

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 code in 'Module1IntroductionApplication.java' has been updated to include imports, the @SpringBootApplication annotation, and a main method that creates an instance of 'PaymentService' and calls its 'pay()' method. The code editor shows the updated code with syntax highlighting.

```

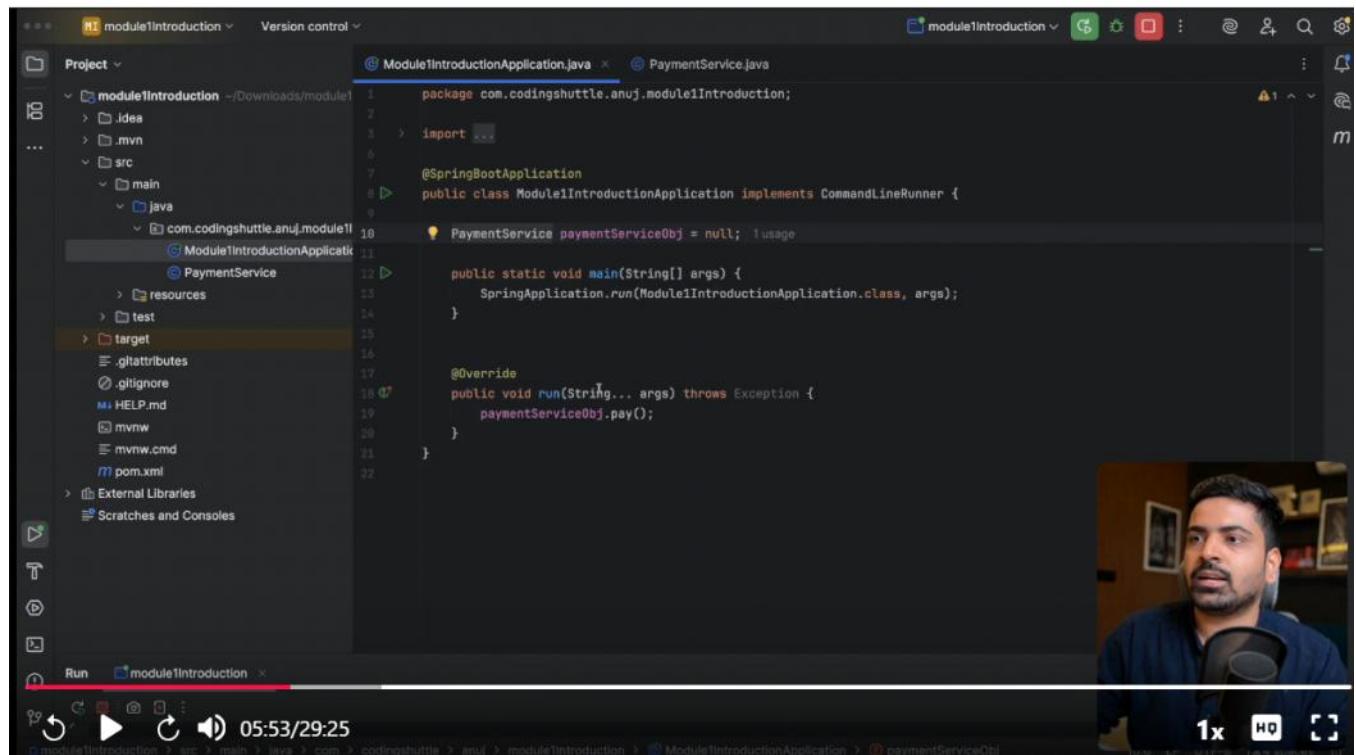
package com.codingshuttle.anuj.module1Introduction;
import ...;
@SpringBootApplication
public class Module1IntroductionApplication {

    public static void main(String[] args) {
        SpringApplication.run(Module1IntroductionApplication.class, args);
        PaymentService paymentServiceObj = new PaymentService();
        paymentServiceObj.pay();
    }
}

```

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 left sidebar displays the project structure with a focus on the "target" folder. The main editor window shows two files: "Module1IntroductionApplication.java" and "PaymentService.java". The code in "Module1IntroductionApplication.java" 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 code in "PaymentService.java" is partially visible at the top of the editor.

After using @Autowired ,

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

```
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 the application starting and executing the `pay()` method:

```
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 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
```

A video feed of the speaker is visible in the bottom right corner, and the play button indicates the video is currently playing.

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 code:

```
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();
    }
}
```

A tooltip "Extract Surround" is shown over the `paymentService()` method. A video feed of the speaker is visible in the bottom right corner.

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 interface.

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.

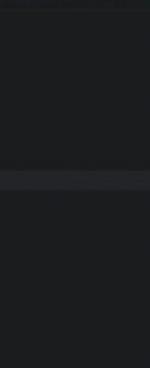
But in the main method, we create

```
@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)



```
module1Introduction main

ect <

module1Introduction ~/Downloads/module1Introduction

  .idea
  .mvn
  src
    main
      java
        com.codingshuttle.anuj.module1Introduction
          EmailNotificationService
          SmsNotificationService
          AppConfig
          Module1IntroductionApplication
          NotificationService
          PaymentService
        impl
          EmailNotificationService
          SmsNotificationService
        Module1IntroductionApplication
        AppConfig.java
        NotificationService.java
        SmsNotificationService.java
        module1Introduction

  2025-11-14T18:38:19.244+05:30 INFO 63514 --- [module1Introduction] [main] o.apache.catalina.core.StandardService : Starting
  2025-11-14T18:38:19.244+05:30 INFO 63514 --- [module1Introduction] [main] o.apache.catalina.core.StandardEngine : Starting
  2025-11-14T18:38:19.259+05:30 INFO 63514 --- [module1Introduction] [main] o.a.c.c.C[Tomcat].[localhost].[] : Initiali
  2025-11-14T18:38:19.259+05:30 INFO 63514 --- [module1Introduction] [main] w.s.c.ServletWebServerApplicationContext : Root Web
  2025-11-14T18:38:19.358+05:30 INFO 63514 --- [module1Introduction] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat s
```

```
Module1IntroductionApplication.java x PaymentService.java AppConfig.java NotificationService.java SmsNotificationService.java module1Introduction

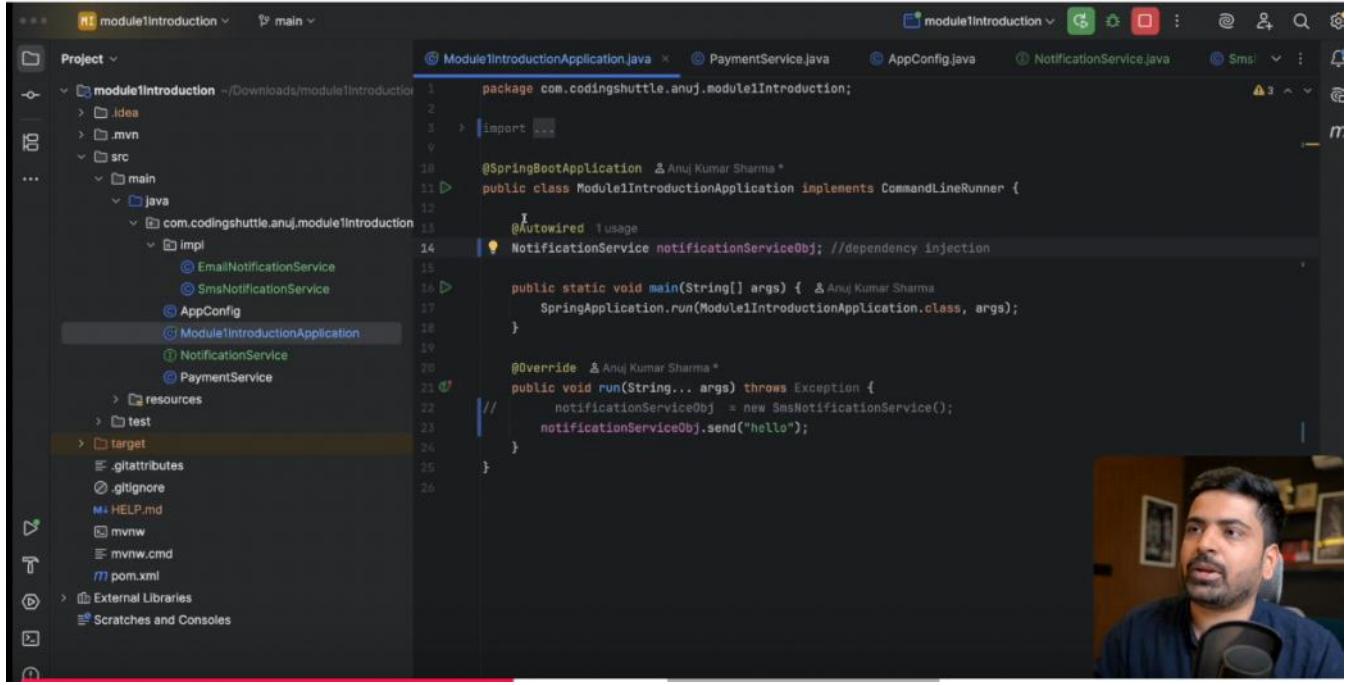
1 package com.codingshuttle.anuj.module1Introduction;
2
3 > import ...
4
5
6 @SpringBootApplication & Anuj Kumar Sharma "
7 public class Module1IntroductionApplication implements CommandLineRunner {
8
9     @Autowired 2 usages
10    NotificationService notificationObj;
11
12    public static void main(String[] args) { & Anuj Kumar Sharma
13        SpringApplication.run(Module1IntroductionApplication.class, args);
14    }
15
16    @Override & Anuj Kumar Sharma"
17    public void run(String... args) throws Exception {
18        notificationObj = new SmsNotificationService();
19        notificationObj.send("hello");
20    }
21
22
23
24
25 }
```

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

Then make one of em as @Primary

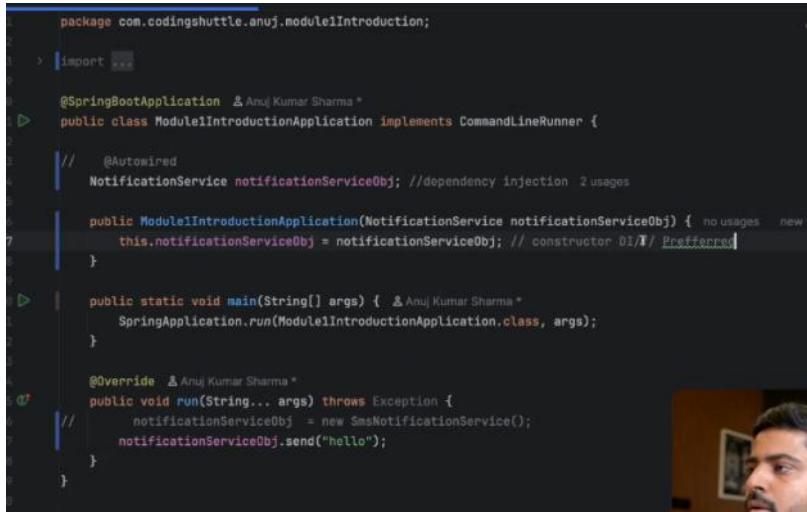
We should use constructor dependencies.

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  ● AppConfig.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     ⚡ Extract Surround // ≡ :
14     @Override
15     public void send(String message) {
16         System.out.println("Email sending... "+message);
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.

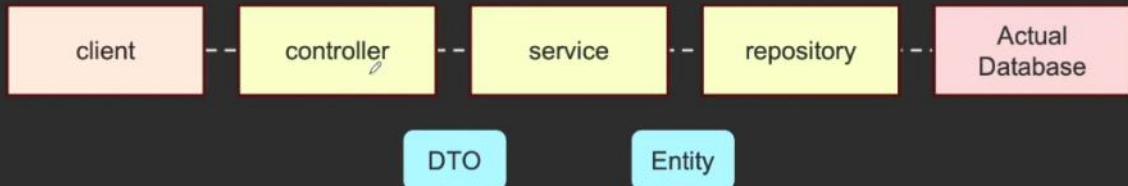
In short:

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.

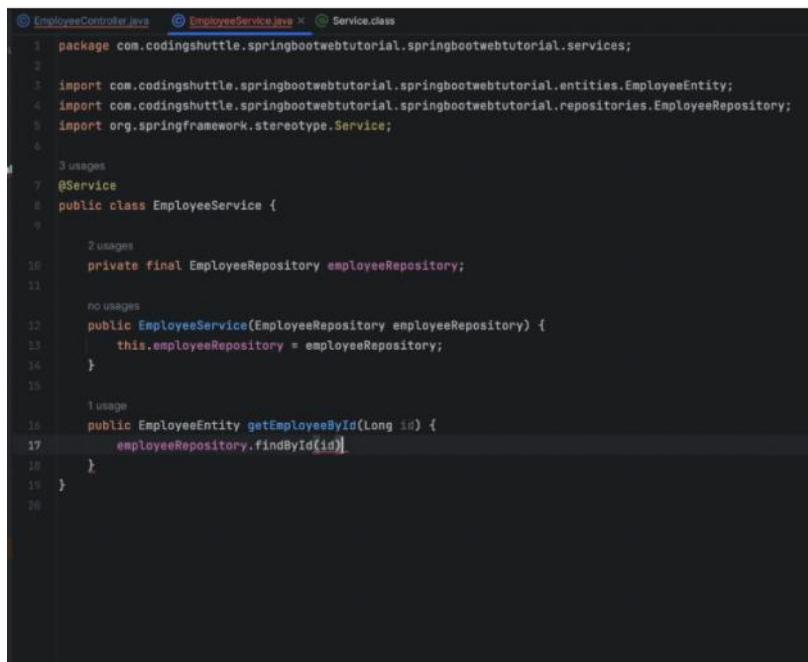


The Service Layer

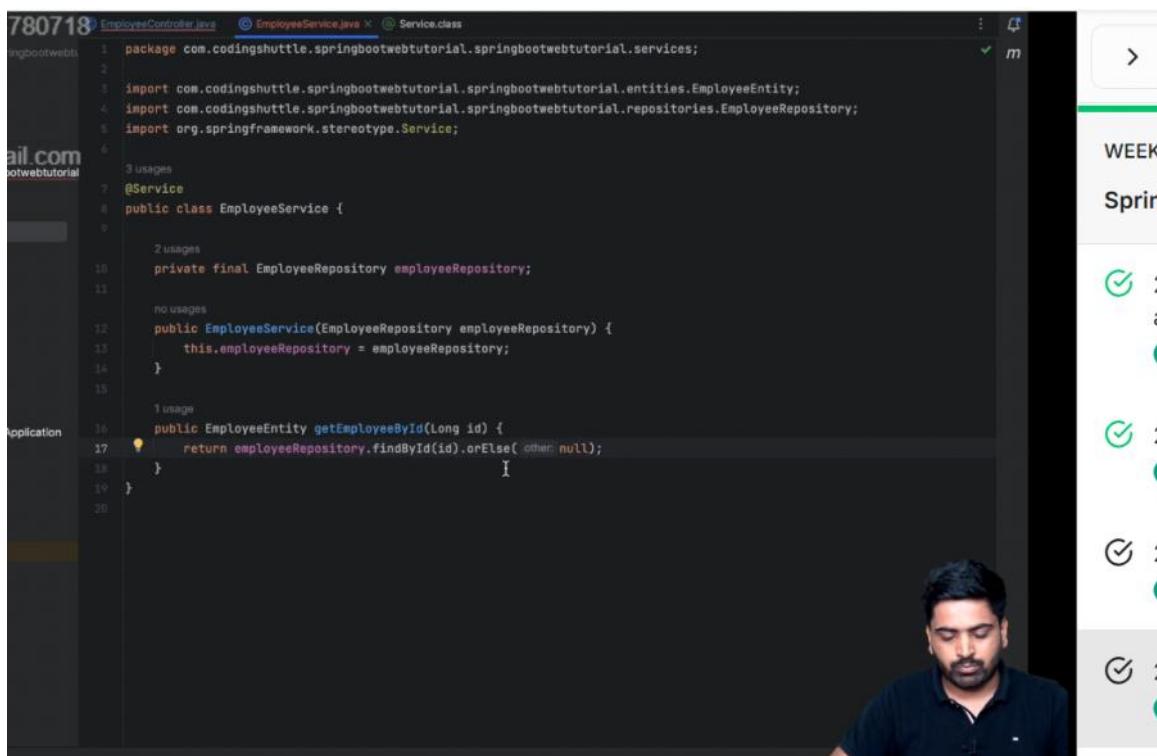
we would be removing EmployeeRepository object from EMployeeController as it is not a good practice.
Rather add EmployeeService object there.

```
@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

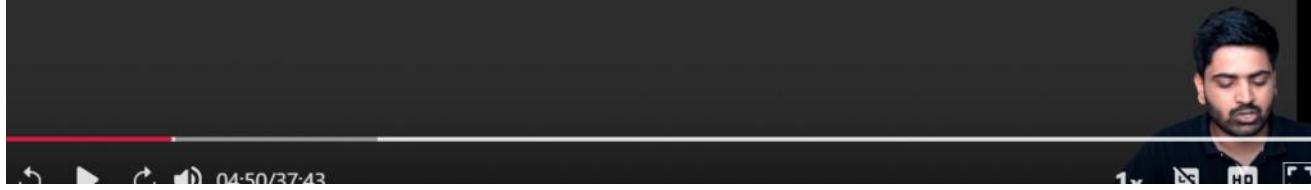
Do you want me to: ✓ Show full working Spring Boot project structure with ModelMapper? ✓ Or explain Java Streams & Lambdas (since you asked earlier) in the context of mapping? ✓ Or give you unit test examples for ModelMapper mappings?

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

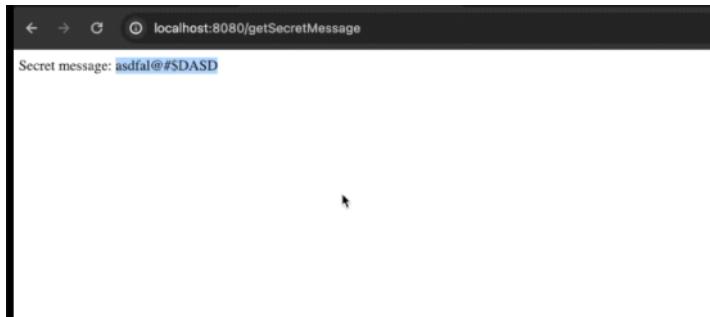
arka8135@gmail.com

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.springbootwebtutorial.controllers EmployeeController.java
```

```
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  SpringBootWebTutorialApplication.java  SpringApplication.class
```

```
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`

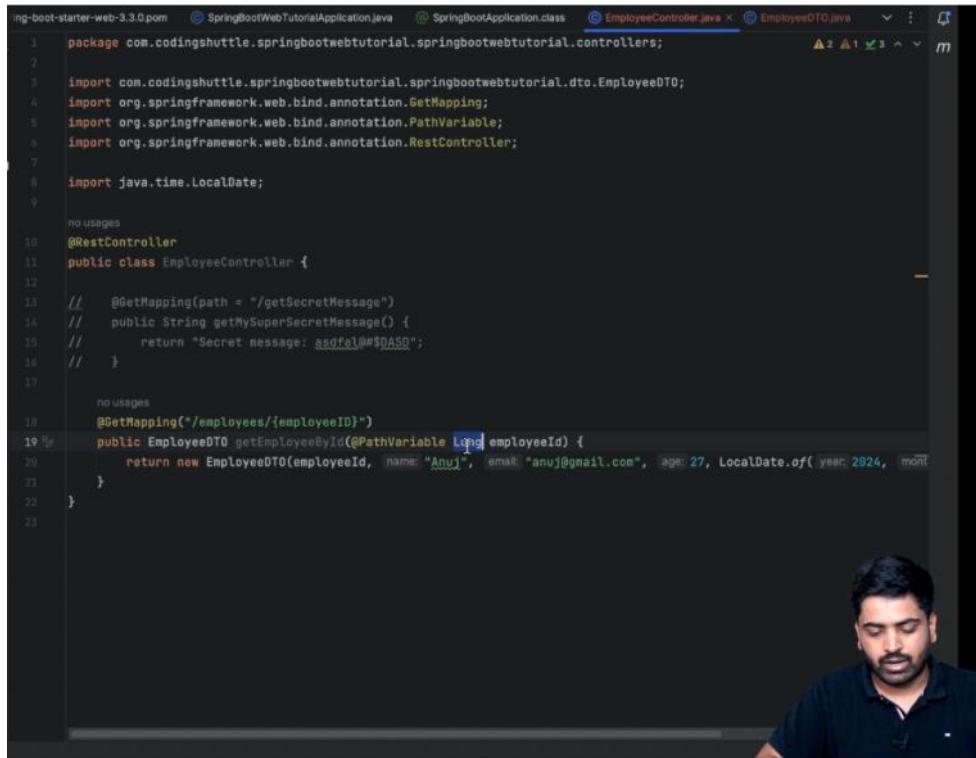
Use path variables when the parameter is an essential part of the URL path that identifies a resource.

`@RequestParam`

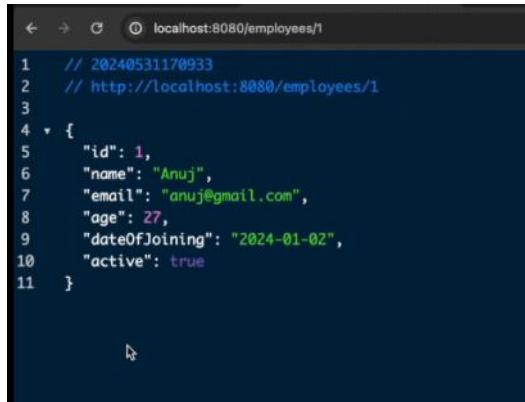
`/employees?id=123`

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 }
```



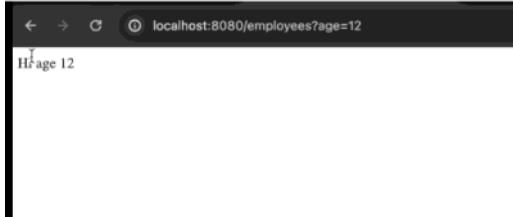
```
← → ⌂ localhost:8080/employees/1
1 // 20240511170933
2 // http://localhost:8080/employees/1
3
4 +
5     "id": 1,
6     "name": "Anuj",
7     "email": "anuj@gmail.com",
8     "age": 27,
9     "dateOfJoining": "2024-01-02",
10    "active": true
11 }
```

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;
}

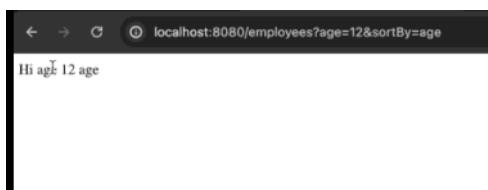
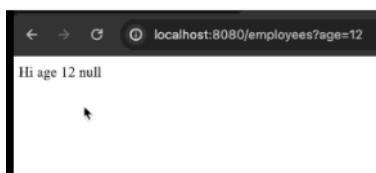
}
```



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 {
    I
    //  @GetMapping(path = "/getSecretMessage")
    //  public String getMySuperSecretMessage() {
    //      return "Secret message: asdfal@#$ASD0";
    //  }

    no usages
    @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:
    }

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

```
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

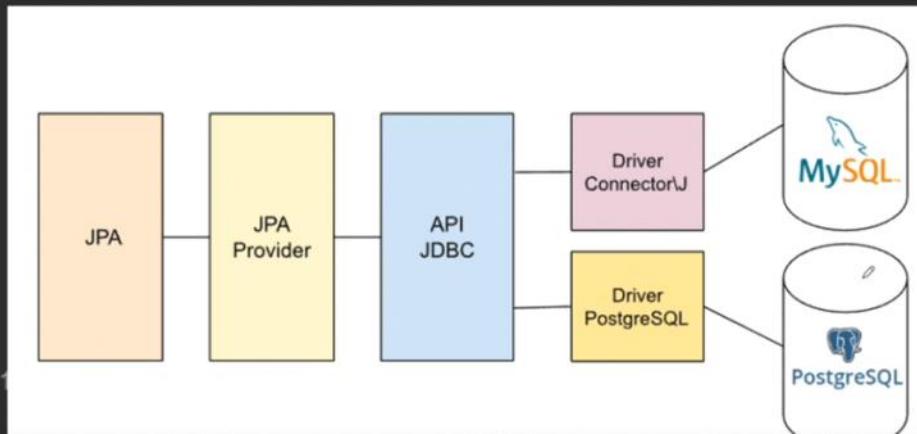
Pretty Raw Preview Visualize JSON

```
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.

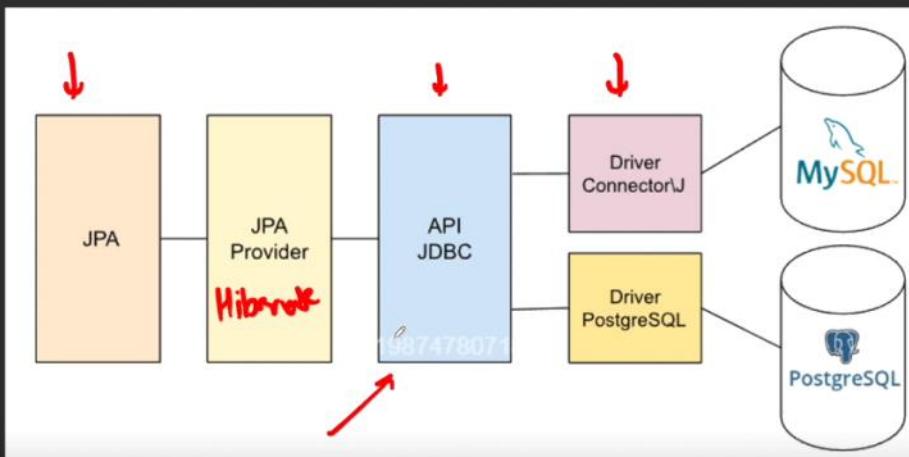
The Persistence layer and JPA Repository

Java Persistence API - JPA



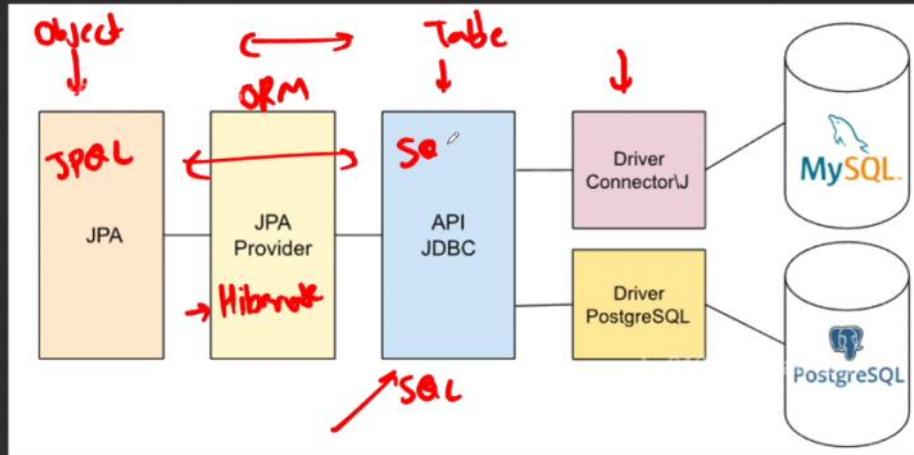
Java Persistence API - JPA

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



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     no usages
12     @Id
13     @GeneratedValue(strategy = GenerationType.AUTO)
14     private Long id;
15     no usages
16     private String name;
17     no usages
18     private String email;
19     no usages
20     private Integer age;
21     no usages
22     private LocalDate dateOfJoining;
23     no usages
24     private Boolean isActive;
25 }
```

```
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
no usage: Provided: EmployeeEntity
@GetMap Make 'getEmployeeById()' return 'com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities.EmployeeEnti More actions... ↴
public
    @PathVariable(name = "employeeId")
    Long id
    ret springbootwebtutorial.
}
```

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( other: null );
}
}
```

Returning list of all employees

```
30     no usages · new *
31
32     @GetMapping
33     public List<EmployeeEntity> getAllEmployees(@RequestParam(required = false, name = "inputAge") Integer age,
34                                                 @RequestParam(required = false) String sortBy) {
35         return employeeRepository.findAll();
36     }

```

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

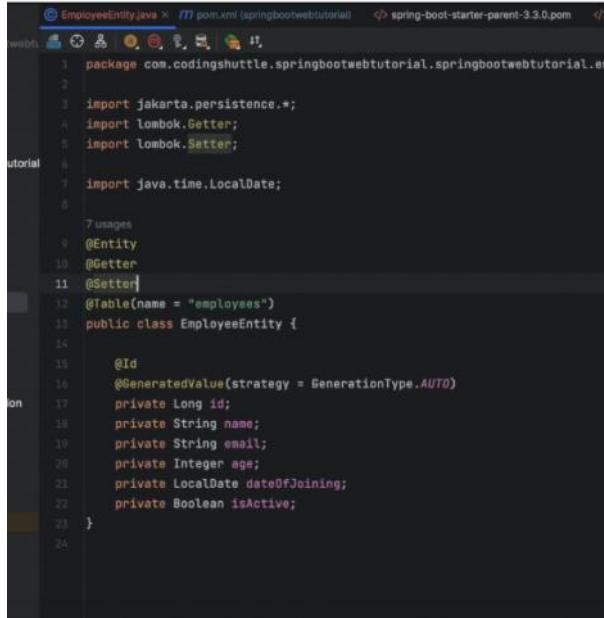
The screenshot shows the Postman application interface. At the top, there's a header bar with 'New', 'Import', 'POST http://localhost:8080/employees', and 'No environment'. On the left sidebar, there are sections for 'MappingTutorial' and 'Tutorial'. The main workspace shows a POST request to 'http://localhost:8080/employees'. The 'Body' tab is selected, showing raw JSON input:

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

Below the body, the response status is 'Status: 200 OK' with a 'Save as example' button. The response body is shown in 'Pretty' format as an empty object: {}.

Now add Lombok dependency

Now we can define getters and setters easily



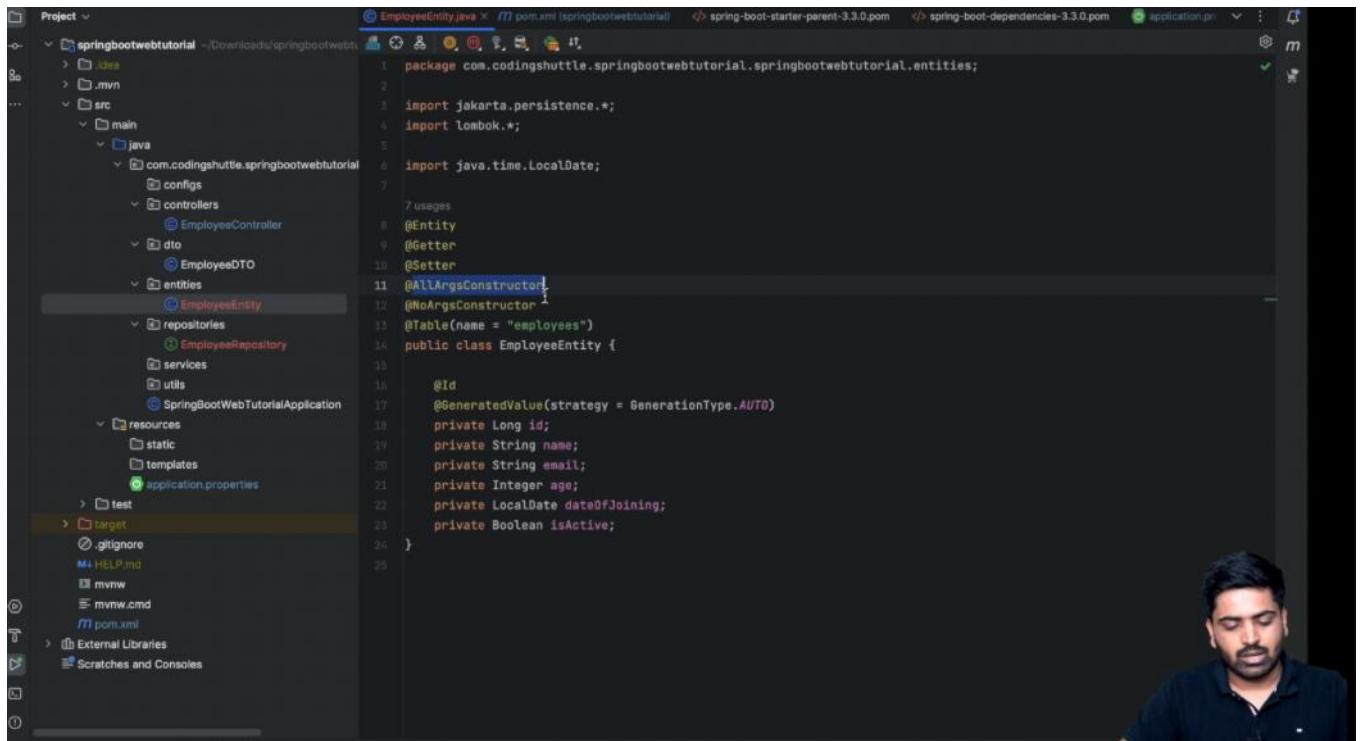
```
EmployeeEntity.java
package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.entities;

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 containing a single employee record:

```
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 containing two employee records:

```
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": "Dham",
13     "email": "dham@codingshuttle.com",
14     "age": 23,
15     "dateOfJoining": "2024-01-01",
16     "isActive": false
17   }
18 ]
```

Here data saving in in-memory database.