation

S . No .	Annotation	Description
15	@FutureOrPresent	Ensures that the annotated date or calendar value is in the future or present.
16	@Digits	Ensures that the annotated number has up to a specified number of integer and fraction digits.
17	@DecimalMin	Ensures that the annotated element is a number with a value no less than the specified minimum, allowing

17	<b>@DecimalMin</b>	Ensures that the annotated element is a number with a value no less than the specified minimum, allowing for decimal points.	
18	<b>@DecimalMax</b>	Ensures that the annotated element is a number with a value no greater than the specified maximum, allowing for decimal points.	

```

status.class  EmployeeDTO.java  NotNull.class  EmployeeService.java  CrudRepository.class  SimpleJpaRepository.class  EmployeeEntity

26     @NotNull(message = "Age of the employee cannot be blank")
27     @Max(value = 80, message = "Age of Employee cannot be greater than 80")
28     @Min(value = 18, message = "Age of Employee cannot be less than 18")
29     private Integer age;
30
31
32     @NotBlank(message = "Role of the employee cannot be blank")
33     @Pattern(regexp = "[ADMIN|USER]", message = "Role of Employee can either be USER or ADMIN")
34     private String role; //ADMIN, USER
35
36
37     @NotNull(message = "Salary of Employee should be not null")
38     @Positive(message = "Salary of Employee should be positive")
39     @Digits(integer = 6, fraction = 2, message = "The salary can be in the form XXXXX.YY")
40     @DecimalMax(value = "100000.99")
41     @DecimalMin(value = "100.50")
42     private Double salary;
43
44
45     @PastOrPresent(message = "DateOfJoining field in Employee cannot be in the future")
46     private LocalDate dateOfJoining;
47
48
49     @JsonProperty("isActive")
50     private Boolean isActive;
51
52
53
54
55

```

## Annotations for Validation

S . No .	Annotation	Description
19	<b>@AssertTrue</b>	Ensures that the annotated boolean field is true.
20	<b>@AssertFalse</b>	Ensures that the annotated boolean field is false.
21	<b>@Valid</b>	Validates the associated object recursively (applies bean validation to nested objects).

EmployeeRoleValidator:

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.annotations;
2
3 import jakarta.validation.ConstraintValidator;
4 import jakarta.validation.ConstraintValidatorContext;
5
6 import java.util.List;
7
8 no usages
9 public class EmployeeRoleValidator implements ConstraintValidator<EmployeeRoleValidation, String> {
10
11     @Override
12     public boolean isValid(String inputRole, ConstraintValidatorContext constraintValidatorContext) {
13         List<String> roles = List.of("USER", "ADMIN");
14         return roles.contains(inputRole);
15     }
16 }
```

The screenshot shows a Java code editor with a file named `EmployeeRoleValidation.java` open. The code defines a custom validation annotation `EmployeeRoleValidation` that checks if a string value is either "USER" or "ADMIN". The code editor interface includes a project tree on the left and various tabs at the top.

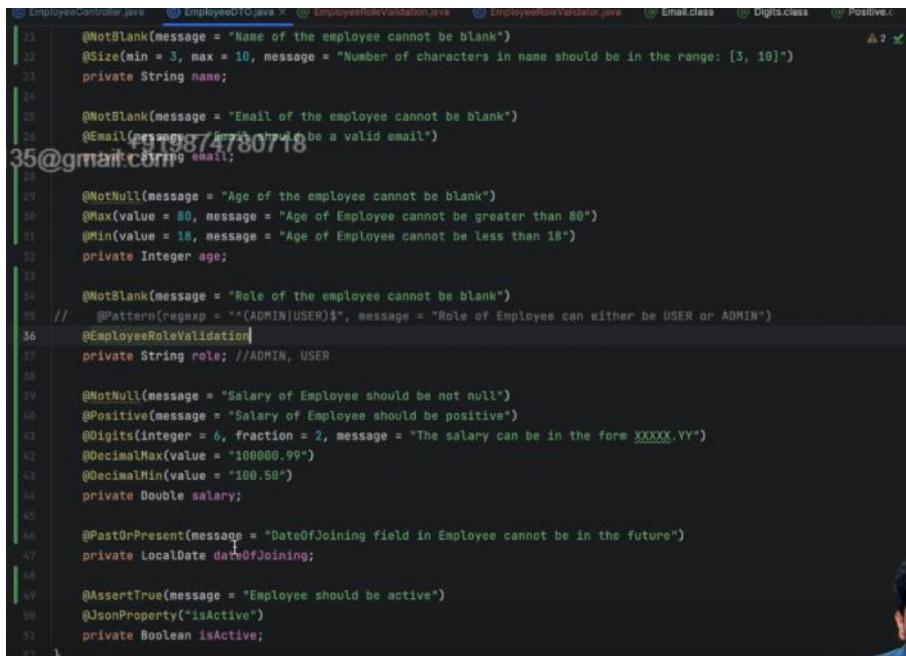
### EmployeeRoleValidation:

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.annotations;
2
3 import jakarta.validation.Payload;
4
5 import java.lang.annotation.ElementType;
6 import java.lang.annotation.Retention;
7 import java.lang.annotation.RetentionPolicy;
8 import java.lang.annotation.Target;
9
10 no usages
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.FIELD, ElementType.PARAMETER})
13 public @interface EmployeeRoleValidation {
14     no usages
15     String message() default "Role of Employee can either be USER or ADMIN";
16     no usages
17     Class<?>[] groups() default {};
18     no usages
19     Class<? extends Payload>[] payload() default {};
20 }
```

This screenshot shows the implementation of the `EmployeeRoleValidation` annotation. It is annotated with `@Retention(RetentionPolicy.RUNTIME)` and `@Target({ElementType.FIELD, ElementType.PARAMETER})`. The annotation has three methods: `message()`, `groups()`, and `payload()`, each with its default value specified.

These are custom annotations - try creating more and more annotations for practice.

Below is our custom validation in practice:



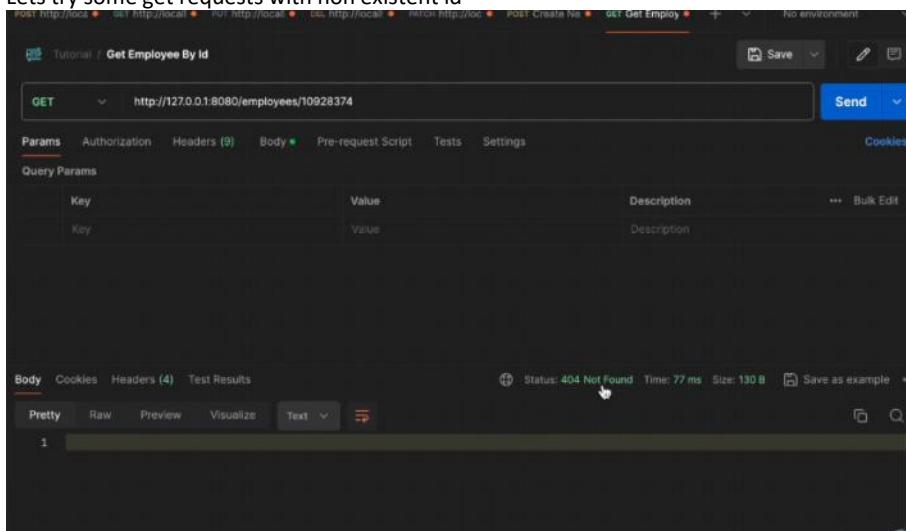
```

21     @NotBlank(message = "Name of the employee cannot be blank")
22     @Size(min = 3, max = 10, message = "Number of characters in name should be in the range: [3, 10]")
23     private String name;
24
25     @NotBlank(message = "Email of the employee cannot be blank")
26     @Email(message = "Email address must be a valid email")
27     private String email;
28
29     @NotNull(message = "Age of the employee cannot be blank")
30     @Max(value = 80, message = "Age of Employee cannot be greater than 80")
31     @Min(value = 18, message = "Age of Employee cannot be less than 18")
32     private Integer age;
33
34     @NotBlank(message = "Role of the employee cannot be blank")
35     // @Pattern(regexp = "(ADMIN|USER)", message = "Role of Employee can either be USER or ADMIN")
36     @EmployeeRoleValidation
37     private String role; //ADMIN, USER
38
39     @NotNull(message = "Salary of Employee should be not null")
40     @Positive(message = "Salary of Employee should be positive")
41     @Digits(integer = 6, fraction = 2, message = "The salary can be in the form XXXXX.YY")
42     @DecimalMax(value = "100000.99")
43     @DecimalMin(value = "100.50")
44     private Double salary;
45
46     @PastOrPresent(message = "DateOfJoining field in Employee cannot be in the future")
47     private LocalDate dateOfJoining;
48
49     @AssertTrue(message = "Employee should be active")
50     @JsonProperty("isActive")
51     private Boolean isActive;

```

## Exception Handling in Springboot MVC

Lets try some get requests with non existent id



The screenshot shows a Postman collection named 'Tutorial' with a single request titled 'Get Employee By Id'. The request is a GET to `http://127.0.0.1:8080/employees/10928374`. The response status is 404 Not Found, with a time of 77 ms and a size of 130 B. The response body is empty.

## Handling Exceptions

# Handling Exceptions

- Use `@ExceptionHandler` to handle specific exceptions in controllers.
- Use `@RestControllerAdvice` for global exception handling.
- Return appropriate HTTP status codes and error messages.
- Use Custom error response class to provide structured error details.

## EmployeeController

```
no usages
@ExceptionHandler(NoSuchElementException.class)
public String handleEmployeeNotFound(NoSuchElementException exception) {
    return "Employee was not found";
}

no usages
@GetMapping(path = "/{employeeId}")
public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);
    return employeeDTO
        .map(employeeDTO1 -> ResponseEntity.ok(employeeDTO1))
        .orElseThrow(() -> new NoSuchElementException("Employee not found"));
}

no usages
@ExceptionHandler(NoSuchElementException.class)
public String handleEmployeeNotFound(NoSuchElementException exception) {
    return "Employee was not found";
}
```

## In your code:

≡ ⌂ ⌂ ⌂ </> Java

```
1 Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);  
2
```

Here:

- `employeeService.getEmployeeById(id)` returns an `Optional<EmployeeDTO>`.
- This means the service might return an employee or might return nothing (empty `Optional`).

## Then you do:

≡ ⌂ ⌂ ⌂

```
1 return employeeDTO  
2     .map(ResponseEntity::ok)  
3     .orElseThrow(() -> new NoSuchElementException("Employee not found"));
```

### What happens here?

- **map(ResponseEntity::ok)**  
If `employeeDTO` has a value (employee found), map converts it into `ResponseEntity.ok(employeeDTO)` and returns it.
- **orElseThrow(...)**  
If `employeeDTO` is empty (employee not found), this part executes and throws `NoSuchElementException`.

So:

- **Employee exists → returns 200 OK with the employee data.**
- **Employee does NOT exist → throws exception → Spring calls your @ExceptionHandler.**

From your snippet, there are two different "paths" depending on whether the Optional has a value:

```
1  @GetMapping(path = "/{employeeId}")
2  public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long
   id) {
3      Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);
4
5      return employeeDTO
6          .map(ResponseEntity::ok)
7              .orElseThrow(() -> new NoSuchElementException("Employee not found")); // <-- returns 200 OK
8      with DTO
9
10     @ExceptionHandler(NoSuchElementException.class)
11     public String handleEmployeeNotFound(NoSuchElementException exception) {
12         return "Employee was not found";
13     }
```

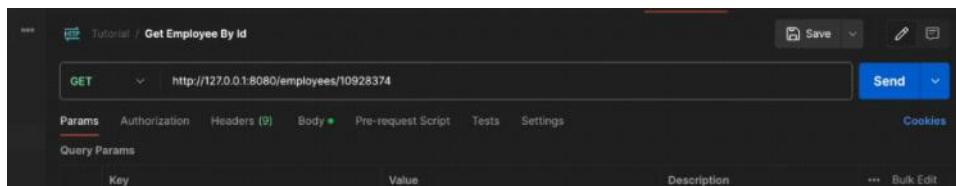
```
no usages
12  @GetMapping(path = "/{employeeId}")
13  public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long id) {
14      Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);
15      return employeeDTO
16          .map(employeeDTO1 -> ResponseEntity.ok(employeeDTO1))
17          .orElseThrow(() -> new NoSuchElementException("Employee not found"));
18  }

no usages
40  |@ExceptionHandler(NoSuchElementException.class)
41  public String handleEmployeeNotFound(NoSuchElementException exception) {
42      return "Employee was not found";
43  }
```

The above was giving http code 200, which we didn't want.

```
no usages
12  @GetMapping(path = "/{employeeId}")
13  public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long id) {
14      Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);
15      return employeeDTO
16          .map(employeeDTO1 -> ResponseEntity.ok(employeeDTO1))
17          .orElseThrow(() -> new NoSuchElementException("Employee not found"));
18  }

no usages
19  |@ExceptionHandler(NoSuchElementException.class)
20  public ResponseEntity<String> handleEmployeeNotFound(NoSuchElementException exception) {
21      return new ResponseEntity<>{ body: "Employee not found", HttpStatus.NOT_FOUND};
22  }
```



The screenshot shows a POSTMAN interface. At the top, it says "Tutorial / Get Employee By Id". Below that, there's a "GET" dropdown and a URL input field containing "http://127.0.0.1:8080/employees/10928374". To the right of the URL is a "Send" button. Below the URL, there are tabs for "Params", "Authorization", "Headers (9)", "Body", "Pre-request Script", "Tests", and "Settings". Under "Headers (9)", there is a table with columns "Key", "Value", and "Description". The "Body" tab is selected, showing a table with columns "Key", "Value", and "Description". At the bottom, there are tabs for "Body", "Cookies", "Headers (5)", and "Test Results". The "Test Results" tab is selected, showing a status message: "Status: 404 Not Found Time: 119 ms Size: 189 B". Below this, there are buttons for "Pretty", "Raw", "Preview", "Visualize", and "Text". The "Text" button is highlighted with a red underline. The response body is displayed as "1 Employee not found".

## Handling exceptions globally

### Create package - advices

The screenshot shows an IDE with several files listed in the top bar: EmployeeController.java, GlobalExceptionHandler.java (which is currently selected), EmployeeDTO.java, EmployeeRoleValidation.java, and EmployeeRoleValidator.java. The GlobalExceptionHandler.java file contains the following code:

```

1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advices;
2
3 import org.springframework.web.bind.annotation.RestControllerAdvice;
4
5 no usages
6 @RestControllerAdvice
7 public class GlobalExceptionHandler {
8 }

```

We have rather used handleEmployeeNotFoundException() here in this GlobalExceptionHandler class.

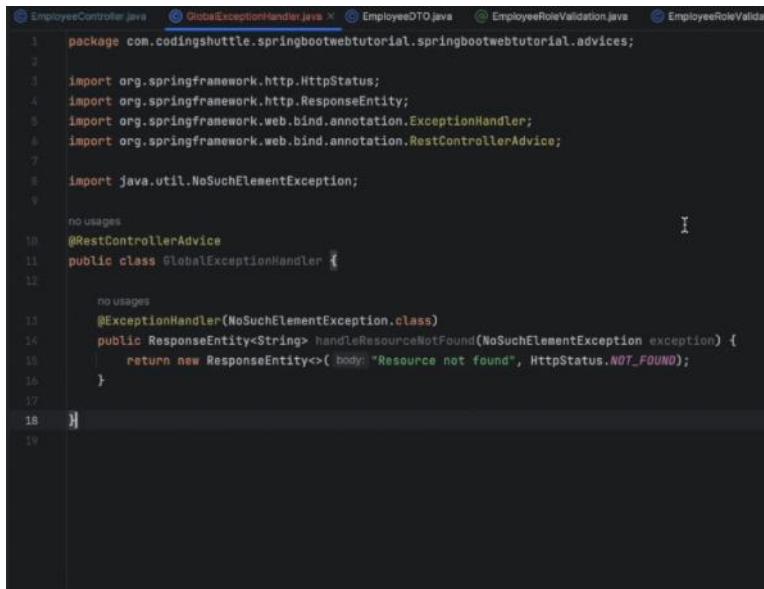
The screenshot shows the GlobalExceptionHandler.java file again. The handleEmployeeNotFoundException() method is highlighted with a red box. The code is as follows:

```

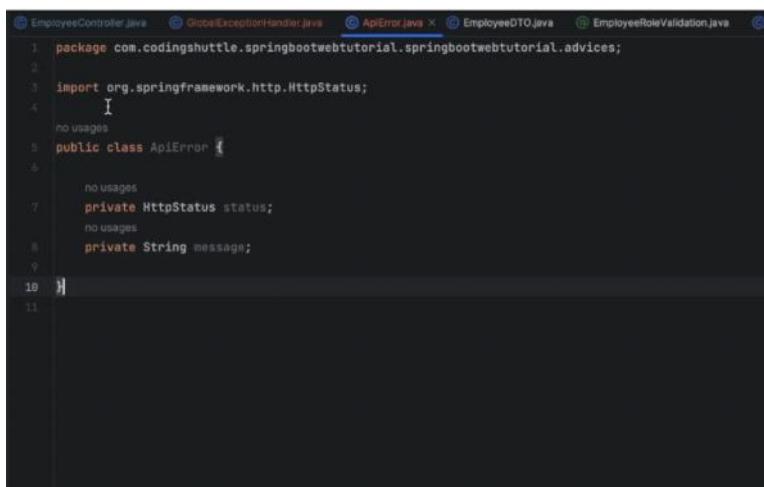
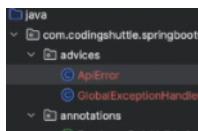
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advices;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.ExceptionHandler;
6 import org.springframework.web.bind.annotation.RestControllerAdvice;
7
8 import java.util.NoSuchElementException;
9
10 no usages
11 @RestControllerAdvice
12 public class GlobalExceptionHandler {
13
14     @ExceptionHandler(NoSuchElementException.class)
15     public ResponseEntity<String> handleEmployeeNotFound(NoSuchElementException exception) {
16         return new ResponseEntity<String>("Employee not found", HttpStatus.NOT_FOUND);
17     }
18 }

```

Now it's a global resourcenotfoundhandler.

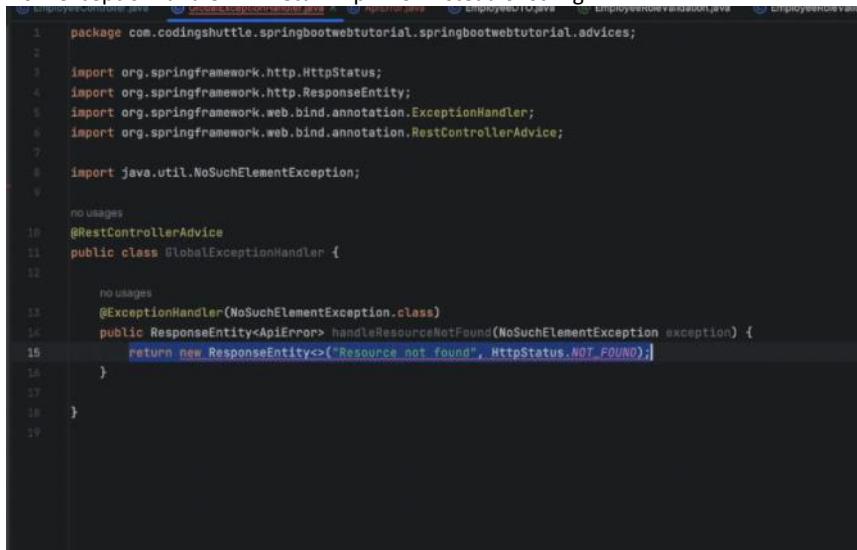


```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advices;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.ExceptionHandler;
6 import org.springframework.web.bind.annotation.RestControllerAdvice;
7
8 import java.util.NoSuchElementException;
9
10 no usages
11
12 @RestControllerAdvice
13 public class GlobalExceptionHandler {
14
15     no usages
16     @ExceptionHandler(NoSuchElementException.class)
17     public ResponseEntity<String> handleResourceNotFound(NoSuchElementException exception) {
18         return new ResponseEntity<>("Resource not found", HttpStatus.NOT_FOUND);
19     }
20
21 }
```

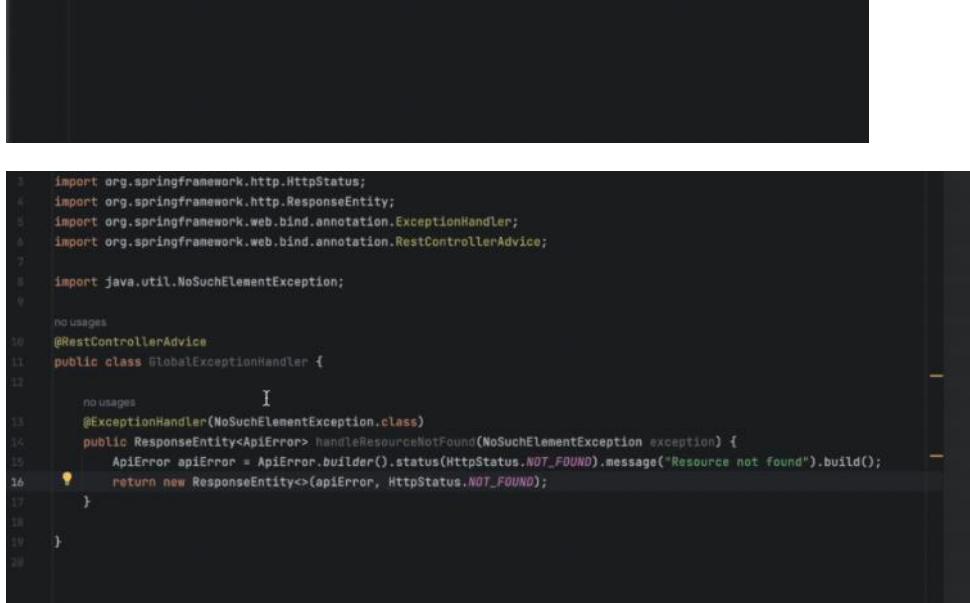


```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advices;
2
3 import org.springframework.http.HttpStatus;
4
5 no usages
6
7 public class ApiError {
8
9     no usages
10    private HttpStatus status;
11    no usages
12    private String message;
13
14 }
```

Now exceptionhandler will return ApiError instead of string



```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advices;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.ExceptionHandler;
6 import org.springframework.web.bind.annotation.RestControllerAdvice;
7
8 import java.util.NoSuchElementException;
9
10 no usages
11
12 @RestControllerAdvice
13 public class GlobalExceptionHandler {
14
15     no usages
16     @ExceptionHandler(NoSuchElementException.class)
17     public ResponseEntity<ApiError> handleResourceNotFound(NoSuchElementException exception) {
18         return new ResponseEntity<>("Resource not found", HttpStatus.NOT_FOUND);
19     }
20
21 }
```



```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.NoSuchElementException;

no usages

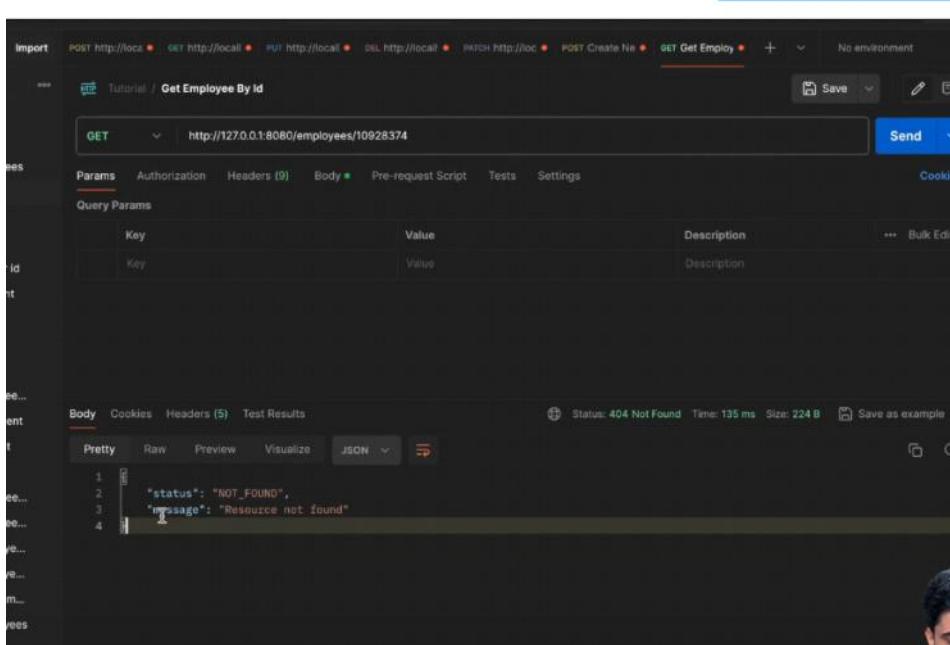
@RestControllerAdvice
public class GlobalExceptionHandler {

    no usages
    @ExceptionHandler(NoSuchElementException.class)
    public ResponseEntity<ApiError> handleResourceNotFound(NoSuchElementException exception) {
        ApiError apiError = ApiError.builder().status(HttpStatus.NOT_FOUND).message("Resource not found").build();
        return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
    }

}

```

Now users will have a detailed clue about what is happening

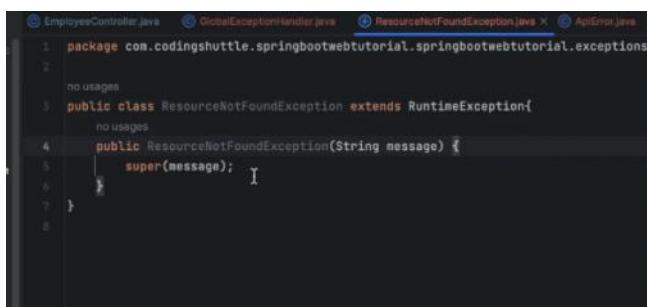


The screenshot shows a Postman collection named "Tutorial / Get Employee By Id". A GET request is made to `http://127.0.0.1:8080/employees/10928374`. The response status is 404 Not Found, with a message body containing:

```
1:     "status": "NOT_FOUND",
2:     "message": "Resource not found"
```

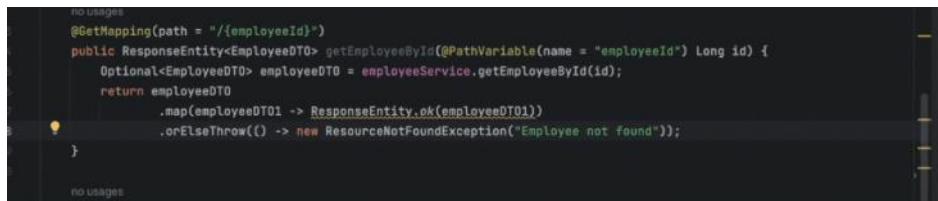
Now lets create another package and name it "Exception" where we will create our own exceptions

We will create ResourceNotFoundException exception.



```
EmployeeController.java  GlobalExceptionHandler.java  ResourceNotFoundException.java  ApiError.java
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.exceptions;
2
3 no usages
4 public class ResourceNotFoundException extends RuntimeException{
5     no usages
6     public ResourceNotFoundException(String message) {
7         super(message);
8     }
9 }
```

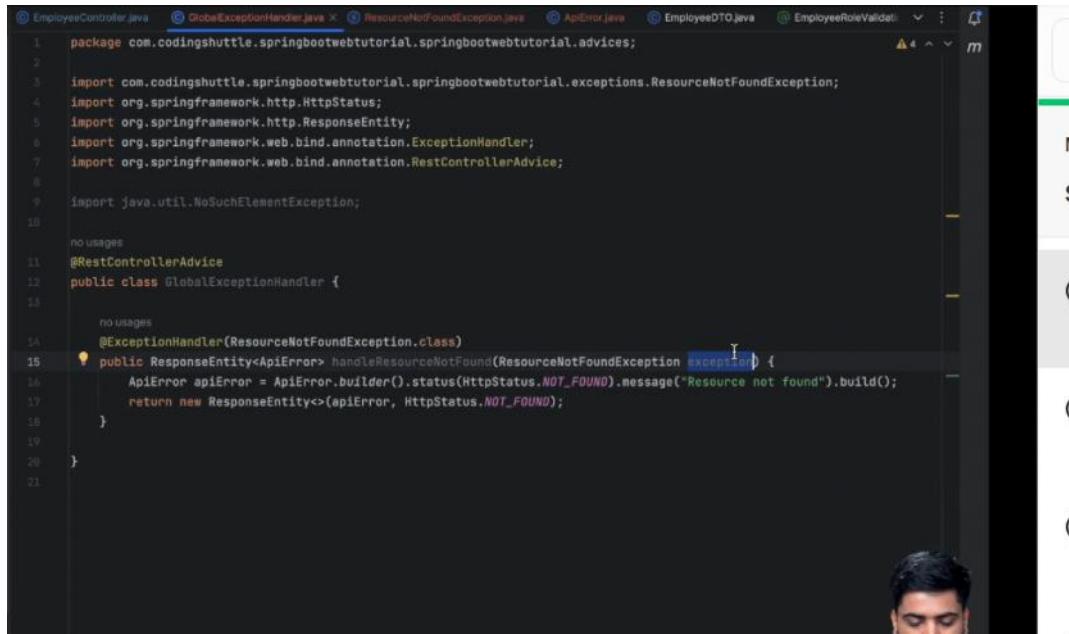
Now in controller,



```
no usages
@GetMapping(path = "/{employeeId}")
public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);
    return employeeDTO
        .map(employeeDTO1 -> ResponseEntity.ok(employeeDTO1))
        .orElseThrow(() -> new ResourceNotFoundException("Employee not found"));
}
```

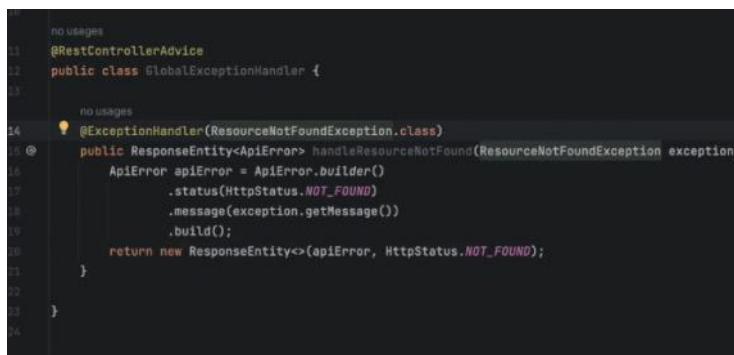
The code shows a `@GetMapping` annotation with a path parameter `{employeeId}`. It calls `getEmployeeById` from `employeeService`, maps the result to a `ResponseEntity`, and handles the absence of a resource with a `ResourceNotFoundException`.

## Global ExceptionHandler



```
EmployeeController.java GlobalExceptionHandler.java ResourceNotFoundException.java ApiError.java EmployeeDTO.java EmployeeRoleValidator.java
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advices;
2
3 import com.codingshuttle.springbootwebtutorial.springbootwebtutorial.exceptions.ResourceNotFoundException;
4 import org.springframework.http.HttpStatus;
5 import org.springframework.http.ResponseEntity;
6 import org.springframework.web.bind.annotation.ExceptionHandler;
7 import org.springframework.web.bind.annotation.RestControllerAdvice;
8
9 import java.util.NoSuchElementException;
10
11 no usages
12 @RestControllerAdvice
13 public class GlobalExceptionHandler {
14
15     no usages
16     @ExceptionHandler(ResourceNotFoundException.class)
17     public ResponseEntity<ApiError> handleResourceNotFound(ResourceNotFoundException exception) {
18         ApiError apiError = ApiError.builder().status(HttpStatus.NOT_FOUND).message("Resource not found").build();
19         return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
20     }
21 }
```

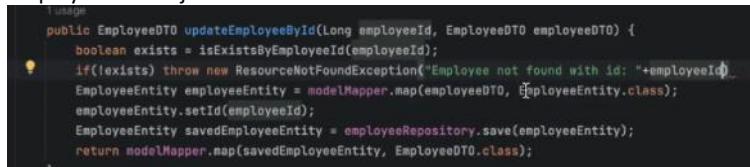
The code defines a `GlobalExceptionHandler` class annotated with `@RestControllerAdvice`. It contains a single method `handleResourceNotFound` that takes a `ResourceNotFoundException` as an argument and returns a `ResponseEntity` with an `ApiError` object and a `NOT_FOUND` status.



```
no usages
@RestControllerAdvice
public class GlobalExceptionHandler {
    no usages
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiError> handleResourceNotFound(ResourceNotFoundException exception) {
        ApiError apiError = ApiError.builder()
            .status(HttpStatus.NOT_FOUND)
            .message(exception.getMessage())
            .build();
        return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
    }
}
```

This screenshot shows the same `GlobalExceptionHandler` class with a slightly different code structure, where the `handleResourceNotFound` method uses the `builder()` method of the `ApiError` class to build the response entity.

## EmployeeService.java



```
no usages
public EmployeeDTO updateEmployeeById(Long employeeId, EmployeeDTO employeeDTO) {
    boolean exists = isExistsByEmployeeId(employeeId);
    if(!exists) throw new ResourceNotFoundException("Employee not found with id: "+employeeId);
    EmployeeEntity employeeEntity = modelMapper.map(employeeDTO, EmployeeEntity.class);
    employeeEntity.setId(employeeId);
    EmployeeEntity savedEmployeeEntity = employeeRepository.save(employeeEntity);
    return modelMapper.map(savedEmployeeEntity, EmployeeDTO.class);
}
```

The code shows a `updateEmployeeById` method that checks if an employee with the given ID exists. If not, it throws a `ResourceNotFoundException`. Otherwise, it maps the `EmployeeDTO` to an `EmployeeEntity`, updates the ID, saves it to the database, and then maps it back to a `EmployeeDTO`.

So we can throw exceptions from our services as well

PUT http://127.0.0.1:8080/employees/3

Body

```

1 {
2   "name": "Anuj",
3   "age": 20,
4   "dateOfJoining": "2024-06-14",
5   "role": "ADMIN",
6   "isActive": true,
7   "email": "anujfjha@gmail.com",
8   "salary": 100.60
9 }

```

Status: 404 Not Found

Pretty Raw Preview Visualize JSON

```

1 {
2   "status": "NOT_FOUND",
3   "message": "Employee not found with id: 3"
4 }

```

Another exception apart from resourcenotfoundexception

```

no usages
@ExceptionHandler(Exception.class)
public ResponseEntity<ApiError> handleInternalServerError(Exception exception) {
    ApiError apiError = ApiError.builder()
        .status(HttpStatus.INTERNAL_SERVER_ERROR)
        .message(exception.getMessage())
        .build();
    return new ResponseEntity<>(apiError, HttpStatus.INTERNAL_SERVER_ERROR);
}

```

PUT http://127.0.0.1:8080/employees/1005

Body

```

1 {
2   "name": "Shivam",
3   "age": 20,
4   "dateOfJoining": "2024-06-14",
5   "role": "ADMIN",
6   "isActive": true,
7   "email": "asidik@a@gmail.com",
8   "salary": 100.60
9 }

```

Status: 500 Internal Server Error Time: 105 ms Size: 1004 B Save as example

Pretty Raw Preview Visualize JSON

```

1 {
2   "status": "INTERNAL_SERVER_ERROR",
3   "message": "Validation failed for argument [0] in public org.springframework.http.ResponseEntity<com.codingshuttle.springbootwebtutorial.dto.EmployeeDTO> com.codingshuttle.springbootwebtutorial.springbootwebtutorial.controllers.EmployeeController.updateEmployeeById(com.codingshuttle.springbootwebtutorial.springbootwebtutorial.dto.EmployeeDTO,java.lang.Long); (Field error object 'employeeDTO' on field 'role': rejected value [ADMIN]; codes [EmployeeRoleValidation.employeeDTO.role.EmployeeRoleValidation.role.EmployeeRoleValidation.java.lang.String.EmployeeRoleValidation]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [EmployeeDTO.role.role]; arguments []]; default message [role]]; default message [Role of Employee can either be USER or ADMIN]\""
4 }

```

Here, the message is not useful to end user.

```

no usages
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ApiError> handleInputValidationErrors(MethodArgumentNotValidException exception) {
    List<String> errors = exception
        .getBindingResult()
        .getAllErrors()
        .stream()
        .map(error -> error.getDefaultMessage())
        .collect(Collectors.toList());
    ApiError apiError = ApiError.builder()
        .status(HttpStatus.BAD_REQUEST)
        .message("Input validation failed")
        .subErrors(errors)
        .build();
}

```

```
no usages
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ApiError> handleInputValidationErrors(MethodArgumentNotValidException exception) {
    List<String> errors = exception
        .getBindingResult()
        .getAllErrors()
        .stream()
        .map(error -> error.getDefaultMessage())
        .collect(Collectors.toList());

    ApiError apiError = ApiError.builder()
        .status(HttpStatus.BAD_REQUEST)
        .message("Input validation failed")
        .subErrors(errors)
        .build();
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}
```



A screenshot of a video player interface. The video frame shows a man with a beard and dark hair, wearing a blue shirt, looking down at something off-camera. The video player has a dark theme with white text. At the top, it says "Tutorial / http://127.0.0.1:8080/employees/1". Below that is a "Send" button. The main area is a code editor for a PUT request to "http://127.0.0.1:8080/employees/1005". The "Body" tab is selected, showing JSON input:

```
1 {
2     "name": "X",
3     "age": 2,
4     "dateOfJoining": "2024-06-14",
5     "role": "ADMIN",
6     "isActive": true,
7     "email": "asldfkja@gmail.com",
8     "salary": 100.60
9 }
```

Below the code editor, the "Test Results" section shows the response:

```
1 {
2     "status": "BAD_REQUEST",
3     "message": "Input validation failed",
4     "subErrors": [
5         "Number of characters in name should be in the range: [3, 10]",
6         "Role of Employee can either be USER or ADMIN",
7         "Age of Employee cannot be less than 18"
8     ]
9 }
```

The status bar at the bottom of the video player says "Status: 400 Bad Request Time: 138 ms Size: 369 B".

Now its good.

Transforming API responses in Spring web MVC

## Transforming API Response

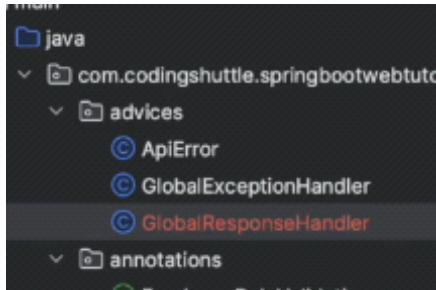
# Transforming API Response

- Extend your class with `@ResponseBodyAdvice<Object>` to define the custom return type.
- Use `@RestControllerAdvice` for global API Response transformation.
- Return appropriate HTTP status codes and error messages.
- You can also return the timestamp of the API response.

## Transforming API Response

# Transforming API Response

- Extend your class with `@ResponseBodyAdvice<Object>` to define the custom return type.
- Use `@RestControllerAdvice` for global API Response transformation.
- Return appropriate HTTP status codes and error messages.
- You can also return the timestamp of the API response.



GlobalResponseHandler

```
no usages.
11 @RestControllerAdvice
12 public class GlobalResponseHandler implements ResponseBodyAdvice<Object> {
13
14     @Override
15     public boolean supports(MethodParameter returnType, Class<? extends HttpMessageConverter<?>> converterType) {
16         return true;
17     }
18
19     no usages.
20     @Override
21     public Object beforeBodyWrite(Object body, MethodParameter returnType, MediaType selectedContentType, Class<?
22         return null;
23     }
24 }
```

A screenshot of a code editor showing the 'GlobalResponseHandler.java' file. The code defines a class that implements the `ResponseBodyAdvice<Object>` interface. It overrides the `supports` method to always return `true`. It also overrides the `beforeBodyWrite` method, which returns `null`. The code editor shows syntax highlighting and line numbers from 11 to 24.

Create another class ApiResponse

## Extra Example: Generic Class vs Object-based Class

## Object-based Box (Old Style)

```
Java
class ObjectBox {
    private Object value;
    public ObjectBox(Object value) { this.value = value; }
    public Object get() { return value; }
}

public class Demo {
    public static void main(String[] args) {
        ObjectBox box = new ObjectBox("Hello");
        String s = (String) box.get(); // Must cast
        System.out.println(s.toUpperCase());
    }
}
Show more lines
```

## Generic Box (Modern Style)

```
Java
class Box<T> {
    private T value;
    public Box(T value) { this.value = value; }
    public T get() { return value; }
}

public class Demo {
    public static void main(String[] args) {
        Box<String> box = new Box<>("Hello");
        String s = box.get(); // No cast needed
        System.out.println(s.toUpperCase());
    }
}
```

Show more lines

**Summary:**

Generics were introduced because:

- Object-based collections lacked type safety.
- Required manual casting.
- Errors appeared at runtime instead of compile time.

Generics fixed this by:

- Enforcing type safety at compile time.
- Eliminating casts.
- Making code reusable and cleaner.

## Lambdas and Streams

```
30 MainClass.java ×
...
1 package streamAPI;
2
3 //Lambda, Stream
4
5 public class MainClass {
6     public static void main(String[] args) {
7
8
9
10    }
11 }
12
```

```
30 MainClass.java ×
...
1 package streamAPI;
2
3 //Lambda, Stream
4
5 public class MainClass {
6     public static void main(String[] args) {
7
8
9
10    }
11 }
12
13 interface Walkable { 1 usage 1 implementation
14     int walk(int steps); no usages
15 }
16
17
18 class WalkFast implements Walkable{ no usages
19
20     @Override no usages
21     public int walk(int steps) {
22         return 0;
23     }
24 }
```

Defining my own anonymous class

```
package streamAPI;

//Lambda, Stream

public class MainClass {
    public static void main(String[] args) {

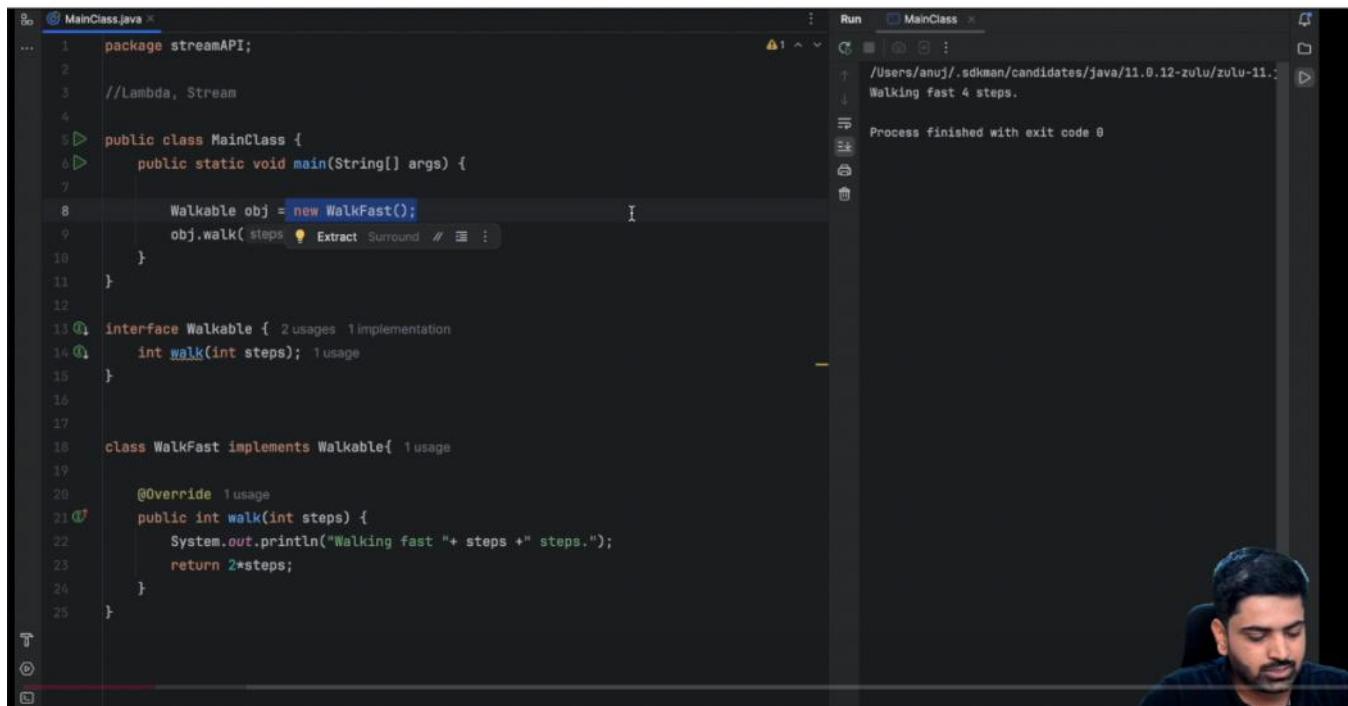
        Walkable obj = new Walkable() {
            @Override no usages
            public int walk(int steps) {
                return 0;
            }
        };
    }
}

interface Walkable { 3 usages 1 implementation
    int walk(int steps); no usages
}

class WalkFast implements Walkable{ no usages

    @Override no usages
    public int walk(int steps) {
        System.out.println("Walking fast "+ steps +" steps.");
        return 2*steps;
    }
}
```

Now take a look



The screenshot shows an IDE interface with two panes. The left pane displays the `MainClass.java` file. The right pane shows the `Run` tab with the output of the program execution.

```
MainClass.java
...
1 package streamAPI;
2
3 //Lambda, Stream
4
5 public class MainClass {
6     public static void main(String[] args) {
7
8         Walkable obj = new WalkFast();
9         obj.walk(4);
10    }
11 }
12
13 interface Walkable { 2 usages 1 implementation
14     int walk(int steps); 1 usage
15 }
16
17
18 class WalkFast implements Walkable{ 1 usage
19
20     @Override 1 usage
21     public int walk(int steps) {
22         System.out.println("Walking fast "+ steps +" steps.");
23         return 2*steps;
24     }
25 }
```

Run tab output:

```
/Users/anuj/.sdkman/candidates/java/11.0.12-zulu/zulu-11: MainClass
Walking fast 4 steps.

Process finished with exit code 0
```

Above implementation is not the only way.

Above is a tedious way to create an instance of an interface.

If an interface have 1 method, then we don't have to define class or anonymous class etc.

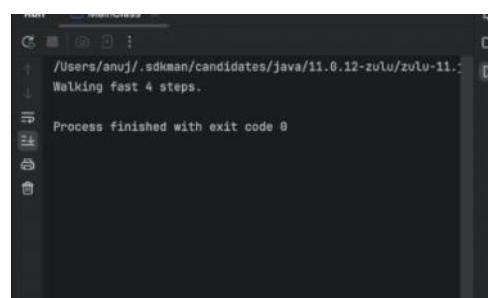
This method can work for only functional interfaces i.e. interfaces containing only one method.

```
public class MainClass {
    public static void main(String[] args) {
        //          return 0;
        //      }
        //  };

        Walkable obj = (int steps) -> {
            System.out.println("Walking fast "+ steps +" steps.");
            return 2*steps;
        };
    }
}

interface Walkable { 2 usages 1 implementation
    int walk(int steps);  no usages 1 implementation
}
```

```
4
5 ▷ public class MainClass {
6 ▷     public static void main(String[] args) {
7
8 //         Older way
9 //         Walkable obj = new WalkFast();
10 //         obj.walk(4);
11
12 //         Walkable obj = new Walkable() {
13 //             @Override
14 //             public int walk(int steps) {
15 //                 return 0;
16 //             }
17 //         };
18
19         Walkable obj = (int steps) -> {
20             System.out.println("Walking fast "+ steps +" steps.");
21             return 2*steps;
22         };
23
24     obj.walk(steps: 4);
25
26     }
27 }
28
29 ⓘ interface Walkable { 2 usages 1 implementation
30 ⓘ     int walk(int steps); 1 usage 1 implementation
31 }
32
```



The screenshot shows a Java IDE interface with a code editor and a terminal window. The code editor displays the MainClass.java file with the functional interface implementation. The terminal window shows the output of the program's execution:

```
/Users/anuj/.sdkman/candidates/java/11.0.12-zulu/zulu-11:
Walking fast 4 steps.

Process finished with exit code 0
```

```
... 5  public class MainClass {  
6      public static void main(String[] args) {  
7          //  
8          //      @Override  
9          //      public int walk(int steps) {  
10         //          return 0;  
11         //      }  
12     //  };  
13  
14     Walkable obj = (steps, isEnabled) -> {  
15         System.out.println("Walking fast "+ steps +" steps.");  
16         return 2*steps;  
17     };  
18  
19     Walkable obj2 = (steps, isEnabled) -> 2*steps;  
20  
21     obj.walk(steps: 4, isEnabled: true);  
22  
23 }  
24  
25  
26  
27  
28  
29 }  
30  
31 ① interface Walkable { 2 usages 1 implementation  
32 ②     int walk(int steps, boolean isEnabled); 1 usage 1 implementation  
33 }  
34
```

## Stream

```
List<String> fruits = List.of("Apple", "Banana", "Kiwi");  
Stream<String> stream = fruits.stream();|
```

```
List<String> fruits = List.of("Apple", "Banana", "Kiwi");  
Stream<String> stream = fruits.stream();  
  
stream.forEach((fruit) -> {  
    System.out.println(fruit);  
});  
}|
```

Stream once used can't be used again.

```

// Walkable obj2 = (steps, isEnabled) -> 2*steps;
// obj.walk(4, true);

List<String> fruits = List.of("Apple", "Banana", "Kiwi");

Stream<String> stream = fruits.stream();

stream.forEach((fruit) -> {
    System.out.println(fruit);
});

stream.forEach((fruit) -> {
    System.out.println(fruit);
});|
```

Here we already used stream, so can't use it again

```

List<String> fruits = List.of("Apple", "Banana", "Kiwi");

Stream<String> stream = fruits.stream();

stream
    .sorted()
    .forEach(fruit -> System.out.println(fruit));|
```

Printing the length only

```

List<String> fruits = List.of("Banana", "Apple", "Kiwi");

Stream<String> stream = fruits.stream();

stream
    .sorted() Stream<String>
    .map(fruit -> fruit.length()) Stream<Integer>
    .forEach(fruit -> System.out.println(fruit));|
```

Here, we have converted our stream to Integer.

```

List<String> fruits = List.of("Banana", "Apple", "Kiwi");

Stream<String> stream = fruits.stream();

stream
    Extract Surround // ⌘ : ring>
    .map(fruit -> fruit.length()) Stream<Integer>
    .map(fruitLength -> 2*fruitLength)
    .forEach(fruit -> System.out.println(fruit));|
```

```
List<String> fruits = List.of("Banana", "Apple", "Kiwi");

Stream<String> stream = fruits.stream();

stream
    .filter(fruit -> fruit.length() < 5) Stream<String>
    .sorted()
    .map(fruit -> fruit.length()) Stream<Integer>
    .map(fruitLength -> 2*fruitLength)
    .forEach(fruit -> System.out.println(fruit));

}

}

interface Walkable { no usages
    int walk(int steps, boolean isEnabled); no usages
```

```
List<String> fruits = List.of("Banana", "Apple", "Kiwi");

List<Integer> fruitsList = fruits
    .stream() Stream<String>
    .map(fruit -> fruit.length()) Stream<Integer>
    .collect(Collectors.toList());
```

```
List<String> fruits = List.of("Banana", "Apple", "Kiwi");

List<Integer> fruitsList = fruits
    .stream() Stream<String>
    .map(fruit -> fruit.length()) Stream<Integer>
    .collect(Collectors.toList());

System.out.println(fruitsList);
```

```
/Users/anuj/.sdkman/candidates/java/11.0.12-zulu/zulu-11.0.12-2020-07-11T14:45:45+0000+0000+0000_zulu-11.0.12-linux-x64
```

```
List<String> fruits = List.of("Banana", "Apple", "Kiwi", "Mango");

Set<Integer> fruitsList = fruits
    .stream() Stream<String>
    .map(fruit -> fruit.length()) Stream<Integer>
    .collect(Collectors.toSet());

System.out.println(fruitsList);
```

```
List<String> fruits = List.of("Banana", "Apple", "Kiwi", "Mango");

Map<String, Integer> fruitsList = fruits
    .stream()
    .map(fruit -> fruit.length())
    .collect(Collectors.toMap(
        fruit -> fruit,
        fruit -> fruit.length()
    )));
    I

System.out.println(fruitsList);
```

## What is Page in Spring Boot?

Page<T> is **not just a list**—it's a **container for paginated data plus extra info.**

Think of it like this:

- You have **1000 books** in your database.
- You don't want to load all at once (slow, memory-heavy).
- Instead, you fetch **small chunks (pages)**, e.g., 10 books at a time.

Page<T> gives you:

1. **The actual data** (e.g., 10 books).
2. **Metadata** about the whole dataset:
  - Total elements (e.g., 1000 books)
  - Total pages (e.g., 100 pages if size = 10)
  - Current page number
  - Whether there's a next/previous page

## How does it serve its purpose?

Imagine a REST API for books:

Java

```
@GetMapping("/books")
public Page<Book> getBooks(@RequestParam int page, @RequestParam int size) {
    return bookRepository.findAll(PageRequest.of(page, size));
}
..
```

Show more lines

- Client calls /books?page=0&size=10.
- Response includes:
  - content: 10 books
  - totalPages: e.g., 100
  - totalElements: e.g., 1000
  - number: current page index
  - hasNext: true/false

This makes **frontend pagination easy**—you don't need extra queries for totals.

@JoinTable(name="book\_category") tells JPA/Hibernate to  
create/use a third table to link Book and Category.

How does it know linking Book and Category? Could have been  
other entities as well?

In relational database, many-to-many relationships are implemented via a join table.

### 3) @JoinTable(...)

This specifies the **join table** and its foreign key columns:

- name = "book\_category": the table that connects books and categories.
- joinColumns = @JoinColumn(name = "book\_id"): the column in the join table that points to the Book primary key.
- inverseJoinColumns = @JoinColumn(name = "category\_id"): the column in the join table that points to the Category primary key.

Put simply: the book\_category table will have **two columns**:

SQL

```
book_id -- FK referencing book(id)
category_id -- FK referencing category(id)
```

## How JPA/Hibernate knows which two entities to link

### 1. Owning entity = the class where the field is declared

In your code, the annotation is on a field inside Book. So one side of the relationship is **Book**.

### 2. Target entity = the generic type of the collection (or targetEntity)

The field type is Set<Category>. So the other side of the relationship is **Category**.

- You can also specify it explicitly:

Java

```
@ManyToMany(targetEntity = Category.class)
private Set<Category> categories;
Show more lines
```

- If you omit targetEntity, Hibernate uses the generic type (Category) from Set<Category>.

### 3. @JoinTable just defines the details of the link table and FK column names

Java

```
@JoinTable(
    name = "book_category",
    joinColumns = @JoinColumn(name = "book_id"),
    inverseJoinColumns = @JoinColumn(name = "category_id"))
```

Show more lines

- name → the **join table** to use (book\_category).
- joinColumns → FK columns in book\_category that **point to Book** (the owning side).
- inverseJoinColumns → FK columns in book\_category that **point to Category** (the target side).

Put together: the association is **Book ↔ Category** because the field is in Book and its type is Set<Category>.

Typically

```

@ManyToMany
@JoinTable(
    name = "book_category",
    joinColumns = @JoinColumn(name = "book_id"),
    inverseJoinColumns = @JoinColumn(name = "category_id"))
private Set<Category> categories;
}

```

## Why Set<Category> and not just Category?

Because the relationship is **Many-to-Many**:

- One **Book** can have **many** categories.
- One **Category** can belong to **many** books.

So the field must be a **collection** type—i.e., it can hold **multiple Category instances**. A single Category (like just Category category;) would model a **Many-to-One** or **One-to-One** relationship, which doesn't match your case.

## Why specifically a Set (and not List)?

Both Set<Category> and List<Category> are supported by JPA/Hibernate, but Set is often preferred for @ManyToMany because:

### 1. No duplicates

A Set prevents the same Category from appearing twice in the collection, which mirrors how the **join table** should behave (typically with a UNIQUE (book\_id, category\_id) constraint).

This creates a many-to-many relationship between `Book` and `Category`, with a join table named `book_category`.

Great question! Let's break this down in **simple, beginner-friendly terms**.

## What does cascade = CascadeType.ALL mean?

In **JPA**, when you have relationships between entities (like Author → Book), you often need to perform operations on both sides. For example:

- If you **save** an Author, you probably want their Books saved too.
- If you **delete** an Author, you might want their Books deleted as well.

Doing this manually for every related entity can be tedious. **Cascade** solves this by saying:

"Whenever you do something to the parent entity, automatically do the same thing to its children."

## What is CascadeType.ALL?

CascadeType.ALL is a shortcut that means **apply all cascade operations**:

- **PERSIST** → Save the child when the parent is saved.

- **MERGE** → Update the child when the parent is updated.
- **REMOVE** → Delete the child when the parent is deleted.
- **REFRESH** → Refresh the child when the parent is refreshed.
- **DETACH** → Detach the child when the parent is detached from the persistence context.

So, CascadeType.ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}.

## Example in your code:

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(nullable = false)
    private String name;
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books;
}
```

This means:

- When you **save** an Author, all Books in books list will also be saved.
- When you **delete** an Author, all Books in books list will also be deleted.
- When you **update** an Author, all Books will also be updated.

## Why is this useful?

Without cascade, you'd have to do:

```
entityManager.persist(author);
for (Book book : author.getBooks()) {
    entityManager.persist(book);
}
```

With cascade:

```
entityManager.persist(author); // Books are saved automatically!
```

## Important Warning:

Be careful with CascadeType.REMOVE in **shared relationships**. If Book is linked to multiple authors or other entities, deleting one author could delete the book entirely, which might not be what you want.

## Quick analogy:

Think of **cascade** like a **domino effect**: Push the first domino (Author), and all connected dominos (Books) fall automatically.

Do you want me to also **explain each CascadeType individually with real-life examples** (like what PERSIST, MERGE, REMOVE do)? Or should I show you **a full working example with Author and Book in JPA?**

## Eager and Lazy Fetching:

Great question! Let's make **lazy fetching** super clear with a simple analogy and example.

### What does fetch = FetchType.LAZY mean?

In JPA, **fetch type** controls **when related data is loaded from the database**.

- **EAGER:** Load the related data **immediately** along with the parent entity.
- **LAZY:** Load the related data **only when you actually access it** in code.

### In your example:

```
@OneToMany(fetch = FetchType.LAZY)
```

```
private List<Book> books;
```

This means:

- When you query an Author, JPA **does NOT immediately fetch all the books**.
- Instead, it creates a **proxy** (a placeholder) for the books list.
- The actual SQL query to load books runs **only when you call author.getBooks()**.

### Why do this?

Imagine you have an Author with **1000 books**. If you only need the author's name, fetching all books upfront is wasteful.

**Lazy fetching saves performance** by loading data **on demand**.

### Example flow:

```
Author author = entityManager.find(Author.class, 1L);
// At this point: only Author data is loaded (id, name).
// books are NOT loaded yet.
System.out.println(author.getName()); // Still no books fetched.
List<Book> books = author.getBooks(); // NOW JPA runs a query to fetch books.
``
```

### What happens behind the scenes?

- JPA uses a **proxy object** for books.
- When you access books, Hibernate fires a query like: `SELECT * FROM book WHERE author_id = ?`

### Analogy:

Think of **LAZY** like **Netflix streaming**:

- You don't download all episodes at once.
- You stream an episode **only when you click play**.

**EAGER** is like downloading the entire season before watching even one episode.

### Important caveat:

If you access `author.getBooks()` **outside of an active session** (e.g., after closing EntityManager), you'll get a `LazyInitializationException`. Solution: either keep the session open, use **JOIN FETCH** in queries, or switch to EAGER (but EAGER can hurt performance).

↳ Do you want me to **show a full code example** comparing **LAZY vs EAGER** with the SQL queries they generate? Or a **diagram** showing how proxies work?

## Why use EAGER?

If you **always need the associated entity whenever you load the parent**, it's better to fetch it upfront to avoid extra queries later.

Example:

- Author and Profile have a **One-to-One** relationship.
- Every time you load an Author, you also need their Profile (e.g., for display).

Instead of doing **two queries** (one for Author, one for Profile), EAGER will fetch both in **one go**.

## **EntityManager**

Great question! entityManager is a **JPA interface** that acts as the main entry point for interacting with the persistence context (the first-level cache and database operations).

## What is EntityManager?

- It's provided by **JPA** (Java Persistence API).
- It manages **entities** and their lifecycle (persist, update, delete, find).
- It acts like a **bridge between your Java objects and the database**.

## What does entityManager.find(Author.class, 1L) do?

- find() loads an entity from the database by its **primary key**.
- Here:
  - Author.class → the type of entity you want.
  - 1L → the primary key value.
- JPA will:
  1. Check if the entity is already in the **persistence context** (cache).
  2. If not, run a SQL query like: SELECT \* FROM author WHERE id = 1;
  3. Return an Author object.

## Where does entityManager come from?

Usually, you **don't create it manually**. It's injected by the framework (Spring, Jakarta EE, etc.):

**In Spring Boot:**

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
@Service
```

```
public class AuthorService {  
    @PersistenceContext  
    private EntityManager entityManager;  
    public Author getAuthor(Long id) {  
        return entityManager.find(Author.class, id);  
    }  
}
```

Spring automatically provides the EntityManager connected to your database.

## Common operations with EntityManager:

- **Persist (insert):** entityManager.persist(author);
- **Find (select):** Author author = entityManager.find(Author.class, 1L);
- **Merge (update):**  
entityManager.merge(author);
- **Remove (delete):** entityManager.remove(author);

## Analogy:

Think of EntityManager as a **remote control for your database**:

- find() → fetch a record.
- persist() → add a record.
- remove() → delete a record.
- merge() → update a record.

⌚ Do you want me to **show a full example of using EntityManager in a Spring Boot service with Author and Book entities**, including how transactions work? Or a **diagram showing how EntityManager interacts with the persistence context and database**?

## Lazy vs Eager (More Lucidly)

Great question! Let's make this crystal clear with a **step-by-step explanation** of why **EAGER fetching means no extra query**.

## What happens with EAGER fetching?

When you mark a relationship as:

```
@OneToOne(fetch = FetchType.EAGER)  
private Profile profile;
```

and then call:

```
Author author = entityManager.find(Author.class, 1L);
```

Hibernate knows:

- You want the Author.
- And because of **EAGER**, you also want the associated Profile **immediately**.

So Hibernate generates **one SQL query with a JOIN**:

```
SELECT a, p.  
FROM author a  
LEFT JOIN profile p ON a.profile_id = p.id  
WHERE a.id = 1;
```

This query fetches:

- All columns from author (like id, name).
- All columns from profile (like id, bio, etc.).

**Result:** Both Author and its Profile are loaded into memory in one go.

## Why no extra query?

Because the JOIN query already includes the Profile data.\ So when you later call:

```
author.getProfile();
```

Hibernate **does NOT need to hit the database again**—the Profile object is already in the persistence context.

## Compare with LAZY fetching:

If you had:

```
@OneToOne(fetch = FetchType.LAZY)  
private Profile profile;
```

Then:

```
Author author = entityManager.find(Author.class, 1L);
```

Hibernate runs:

```
SELECT * FROM author WHERE id = 1;  
``
```

Only the Author is loaded.\ When you later call:

```
author.getProfile();
```

Hibernate fires **a second query**:

```
SELECT * FROM profile WHERE id = ?;
```

So **LAZY = 2 queries, EAGER = 1 query**.

## Analogy:

- **EAGER** = You order a **combo meal** (burger + fries) → Both come together.
- **LAZY** = You order a burger first, then later decide to order fries → Two separate trips.

## When is EAGER useful?

- For **One-to-One** or **Many-to-One** relationships where the associated entity is **always needed**.
- Example: User and Profile—you almost always show both together.

**⚠ Warning:**\ Don't use EAGER on large collections (@OneToMany, @ManyToMany) because:

- It can load thousands of rows unnecessarily.
- It can cause performance issues and huge JOIN queries.

 Do you want me to **show a full code example comparing LAZY vs EAGER with the actual SQL queries Hibernate generates**? Or a **diagram showing how data flows in both cases**?

## N+1 Select Problem

Great question! The **N+1 Select Problem** is one of the most common performance issues in JPA/Hibernate, and it can be confusing at first. Let me explain it **step by step with a simple example**.

### What does this code do?

```
List<Author> authors = entityManager.createQuery("SELECT a FROM Author a").getResultList();
for (Author author : authors) {
    System.out.println(author.getBooks()); // This triggers a separate query for each author
}


- First line:\ SELECT a FROM Author a → Hibernate runs one query to fetch all authors.\ Example SQL: SELECT * FROM author; Suppose this returns 10 authors.
- Second part:\ For each author, you call author.getBooks().\ If books is LAZY (default for @OneToOne), Hibernate does not load books initially.\ So when you access getBooks(), Hibernate fires another query: SELECT * FROM book WHERE author_id = ?; This happens once per author.

```

### Why is it called N+1?

- 1 query** for all authors.
- N queries** for their books (where N = number of authors).
- Total = **N + 1 queries**.

Example:

- Authors: 10
- Queries: 1 (authors) + 10 (books) = 11 queries.

### Why is this bad?

- If you have **100 authors**, that's **101 queries**.
- If each author has multiple books, this becomes very slow.
- It's a classic performance bottleneck.

### How to fix it?

Use **JOIN FETCH** to load authors and their books in **one query**:

```
List<Author> authors = entityManager.createQuery(
    "SELECT a FROM Author a JOIN FETCH a.books"
).getResultList();
Hibernate runs:
```

```
SELECT a, b FROM author a
JOIN book b ON a.id = b.author_id;
```

Now:

- All authors **and** their books are fetched in **one query**.
- No extra queries when calling `author.getBooks()`.

## **Analogy:**

- **N+1 problem** = You go to the store, buy 1 item, then go back N times for each additional item.
- **JOIN FETCH** = You buy everything in one trip.

## **TL;DR**

- N+1 happens because **LAZY loading** triggers extra queries inside loops.
- Fix it with **JOIN FETCH** or **EntityGraph** to load related data upfront.

☞ Do you want me to **show a full working example with Author-Book entities, the queries Hibernate generates for LAZY vs JOIN FETCH, and how to avoid N+1 in Spring Boot?** Or a **diagram showing the difference visually?**

## **1. What is a Repository in JPA?**

A **repository** is basically an interface that acts as a **bridge between your application and the database**. Instead of writing SQL queries manually, you use repository interfaces provided by **Spring Data JPA** to perform database operations.

Think of it like this:

- You want to **save, update, delete, or fetch** data from the database.
- Instead of writing boilerplate code (like EntityManager or raw SQL), you just call methods from the repository.

## **2. Types of Repositories**

Spring Data JPA provides several ready-made interfaces:

- **CrudRepository**
  - Basic CRUD operations: `save()`, `findById()`, `delete()`, etc.
- **JpaRepository**
  - Extends CrudRepository and adds **JPA-specific features** like:
    - `flush()` (forces changes to DB immediately)
    - Batch operations
    - Better support for pagination and sorting
- **PagingAndSortingRepository**
  - Adds methods for **pagination** and **sorting**.

## **3. Extending JpaRepository**

When you create your own repository, you usually **extend JpaRepository** because it gives you all the CRUD methods **plus extra JPA features**.

Example:

Java

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
}  
``
```

Show more lines

Here:

- Employee → Your entity class
- Long → Type of the primary key

Now you can use:

Java

```
employeeRepository.findAll();
employeeRepository.save(employee);
employeeRepository.deleteById(1L);
``
```

Show more lines

**No need to write SQL or JPQL manually!**

## 4. Custom Query Methods

Spring Data JPA lets you define **custom methods** by just following naming conventions:

Example:

Java

```
List<Employee> findByDepartment(String department);
```

Show more lines

Spring will automatically generate the query:

SQL

```
SELECT * FROM employee WHERE department = ?
```

Show more lines

You can also use **@Query** for more complex queries:

Java

```
@Query("SELECT e FROM Employee e WHERE e.salary > :salary")
List<Employee> findEmployeesWithSalaryGreaterThan(@Param("salary") double salary);
``
```

Show more lines

## Why is this powerful?

- No boilerplate code
- Automatic query generation
- Easy to maintain
- Supports custom queries when needed

☞ Do you want me to **draw a simple diagram** showing how JpaRepository fits into the hierarchy and how custom repositories work? Or should I **write a full example with Entity + Repository + Service + Controller** for better clarity?

# Custom query method

Java

```
List<Employee> findByDepartment(String department);
```

Spring will automatically generate the query:

is findByDepartment defined by Springboot JPA?

Great question! The method `findByDepartment(String department)` is **not predefined by Spring Boot JPA**. Instead, it is a **custom query method** that Spring Data JPA automatically implements for you based on its **method name conventions**.

Here's how it works:

- Spring Data JPA looks at the method name: `findByDepartment` → It sees **findBy** followed by a property name (`department`).
- It matches `department` with a field in your **Entity class** (e.g., `Employee` has a field `department`).
- It then automatically generates the query: `SELECT e FROM Employee e WHERE e.department = ?1`
- You **don't need to write any SQL or JPQL** for this. Spring does it for you at runtime.

So:

- You define the method in your repository interface.
- Spring Data JPA parses the name and creates the query dynamically.

#### Example:

```
@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    private String department;
    // getters and setters
}
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<Employee> findByDepartment(String department);
}
```

Now you can call:

```
employeeRepository.findByDepartment("IT");
Spring will return all employees whose department is "IT".
```

☞ Do you want me to **list all common naming conventions** (like `findByName`, `findBySalaryGreater Than`, `findByDepartmentAndRole`) that Spring Data JPA supports? Or should I **show you how to use @Query for custom queries?**

## Common Query Method Keywords

Some commonly used keywords for query methods:

- **findBy**: Finds entities based on certain properties.
- **existsBy**: Checks if an entity exists based on certain properties.
- **deleteBy**: Deletes entities based on certain properties.
- **countBy**: Counts entities based on certain properties.

Here's how you can use query methods in a repository:

```
public interface BookRepository extends JpaRepository<Book, Long> {

    // Find books by title
    List<Book> findByTitle(String title);

    // Find books by title or author
    List<Book> findByTitleOrAuthor(String title, String author);

    // Find books by a certain price range
    List<Book> findByPriceBetween(Double minPrice, Double maxPrice);

    // Find books ordered by title
    List<Book> findByOrderByTitleAsc();
}
```

- **findByTitle(String title)**: Finds all books with the given title.
- **findByTitleOrAuthor(String title, String author)**: Finds books with a matching title or author.
- **findByPriceBetween(Double minPrice, Double maxPrice)**: Finds books within the given price range.
- **findByOrderByTitleAsc()**: Finds books ordered by their titles in ascending order.

## 5. Examples of Query Methods

### Example 1: Find Books by Title

```
// Repository method
List<Book> findByTitle(String title);

// Usage in service or controller
List<Book> books = bookRepository.findByTitle("Spring in Action")
```

This method retrieves all books with the title "Spring in Action".

### Example 2: Find Books by Title or Author

```
// Repository method
List<Book> findByTitleOrAuthor(String title, String author);

// Usage in service or controller
List<Book> books = bookRepository.findByTitleOrAuthor("Spring", 'Cra' +
```

This query retrieves books that either have the title "Spring" or are written by "Craig Walls".

If you want to write the query yourself instead of relying on method-name conventions, Spring Data JPA gives you several clean options—each suited to different needs.

- 1) @Query with **JPQL** (most common)

```
1 public interface EmployeeRepository extends JpaRepository<Employee, Long> {
2
3     // Simple filter
4     @Query("SELECT e FROM Employee e WHERE e.department = :dept")
5     List<Employee> findByDepartment(@Param("dept") String department);
6
7     // Multiple conditions
8     @Query("""
9             SELECT e FROM Employee e
10            WHERE e.department = :dept
11              AND e.salary >= :minSalary
12            ORDER BY e.salary DESC
13             """)
14     List<Employee> findByDeptWithMinSalary(
15         @Param("dept") String department,
16         @Param("minSalary") Double minSalary
17     );
18 }
```

## Method 1: Simple filter using @Query

```
1 @Query("SELECT e FROM Employee e WHERE e.department = :dept")
2 List<Employee> findByDepartment(@Param("dept") String department);
```

### What happens at runtime

1. You call `employeeRepository.findByDepartment("IT")`.
2. Spring injects "IT" for `:dept`, creating the JPQL:

```
SELECT e FROM Employee e WHERE e.department = 'IT'
```

### What this means

- `@Query(...)` is JPQL (Java Persistence Query Language), **not raw SQL**.
  - JPQL uses **entity names** and **field names** (e.g., `Employee`, `department`), not table/column names.
- `:dept` is a **named parameter** in the query.
- `@Param("dept") String department` tells Spring: bind the method argument `department` to the query's `:dept`.

## Method 2: Multiple conditions + ordering

☰ 🔍 ⌂ ↻ ⌂ Java

```
1 @Query("""
2     SELECT e FROM Employee e
3     WHERE e.department = :dept
4         AND e.salary >= :minSalary
5     ORDER BY e.salary DESC
6     """)
7 List<Employee> findByDeptWithMinSalary(
8     @Param("dept") String department,
9     @Param("minSalary") Double minSalary
10 );
11 ``
```

### What this means

- This is still **JPQL**—we're selecting Employee entities.
- We filter by **two parameters**:
  - department equals :dept
  - salary is **greater than or equal** to :minSalary
- We sort the results by salary in **descending** order (highest first).
- The `""" ... """` is a **Java text block** (available since Java 15+), which lets you write multi-line strings neatly.

### What happens at runtime

1. You call:

```
Java
employeeRepository.findByDeptWithMinSalary("IT", 100000.0);
Show more lines
```

2. Spring binds:
  - :dept → "IT"
  - :minSalary → 100000.0
3. The JPQL becomes:

```
SELECT e FROM Employee e
WHERE e.department = 'IT' AND e.salary >= 100000.0
ORDER BY e.salary DESC
```
4. JPA executes this and returns a **sorted list of Employee** objects.

### Key points

- **Parameter binding**: Names in @Param must match the :name in the query.
- **Ordering**: Done in the query itself; no need for separate sorting in code.
- **Type safety**: JPQL uses entity field types (String for department, Double for salary).

### Example 3: Custom Query using JPQL

```
// Custom query method in the repository
@Query("SELECT b FROM Book b WHERE b.title LIKE %:keyword%")
List<Book> searchBooksByKeyword(@Param("keyword") String keyword)

// Usage in service or controller
List<Book> books = bookRepository.searchBooksByKeyword("Hibernate")
```

This custom query searches for books with the keyword "Hibernate" in the title.

## MySQL with Spring Data JPA

MySQL command line client

After starting MySQL, create a new database:

```
CREATE DATABASE springbootdb;
```

In your `pom.xml`, add the following dependencies:

```
<dependencies>
    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- MySQL Connector -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

### 3. Configure Spring Boot to Connect with MySQL

In the `application.properties` file (or `application.yml` if you're using YAML), configure the connection to the MySQL database.

#### `application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb
spring.datasource.username=root
spring.datasource.password=yourpassword

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

## 1. What is Pagination and Sorting?

- **Pagination:** Instead of fetching thousands of rows at once (which is slow and memory-heavy), you fetch data **page by page**.  
Example: Page 1 → 10 records, Page 2 → next 10 records.
- **Sorting:** Arrange the data in **ascending or descending order** based on one or more fields (e.g., sort books by title or price).

## 2. Pageable and Sort Interfaces

Spring Data JPA gives you two helper interfaces:

- **Pageable:** Defines **page number**, **page size**, and optionally **sorting**.
- **Sort:** Defines sorting rules (field name + direction).

These can be passed directly to repository methods.

## 3. How Pagination Works

Your repository already has:

Java

```
public interface BookRepository extends JpaRepository<Book, Long> {  
}  
``
```

Show more lines

Because it extends JpaRepository, you automatically get:

Java

```
Page<Book> findAll(Pageable pageable);
```

Show more lines

## 4. Using Pageable

You create a Pageable object like this:

Java

```
Pageable pageable = PageRequest.of(0, 5); // Page 0, size 5  
Page<Book> page = bookRepository.findAll(pageable);  
``
```

Show more lines

- PageRequest.of(pageNumber, pageSize)
- Page numbers start at **0**.
- Page<Book> gives you:
  - getContent() → List of books
  - getTotalPages() → Total number of pages
  - getTotalElements() → Total records
  - hasNext() → Is there another page?

```
Pageable pageable = PageRequest.of(0, 5); // Page 0, size 5
Page<Book> page = bookRepository.findAll(pageable);
``
```

Is it giving me only 0th page books

Yes, exactly!

```
Pageable pageable = PageRequest.of(0, 5); // Page 0, size 5
Page<Book> page = bookRepository.findAll(pageable);
This will only fetch the first page (page index = 0) with 5 records from the Book table.
```

## Why only the 0th page?

- PageRequest.of(0, 5) means:
  - 0 → Page number (zero-based index)
  - 5 → Page size (number of records per page)
- So Spring Data JPA runs a query like: SELECT \* FROM book LIMIT 5 OFFSET 0; (In SQL terms: start from row 0, fetch 5 rows)

## How to get the next pages?

- For page 1 (second page):  
Pageable pageable = PageRequest.of(1, 5); // Page 1, size 5  
Page<Book> page = bookRepository.findAll(pageable);  
This will run: SELECT \* FROM book LIMIT 5 OFFSET 5;
- For page 2:  
PageRequest.of(2, 5); // OFFSET = 10

## How to add sorting?

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("title").ascending());
Page<Book> page = bookRepository.findAll(pageable);
``
```

This will run:  
SELECT \* FROM book ORDER BY title ASC LIMIT 5 OFFSET 0;

## How to check metadata?

Page<Book> gives you:

- page.getContent() → List of books in this page
- page.getTotalPages() → Total number of pages
- page.getTotalElements() → Total number of records
- page.hasNext() → Is there another page?

☞ Do you want me to **show a full example with a REST API** where you pass page, size, and sort as query parameters (like /books?page=0&size=5&sort=title,asc) and Spring handles everything automatically? Or should I give you a **diagram explaining how LIMIT/OFFSET works behind the scenes?**

Yes, exactly!

```
Pageable pageable = PageRequest.of(0, 5); // Page 0, size 5
```

```
Page<Book> page = bookRepository.findAll(pageable);
This will only fetch the first page (page index = 0) with 5 records from the Book table.
```

## Why only the 0th page?

- PageRequest.of(0, 5) means:
  - 0 → Page number (zero-based index)
  - 5 → Page size (number of records per page)
- So Spring Data JPA runs a query like: SELECT \* FROM book LIMIT 5 OFFSET 0; (In SQL terms: start from row 0, fetch 5 rows)

## How to get the next pages?

- For page 1 (second page):  
Pageable pageable = PageRequest.of(1, 5); // Page 1, size 5  
Page<Book> page = bookRepository.findAll(pageable);  
This will run: SELECT \* FROM book LIMIT 5 OFFSET 5;
- For page 2:  
PageRequest.of(2, 5); // OFFSET = 10

## How to add sorting?

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("title").ascending());
Page<Book> page = bookRepository.findAll(pageable);
``
```

This will run:  
SELECT \* FROM book ORDER BY title ASC LIMIT 5 OFFSET 0;

## How to check metadata?

Page<Book> gives you:

- page.getContent() → List of books in this page
- page.getTotalPages() → Total number of pages
- page.getTotalElements() → Total number of records
- page.hasNext() → Is there another page?

☞ Do you want me to **show a full example with a REST API** where you pass page, size, and sort as query parameters (like /books?page=0&size=5&sort=title,asc) and Spring handles everything automatically? Or should I give you a **diagram explaining how LIMIT/OFFSET works behind the scenes?**

## 5. Adding Sorting

You can add sorting to PageRequest:

Java  
Pageable pageable = PageRequest.of(0, 5, Sort.by("title").ascending());  
Page<Book> page = bookRepository.findAll(pageable);  
Show more lines  
Or multiple fields:

Java  
Sort sort = Sort.by("price").descending().and(Sort.by("title").ascending());  
Pageable pageable = PageRequest.of(0, 10, sort);  
Show more lines

## 6. Example Controller

Java  
@GetMapping("/books")  
public Page<Book> getBooks(@RequestParam int page, @RequestParam int size) {  
 Pageable pageable = PageRequest.of(page, size, Sort.by("title").ascending());  
 return bookRepository.findAll(pageable);  
}  
"  
Show more lines  
Call:  
GET /books?page=0&size=5

### Why is this powerful?

- No manual SQL for pagination.
- Works with any database supported by JPA.
- Handles large datasets efficiently.
- Built-in metadata (total pages, elements, etc.).

## 6. Examples

### Example 1: Paginating Through Authors

```
public void getPaginatedAuthors(int page, int size) {  
    Pageable pageable = PageRequest.of(page, size);  
    Page<Author> authorPage = authorRepository.findAll(pageable);  
  
    List<Author> authors = authorPage.getContent();  
    authors.forEach(System.out::println);  
}
```

### Why do we need locking?

Imagine two users editing the same record at the same time:

- User A loads Employee record (salary = 50,000).
- User B loads the same Employee record (salary = 50,000).
- User A updates salary to 60,000 and saves.
- User B updates salary to 70,000 and saves.

Without locking, **User B's update will overwrite User A's update**, and we lose data integrity.

### Optimistic Locking

### What's happening in Optimistic Locking?

Imagine you have a **shared document** (like Google Docs), and two people open it at the same time:

- **User A** opens the document (version = 1).
- **User B** opens the same document (version = 1).

Both start editing.

## Step 1: User A saves changes

- JPA sends an update query:

```
1 UPDATE employee
2 SET salary = 60000, version = 2
3 WHERE id = ? AND version = 1;
```

- This means:
  - Update the row **only if its version is still 1.**
  - After updating, set version to 2.

Success! Because the row was version 1.

## Step 2: User B tries to save changes

- JPA sends:

```
1 UPDATE employee
2 SET salary = 70000, version = 2
3 WHERE id = ? AND version = 1;
4
```

- But now the row's version is 2, not 1 (because User A already updated).
- So **no rows match the WHERE condition.**
- JPA detects this and throws **OptimisticLockException**.

## Why does this happen?

Because **User B's copy is stale**. It was based on version 1, but the DB is now version 2. This prevents overwriting User A's changes accidentally.

## Key Idea:

- **Optimistic Locking doesn't lock the row upfront.**
- It just checks at **commit time**:  
"Is the version still what I read? If yes, update. If not, fail."

## Real-world analogy:

Think of a **ticket counter**:

- You take token #1 (version 1).
- Someone else takes token #1 too.
- When you go to the counter, they say:  
"Sorry, token #1 was already served. You need to refresh and try again."

Optimistic locking example:

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private int quantity;

    @Version
    private int version; // Version field for optimistic locking

    // Getters and Setters
}
```

### Service Layer Example:

```
@Transactional
public void updateProductQuantity(Long productId, int newQuantity) {
    Product product = productRepository.findById(productId).orElse(null);
    product.setQuantity(newQuantity);
    productRepository.save(product); // Version is checked here
}
```

If two transactions try to update the same product concurrently, the one that finishes first will succeed, while the other will fail due to a version mismatch.

### Pessimistic Locking in JPA (with Example)

Pessimistic locking ensures that data is locked during the entire transaction, preventing other transactions from reading or modifying it.

### Example:

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private int quantity;

    // No version field needed for pessimistic locking
}
```

### Service Layer

```
@Transactional
public void updateProductQuantityWithLock(Long productId, int newQuantity) {
    Product product = productRepository.findById(productId, LockModeType.PESSIMISTIC_WRITE)
        .orElseThrow();
    product.setQuantity(newQuantity);
    productRepository.save(product);
}
```

## What is a Configuration Class in Spring Boot?

- A class annotated with `@Configuration` tells Spring that this class contains **bean definitions**.
- Beans are objects managed by the Spring IoC (Inversion of Control) container.
- Instead of creating objects manually with `new`, you define them as beans, and Spring injects them wherever needed.

## What is JPA Auditing?

JPA Auditing is a feature in Spring Data JPA that **automatically fills audit-related fields** in your entities, such as:

- **Created Date** → When the record was first inserted.
- **Last Modified Date** → When the record was last updated.
- **Created By / Modified By** → Which user performed the action (optional).

Instead of manually setting these values in every entity, Spring does it for you.

## How does it work?

1. You enable auditing with:

```
Java  
@EnableJpaAuditing  
Show more lines  
in a configuration class (like your MapperConfig).
```

2. In your entity, you add special annotations:

```
Java  
@Entity  
public class Product {  
    @Id  
    private Long id;  
  
    @CreatedDate  
    private LocalDateTime createdDate;  
  
    @LastModifiedDate  
    private LocalDateTime lastModifiedDate;  
}  
Show more lines
```

3. Spring automatically populates these fields when:
  - You **save** a new entity → createdDate is set.
  - You **update** an entity → lastModifiedDate is updated.

## Why is this useful?

- No need to manually set timestamps in your code.
- Ensures consistency across all entities.
- Great for auditing and tracking changes in your database.

# Auditing and Advance Auditing with Hibernate Evers

## How do you enable it?

You use the annotation:

```
1 @EnableJpaAuditing
```

on a **configuration class** (usually your main Spring Boot application class or a separate config class).

This tells Spring: "Turn on auditing for JPA entities."

## How does Spring know the current user?

For fields like `CreatedBy` and `LastModifiedBy`, Spring needs to know **who is performing the action**. That's where **AuditorAware** comes in.

### What is AuditorAware?

It's an interface you implement to provide the current user.

Example:

```
1 @Component
2 public class AuditorAwareImpl implements AuditorAware<String> {
3     @Override
4     public Optional<String> getCurrentAuditor() {
5         // Example: Get username from Security Context
6         return
7             Optional.ofNullable(SecurityContextHolder.getContext().getAuthentication().getName());
8     }
}
```

Here:

- `getCurrentAuditor()` returns the username of the logged-in user.
- Spring uses this value to populate `@CreatedBy` and `@LastModifiedBy` fields.

## Putting it all together

1. Add `@EnableJpaAuditing` in your config class.
2. Create an `AuditorAware` implementation to return the current user.
3. Annotate your entity fields with:
  - `@CreatedDate`
  - `@LastModifiedDate`
  - `@CreatedBy`
  - `@LastModifiedBy`

Spring will automatically fill these fields during `save()` or `update()` operations.

## What is AuditorAware?

AuditorAware<T> is an **interface provided by Spring Data JPA**.

Its job: **Tell Spring who the current user (auditor) is** when saving or updating an entity.

Why? Because when you use annotations like @CreatedBy or @LastModifiedBy in your entity, Spring needs to know **which user performed the action**.

## What is an HTTP client?

An **HTTP client** is a piece of software (or a library in your code) that allows your application to **send HTTP requests** and **receive HTTP responses** over the web. In other words, it's the tool that helps your program talk to other servers using the **HTTP protocol** (the same protocol browsers use to load web pages).

## Examples in Java

- **RestTemplate** → Older, synchronous client (blocking).
- **WebClient** → Modern, reactive, non-blocking client.
- **RestClient** → Newer, simpler API introduced in Spring 6.

Yes, exactly! **Postman is an HTTP client**, but with a **graphical interface** instead of being a Java library.

Here's why Postman qualifies as an HTTP client:

- It lets you **send HTTP requests** (GET, POST, PUT, DELETE, etc.) to any server.
- It shows you the **HTTP response** (status code, headers, body).
- It supports **authentication**, headers, query params, and body formats (JSON, XML, form-data).
- It can handle **HTTPS** and other advanced features like cookies, environment variables, and scripting.

So, Postman is basically a **tool for developers to test APIs** without writing code, while libraries like **RestTemplate**, **WebClient**, and **RestClient** are used **inside your Java application** to make HTTP calls programmatically.