Name: Raj Prajapati

Institute: Tops Technologies

Subject: Spring Framework

Spring

Spring

**IOC Container Introduction:

Spring Framework's **Inversion of Control (IoC)** and its modules (spring-core.jar and spring-context.jar):

---

1. Containers

The **IoC Container** is the backbone of the Spring Framework, responsible for managing the lifecycle and configuration of beans (objects). It does this by creating, configuring, and assembling dependencies. There are two main types of containers:

- **BeanFactory**: A basic container that provides support for dependency injection and bean lifecycle management. It's lightweight and used in resource-constrained environments.
- **ApplicationContext**: A more advanced container that extends BeanFactory and provides additional features like event propagation, internationalization, and integration with other frameworks like AOP.

Key Role: The IoC container handles the entire object creation process and injects dependencies where needed.

---

2. Dependency Injection (DI)

Dependency Injection is a design pattern where the IoC container injects objects (dependencies) into other objects instead of creating them directly. Spring supports the following DI methods:

- **Constructor Injection**: Dependencies are provided through a class constructor.
- **Setter Injection**: Dependencies are set via public setter methods.
- **Field Injection**: Dependencies are directly injected into fields using annotations like @Autowired.

Key Advantage: It promotes loose coupling by allowing objects to depend on interfaces or abstractions rather than concrete implementations.

---

3. Auto Wiring

Autowiring simplifies dependency injection by letting the IoC container automatically wire (inject) relationships between beans. It reduces manual configuration. Spring provides several autowiring modes:

- **@Autowired**: Automatically resolves and injects dependencies based on type.
- **@Qualifier**: Specifies which bean to use when multiple candidates match the type.
- **@Primary**: Marks a bean as the default choice when multiple candidates exist.

Key Feature: Autowiring eliminates the need for explicit XML configuration, streamlining the setup process.

---

4. Lifecycle Management

Spring manages the lifecycle of beans, from their initialization to destruction. The container provides hooks for custom initialization and destruction methods:

- **Initialization Phase**: After the bean is instantiated, custom methods annotated with @PostConstruct (or declared in XML configuration) are called.
- **Destruction Phase**: When the container is shut down, methods annotated with @PreDestroy (or declared in XML) are executed.

Key Component: The IoC container ensures that beans are properly initialized, dependencies are injected, and resources are cleaned up efficiently.

---

5. Stereotypes

Stereotypes are annotations that help define the role of a class in the application. These annotations also act as metadata for Spring to register the class as a bean in the container:

- **@Component**: A general-purpose stereotype for any Spring-managed component.
- **@Service**: Specifically used for service-layer classes.
- **@Repository**: Indicates Data Access Object (DAO) classes.
- **@Controller**: Marks a class as a web controller in Spring MVC applications.
- **@RestController**: A combination of @Controller and @ResponseBody, used for RESTful APIs.

Key Benefit: Stereotypes make it easy to define and organize the different layers of an application by role.

Together, these elements make the **IoC Container** in the Spring Framework a powerful tool for managing the lifecycle, dependencies, and roles of objects in an application.

Q) Difference Between **BeanFactory** and **ApplicationContext**?

Ans: The **BeanFactory** and **ApplicationContext** are both types of IoC containers in Spring, but they differ in functionality and features.

Let me explain the Differences:

---

1. Basic Purpose

- **BeanFactory**:
    - It is the simpler and more basic container in Spring.
    - Its primary purpose is to manage the lifecycle of beans and handle dependency injection.
    - It lazily initializes beans, meaning beans are created only when they are requested.
- **ApplicationContext**:
    - It is an advanced container built on top of BeanFactory.
    - It provides additional enterprise-oriented features like event handling, internationalization, and application-layer-specific functionalities.
    - It eagerly initializes beans by default, creating them at startup unless explicitly configured otherwise.

---

2. Features

- **BeanFactory**:
    - Offers basic dependency injection.
    - Does not provide support for advanced features like events, AOP (Aspect-Oriented Programming), or declarative transaction management.

- Requires manual loading of the XML configuration file.

- **ApplicationContext**:
  - Includes everything BeanFactory offers but with additional features like:
    - Event propagation and listener mechanisms.
    - Built-in support for resolving message sources (internationalization).
    - Integration with Spring AOP and declarative transactions.
    - Annotation-based configuration and scanning (@Component, @Service, etc.).
  - Automatically detects and registers beans from annotated classes.

---

3. Use Cases

- **BeanFactory**:
  - Suitable for lightweight applications with simple requirements.
  - Preferred in memory-constrained environments where performance is critical.
- **ApplicationContext**:
  - Ideal for enterprise applications with complex requirements.
  - Recommended for most Spring applications due to its rich set of features.

---

4. Examples of Usage

- **BeanFactory**:

```
Resource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
MyBean myBean = factory.getBean(MyBean.class);
```

- **ApplicationContext**:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
MyBean myBean = context.getBean(MyBean.class);
```

Summary Table

Feature	BeanFactory	ApplicationContext
Initialization	Lazy (on request)	Eager (at startup)
Event Handling	Not supported	Supported
Internationalization	Not supported	Supported
AOP & Transactions	Limited	Fully supported
Annotation Support	Limited	Extensive
Use Case	Lightweight/simple apps	Enterprise/complex apps

In general, **ApplicationContext** is preferred for most real-world applications due to its extended functionality and developer convenience. However, understanding both is valuable depending on the context of your project.

❖

Q) Difference Between Initialization and instantiation of Object?

Ans: The difference between **initialization** and **instantiation** lies in their roles during the object creation process in programming:

1. Instantiation

- Instantiation refers to the act of **creating a new instance** (object) of a class.
- This happens when you use keywords like new in Java (or equivalent in other languages). Essentially, it allocates memory for the object and sets up the basic structure.
- **Example:**

```
Employee emp = new Employee(); // Instantiation
```

❖ Here, a new Employee object is created, and its default constructor is called.

2. Initialization

- Initialization is the process of **preparing the instantiated object** with specific values or state. This can involve setting its properties, calling methods, or any setup required before the object is fully usable.
- Initialization often happens after instantiation, either explicitly (using setters or methods) or implicitly (via constructors or frameworks like Spring).
- **Example:**

```
emp.setName("John Doe"); // Initialization
emp.setDepartment(new Department("HR"));
```

Key Differences:

Aspect	Instantiation	Initialization
Definition	Allocating memory and creating an object.	Setting up the object's internal state.
Purpose	To create a "shell" of an object.	To make the object fully operational.
Occurs When?	When new is called or frameworks instantiate.	After instantiation (e.g., via setters, constructors).

Example in Spring	Bean creation by Spring IoC container.	Property injection via @Autowired, @Value.

In simple terms:

- **Instantiation** is "creating the object."
- **Initialization** is "configuring the object."

Would you like to dive deeper into how Spring handles these processes during the lifecycle of a bean? Let me know!

❖

Q) How to inject runtime values into beans using annotation-based configurations?

Ans: You're absolutely right! In XML-based configuration, we can provide values for objects (beans) at runtime using <property> elements. The same can be achieved using annotation-based configurations by leveraging **@Value annotation**, **environment variables**, or **external property files**.

Let me guide you step-by-step:

---

1. Using @Value Annotation

The @Value annotation allows you to inject literal values or expressions into fields, setter methods, or constructor arguments.

Example:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Employee {
    @Value("John Doe") // Injecting runtime value
    private String name;

    @Value("#{department.name}") // Expression Language (SpEL)
    private String departmentName;

    public void displayInfo() {
        System.out.println("Employee Name: " + name);
        System.out.println("Department Name: " + departmentName);
    }
}
```

2. Using External Property Files

To inject values from a configuration file, you can use @PropertySource along with @Value.

Example:

Step 1: Create a application.properties file

```
Properties

employee.name=John Doe
employee.department=HR
```

Step 2: Load the properties file using @PropertySource

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {
}
```

Step 3: Inject values using @Value

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Employee {
    @Value("${employee.name}") // Referencing property key
    private String name;

    @Value("${employee.department}")
    private String departmentName;

    public void displayInfo() {
        System.out.println("Employee Name: " + name);
        System.out.println("Department Name: " + departmentName);
    }
}
```

3. Using Environment Variables

Spring allows injection of system environment variables using @Value. This is especially useful for runtime values like hostnames or credentials.

Example:

```java
@Component
public class ServerConfig {
    @Value("${JAVA_HOME}") // Injecting an environment variable
    private String javaHome;

    public void displayConfig() {
        System.out.println("JAVA_HOME: " + javaHome);
    }
}
```

---

4. Using @Bean with Arguments

Another approach is to define runtime values directly in the configuration class using the **@Bean** method.

Example:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public Employee employee() {
        Employee employee = new Employee();
        employee.setName("John Doe"); // Injecting runtime value
        employee.setDepartmentName("HR");
        return employee;
    }
}
```

These approaches allow you to inject runtime values into beans using annotation-based configurations.

❖

@Bean Explanation:

The **@Bean** annotation in Spring is a key part of the Java-based configuration approach. It is used to define and register a method as a Spring-managed bean within the **application context**. When a method is **annotated** with **@Bean**, Spring will call that method and manage the returned object as a bean.

Q) What Exactly Does @Bean Do?

1. **Defines a Bean**:
    - The method annotated with **@Bean** acts as a factory method for a bean. The object returned by this method will be registered as a bean in the application context.
    - **Example:**

```
@Bean
public Employee employee() {
    return new Employee();
}
```

2. **Singleton Scope by Default**:
    - Beans created with @Bean are singleton by default, meaning Spring creates only one instance of the bean per application context.
    - You can specify a different scope if needed:

```
@Bean
@Scope("prototype")
public Employee employee() {
    return new Employee();
}
```

3. **Explicit Definition of Beans**:
    - It allows you to explicitly define beans in the configuration class, giving you complete control over their lifecycle and initialization.
4. **Dependency Injection**:

- You can inject dependencies into beans using the method parameters or other Spring mechanisms.

Example:

```
@Bean
public Department department() {
    return new Department("HR");
}

@Bean
public Employee employee(Department department) {
    Employee emp = new Employee();
    emp.setDepartment(department);
    return emp;
}
```

Advantages of Using @Bean

- Eliminates the need for XML configuration and makes the configuration concise and readable.
- Allows direct control over the creation of beans and their dependencies.
- Works seamlessly with other annotation-based features like @Autowired.

Key Points

- The @Bean annotation is placed on methods within a @Configuration class.
- Each method should return the type of object you want as a bean in the application context.

❖

Q) What is the Order of initializing(creation of object) bean in Spring?

Ans: When Spring resolves dependencies and initializes beans in an annotation-based configuration, **the order of bean initialization is determined by dependency relationships**. Let's break it down using the example provided:

Example:

```
@Bean
public Department department() {
    return new Department("HR");
}

@Bean
public Employee employee(Department department) { // Dependency injected here
    Employee emp = new Employee();
    emp.setDepartment(department);
    return emp;
}
```

Process Explanation

1. **Dependency Detection**:
    - Spring examines the employee() bean method and notices that it has a parameter of type Department.
    - Spring realizes that the Employee bean depends on the Department bean.
2. **Resolution of Dependencies**:
    - Before creating the Employee bean, Spring checks to see if the Department bean is already created.
    - If the Department bean isn't created yet, Spring invokes the department() factory method first to create the Department bean.
3. **Creation of Dependent Bean**:
    - Once the Department bean is available, Spring proceeds to invoke the employee() factory method, passing the Department instance as a parameter.
    - This ensures that the dependency (Department) is ready when creating the Employee bean.

The Key Principle

Spring follows a **"dependency first" initialization order**. In other words, if a bean has dependencies, Spring ensures all the dependent beans are fully initialized before creating the main bean.

In this case:

- The department() method is called first to create the Department bean.
- Then, the employee() method is called to create the Employee bean, with the Department bean passed as an argument.

Additional Notes

- If there are **circular dependencies**, Spring uses techniques like proxying to break the loop. For example, Bean A depends on Bean B, and Bean B depends on Bean A.
- For beans with no explicit dependencies (e.g., not injected into another bean), the order of initialization may be arbitrary unless explicitly controlled.

❖

We've covered XML-based and annotation-based configurations, along with bean creation processes and dependency injection.

We can dive into:

- **Spring Bean Lifecycle**: Understand the steps from initialization to destruction, including hooks like @PostConstruct and @PreDestroy.
- **Scopes of Beans**: Learn about singleton, prototype, request, session, and other scopes.
- **Spring Profiles**: Manage environment-specific configurations with @Profile.
- **Spring AOP (Aspect-Oriented Programming)**: Explore cross-cutting concerns like logging and security using aspects.
- **Spring Data**: Dive into database interaction with Spring JDBC or JPA

Let's start with **Spring Bean Lifecycle**. Understanding the lifecycle gives you insight into how Spring manages beans, from creation to destruction.

1. Lifecycle Phases

Here are the key phases of a Spring bean's lifecycle:

- **Instantiation**: The bean is created by the IoC container.
- **Property Injection**: Dependencies are injected into the bean.
- **Custom Initialization**: Any custom init logic is executed.
- **Ready to Use**: The bean is fully initialized and ready to serve.
- **Destruction**: The bean is cleaned up when the container shuts down.

By using methods like @PostConstruct and @PreDestroy, or implementing interfaces like DisposableBean, you're overriding Spring's default lifecycle behavior for initializing and destroying beans.

Here's how it all ties together:

- **Default Methods**: Spring implicitly calls lifecycle methods such as **close()** or **shutdown()** if they exist in the bean. These are considered default behaviors.
- **Overriding Default Behavior**:
    - **Annotations (@PostConstruct and @PreDestroy)**: These provide explicit instructions for initialization and destruction.
    - **Lifecycle Interfaces (DisposableBean or InitializingBean)**: These allow you to implement specific methods for bean lifecycle management.
    - **Parameters in @Bean (initMethod and destroyMethod)**: These override or specify custom methods for lifecycle events.

Each approach ensures you have full control over bean initialization and cleanup logic, replacing or extending default methods.

---

2. Customization Hooks

Spring provides ways to inject custom logic during the lifecycle of a bean.

Using @PostConstruct and @PreDestroy Annotations

These annotations allow you to define custom logic during initialization and destruction.

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.stereotype.Component;

@Component
public class Employee {
    @PostConstruct
    public void init() {
        System.out.println("Employee bean is initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Employee bean is about to be destroyed");
    }
}
```

Using InitializingBean and DisposableBean Interfaces

Implementing these interfaces allows you to override specific lifecycle methods.

```java
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Employee implements InitializingBean, DisposableBean {
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Initialization logic in afterPropertiesSet");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("Cleanup logic in destroy");
    }
}
```

Using @Bean with initMethod and destroyMethod

When defining beans in @Configuration classes, you can specify initialization and destruction methods.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean(initMethod = "init", destroyMethod = "cleanup")
    public Employee employee() {
        return new Employee();
    }
}
```

---

3. Example Workflow

1. Spring container starts and creates beans.
2. Dependencies are injected.
3. Custom initialization methods (**@PostConstruct**, **initMethod**) are called.
4. Bean is used in the application.

5. When the container shuts down, destruction methods **(@PreDestroy, destroyMethod**) are called.

Q) When we use only @Bean without this just @Bean(initMethod = "init", destroyMethod = "cleanup”) then also it will run, so what is the need of using this parameters?

Ans: When using @Bean(initMethod = "init", destroyMethod = "cleanup"), you're explicitly instructing Spring to invoke the init and cleanup methods during the respective bean lifecycle phases. However, **if you do not specify these parameters, Spring will not automatically detect or execute lifecycle methods unless other mechanisms (like annotations or interfaces) are used**.

Here's a breakdown:

1. **Without Explicit Parameters in @Bean**: If you don't specify initMethod or destroyMethod, Spring won't automatically invoke init or cleanup methods unless:
    - The methods are annotated with @PostConstruct or @PreDestroy.
    - The bean class implements InitializingBean or DisposableBean interfaces.

Example with @PostConstruct and @PreDestroy:

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
public class Employee {
  @PostConstruct
  public void init() {
    System.out.println("Employee bean is initialized!");
  }
  @PreDestroy
  public void cleanup() {
    System.out.println("Employee bean is being destroyed!");
  }
```

```
}
```

2. **With Explicit Parameters in @Bean**: When you include initMethod and destroyMethod in the @Bean annotation, you're manually specifying the methods to invoke. This approach works even if the init and cleanup methods aren't annotated with @PostConstruct or @PreDestroy.

Example:

```
@Bean(initMethod = "init", destroyMethod = "cleanup")
public Employee employee() {
   return new Employee();
}
```

3. **Why Use initMethod and destroyMethod?**
    - Useful for cases where you cannot modify the bean class (e.g., third-party libraries).
    - Provides flexibility without requiring lifecycle-specific annotations in your bean code.

So, specifying initMethod and destroyMethod is necessary unless you're using one of the built-in detection mechanisms (like annotations or interfaces).

Q) If I am using @Postconstruct and @PreDestroy then is there any need of specify initMethod and destroyMethod parameters in the @Bean annotation?

Ans: When you're using annotations like @PostConstruct and @PreDestroy, there's usually **no need to specify initMethod and destroyMethod parameters** in the @Bean annotation because the annotated methods already handle initialization and destruction.

However, there are situations where passing parameters in the @Bean annotation might still be useful:

When to Use initMethod and destroyMethod Parameters:

1. **Legacy Code or External Classes**:
    - If you're working with a third-party library or an external class whose methods cannot be annotated (e.g., you can't modify the source code),

initMethod and destroyMethod allow you to manage lifecycle methods without modifying the bean class.

- **Example:**

```
@Bean(initMethod = "start", destroyMethod = "stop")
public ExternalService externalService() {
   return new ExternalService();
}
```

2. **Bean Classes Without Annotations**:
   - If the bean class isn't annotated with @PostConstruct or @PreDestroy, you can still specify lifecycle methods explicitly.
   - **Example:**

```
@Bean(initMethod = "initializeResources", destroyMethod = "cleanupResources")
public ResourceManager resourceManager() {
   return new ResourceManager();
}
```

3. **Override Default Behavior**:
   - Spring automatically calls methods named **close()** or **shutdown()** as default destroy methods. If you want to override this behavior or specify different lifecycle methods, you can use the initMethod and destroyMethod parameters.
4. **Simplify Configuration for Simple Beans**:
   - When you prefer to configure lifecycle methods in the Java configuration instead of annotating every class, you can centralize these details in your @Bean definition.

Conclusion:

If you're in control of your bean classes and can use @PostConstruct and @PreDestroy, it's more declarative and convenient. On the other hand, initMethod and destroyMethod are lifesavers for external libraries, legacy code, or cases where annotations aren't feasible.

**Bean Scopes:

Spring Framework provides several bean scopes to define the lifecycle and visibility of beans within the application context.

Here's a detailed explanation of each scope:

1. **Singleton (Default)**: A single instance of the bean is created and shared across the entire Spring IoC container. All requests for this bean return the same instance.
2. **Prototype**: A new instance of the bean is created every time it is requested from the container.
3. **Request**: A single instance of the bean is created for each HTTP request. This scope is valid only in a web-aware Spring ApplicationContext.
4. **Session**: A single instance of the bean is created for each HTTP session. This is also valid only in a web-aware Spring ApplicationContext.
5. **Application**: A single instance of the bean is created for the lifecycle of a ServletContext. This scope is valid in a web-aware Spring ApplicationContext.
6. **WebSocket**: A single instance of the bean is created for the lifecycle of a WebSocket. This is valid in a web-aware Spring ApplicationContext.

Each scope serves a specific purpose, allowing developers to control how beans are created and managed in different contexts.

1. Singleton Scope (Default)

- **Definition**: A single instance of the bean is created and shared across the entire Spring IoC container. All requests for this bean return the same instance.
- **Usage**:

```
@Bean
@Scope("singleton")
public MyBean mySingletonBean() {
    return new MyBean();
}
```

- **Key Points**:
  - Singleton beans are cached and reused.
  - They must be thread-safe in multi-threaded environments.

---

2. Prototype Scope

- **Definition**: A new instance of the bean is created every time it is requested from the container.
- **Usage**:

```
@Bean
@Scope("prototype")
public MyBean myPrototypeBean() {
    return new MyBean();
}
```

- **Key Points**:
  - Prototype beans are not managed by the container after creation.
  - Dependency injection into singleton beans requires careful handling (e.g., using ObjectFactory).

---

3. Request Scope

- **Definition**: A single instance of the bean is created for each HTTP request. This scope is valid only in a web-aware Spring ApplicationContext.
- **Usage**:

```
@Bean
@Scope("request")
public MyBean myRequestBean() {
    return new MyBean();
}
```

- **Key Points**:
  - Each HTTP request gets its own instance.
  - Using this scope in a non-web application will throw an exception.

---

4. Session Scope

- **Definition**: A single instance of the bean is created for each HTTP session. This scope is valid only in a web-aware Spring ApplicationContext.
- **Usage**:

```
@Bean
@Scope("session")
public MyBean mySessionBean() {
    return new MyBean();
}
```

- **Key Points**:
  - The bean instance is tied to the lifecycle of the HTTP session.
  - Proper cleanup is essential when the session expires.

---

5. Application Scope

- **Definition**: A single instance of the bean is created for the lifecycle of a ServletContext. This scope is valid in a web-aware Spring ApplicationContext.
- **Usage**:

```
@Bean
@Scope("application")
public MyBean myApplicationBean() {
    return new MyBean();
}
```

- **Key Points**:
  - The bean instance is shared across the entire application.
  - Misusing this scope in a non-web application will cause errors.

---

6. WebSocket Scope

- **Definition**: A single instance of the bean is created for the lifecycle of a WebSocket. This scope is valid in a web-aware Spring ApplicationContext.
- **Usage**:

```
@Bean
@Scope("websocket")
public MyBean myWebSocketBean() {
        return new MyBean();
}
```

- **Key Points**:
    - Each WebSocket connection gets its own instance.
    - This scope is specific to WebSocket applications.

Additional Notes

- **Custom Scopes**: You can define your own custom scopes if the default ones don't meet your requirements.
- **Thread Scope**: A thread scope is available but not registered by default. It creates a bean instance per thread.

Spring's bean scopes allow developers to control the lifecycle and visibility of beans effectively.

❖

**Spring Profiles:

- Spring Profiles are a powerful feature in the Spring Framework that allow developers to manage environment-specific configurations. They enable applications to adapt seamlessly to different environments, such as development, testing, and production.
- Spring Profiles simplify this process by enabling you to define specific beans and configurations for each environment.

Here's a detailed explanation:

Q) What Are Spring Profiles?

Ans: Spring Profiles provide a way to segregate parts of your application configuration and make them available only in certain environments. This is particularly useful for managing environment-specific beans, properties, or settings.

****Key Features of Spring Profiles**

1. **Profile Declaration**: Profiles are declared using the @Profile annotation or in configuration files, allowing you to associate beans with specific profiles.
2. **Environment-Specific Configuration**: You can create distinct configurations for environments like dev, test, and prod, ensuring that your application behaves appropriately in each environment.
3. **Activation**: Profiles can be activated via various methods, such as:
    - Programmatically using ConfigurableEnvironment in code.
    - Environment variables or system properties (spring.profiles.active).
    - Configuration files like application.properties or application.yml.

****How to Use Spring Profiles**

1. Using @Profile Annotation

Spring's **@Profile** annotation is the most common way to define beans for specific profiles.

For example:

```java
@Component
@Profile("dev")
public class DevDataSourceConfig {
    // Configuration for the development environment
}
```

- **Behavior**: The bean annotated with @Profile("dev") will only be instantiated when the dev profile is active.
- The bean will only be active when the dev profile is enabled.
- **Logical Operators:**
- Spring supports logical operators like ! (NOT) and & (AND) to define more specific activation conditions and to Combine Profiles:

```java
@Component
@Profile("!prod & dev")
public class DevOnlyConfig {
    // Active only in development and not in production
}
```

2. Declaring Profiles in XML

Profiles can also be configured in XML using the <beans> tag:

<beans profile="dev">

  <bean id="devDataSourceConfig" class="com.example.DevDataSourceConfig"/>

</beans>

3. Activating Profiles

Profiles can be activated in several ways:

**Programmatically Activating Profiles

- Activating profiles programmatically allows greater control, especially in scenarios where you need dynamic configurations based on runtime conditions. Here's an expanded explanation and example of how it works:
- You can use the **ConfigurableApplicationContext** interface to manually set active profiles before refreshing the application context. This is particularly useful when building applications that require environment-specific behavior to be decided at runtime.

Detailed Example

Here's the breakdown of your provided code:

```
ConfigurableApplicationContext context = new AnnotationConfigApplicationContext

// Setting the active profile to "dev"
context.getEnvironment().setActiveProfiles("dev");

// Registering the configuration class
context.register(AppConfig.class);

// Refreshing the context to apply the configuration and activate the profile
context.refresh();
```

Explanation

- **Step 1:** A new AnnotationConfigApplicationContext object is created, which represents the application context using Java-based configuration.
- **Step 2:** The active profile is set to "dev" using setActiveProfiles. At this point, only beans annotated with @Profile("dev") will be loaded when the context is refreshed.
- **Step 3:** The register method links the desired configuration class (e.g., AppConfig) to the application context.
- **Step 4:** Calling refresh finalizes the configuration, ensuring that only the relevant beans and settings tied to the active profile are applied.

Why Use Programmatic Activation?

- **Runtime Flexibility:** Enables profiles to be activated dynamically based on system variables, user inputs, or other runtime conditions.
- **Fine-grained Control:** Allows conditional logic to determine which profile should be activated.
- **Testing Convenience:** Simplifies switching between profiles during unit testing or debugging sessions.

****Using application.properties:**

```
Properties

spring.profiles.active=dev
```

**Using Command Line:

```
Bash

java -Dspring.profiles.active=dev -jar app.jar
```

Default Profile

If no profile is explicitly activated, Spring uses the default profile. You can customize the default profile using:

```
Properties

spring.profiles.default=none
```

Combining Profiles

You can include additional profiles using the **spring.profiles.include** property:

```
Properties

spring.profiles.include=common,local
```

This adds profiles on top of those activated by **spring.profiles.active**.

Benefits of Spring Profiles

- **Environment-Specific Configurations**: Easily manage configurations for development, testing, and production environments.
- **Flexibility**: Activate profiles dynamically based on the environment.
- **Maintainability**: Keep configurations organized and reduce complexity.

Spring Profiles are a core feature for building adaptable and environment-aware applications.

**Spring AOP Module:

Spring AOP (Aspect-Oriented Programming) is a powerful module within the Spring framework that enables developers to separate cross-cutting concerns such as logging, security, transaction management, and exception handling from the business logic. It allows you to modularize concerns that affect multiple parts of an application without cluttering the core logic.

- In Spring AOP**, ****cross-cutting concerns**** refer to functionalities that are used across different parts of an application but are **not** directly related to its core business logic. These concerns **"cut across"** multiple components, hence the name.
- **For example**, imagine you have multiple service methods in an application. You may need to **log** information, **handle security**, or **manage transactions** across many of these methods. Instead of writing the same code in every method, you define an aspect that **applies the logic consistently** wherever needed.

Key Concepts of Spring AOP

1. **Aspect**: A module that encapsulates the cross-cutting concerns (e.g., logging, security).
2. **Advice**: The action performed by an aspect at a particular join point (e.g., before, after, around).
3. **Pointcut**: A predicate that determines whether an advice should be executed for a given method execution.
4. **Join Point**: A point in the application where an aspect can be executed (e.g., method execution).
5. **Weaving**: The process of applying aspects to target objects to create advised objects.

Types of Advice in Spring AOP

- **Before Advice**: Runs before the method execution.
- **After Returning Advice**: Runs after the method executes successfully.
- **After Throwing Advice**: Runs if the method throws an exception.
- **After Advice**: Runs after the method execution (regardless of success or failure).

- **Around Advice**: Runs before and after the method execution.

Example: Logging Aspect

Imagine you want to log every method execution in a service.

Step 1: Add Dependencies (Spring Boot Starter AOP)

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

Step 2: Create an Aspect Class

```
import org.aspectj.lang.annotation.*;
import org.aspectj.lang.JoinPoint;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect
{
        @Before("execution(* com.example.service.*.*(..))")
        public void beforeAdvice(JoinPoint joinPoint)
        {
            System.out.println("Executing method: " +
            joinPoint.getSignature().getName());
        }
        @AfterReturning("execution(* com.example.service.*.*(..))")
        public void afterReturningAdvice(JoinPoint joinPoint)
        {
             System.out.println( "Completed method: " +
            joinPoint.getSignature().getName());
        }
```

}

Step 3: Enable AspectJ Support

Ensure that Spring recognizes aspects by adding @EnableAspectJAutoProxy in your configuration.

```
import org.springframework.context.annotation.*;

@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
}
```

How It Works

- @**Before** executes before any method within **com.example.service** is invoked.
- @**AfterReturning** executes after the method successfully returns a result.
- Spring automatically applies these aspects to matching methods without modifying business logic.

Spring AOP is a great way to improve code maintainability and keep concerns like logging, security, and transactions modular.

JoinPoint and PointCut

JoinPoint and **Pointcut** are fundamental concepts in Spring AOP that help define where and when advice should be applied in an application.

---

1. JoinPoint – The Execution Context

A **JoinPoint** represents a specific point in the execution of a program where an aspect can be applied. It could be:

- A method execution
- An object initialization
- An exception being thrown

Key Properties of JoinPoint

- Provides access to method details (e.g., name, arguments).

- Helps retrieve the class name and method signature.
- Allows us to customize aspect behavior dynamically.

Example: Using JoinPoint in Advice

```
@Before("execution(* com.example.service.MyService.processData(..))")
public void beforeAdvice(JoinPoint joinPoint) {
   System.out.println("Executing method: " + joinPoint.getSignature().getName());
   System.out.println("Target Class: " + joinPoint.getTarget().getClass().getName());
   System.out.println("Arguments: " + Arrays.toString(joinPoint.getArgs()));
}
```

Explanation

- **joinPoint.getSignature().getName()** → Retrieves the method name.
- **joinPoint.getTarget().getClass().getName()** → Gets the class where the method is executed.
- **joinPoint.getArgs()** → Fetches method arguments as an array.

---

2. Pointcut – The Filtering Mechanism

A **Pointcut** defines the criteria that determine which methods should be intercepted by an aspect. In simple terms, it's a filter that selects methods where advice should run.

Types of Pointcut Expressions

Expression	Meaning
execution(* com.example.service.MyService.*(..))	Matches all methods in MyService
execution(* com.example.service.MyService.processData(..))	Matches only processData(..)
within(com.example.service..*)	Matches all methods inside com.example.service package
args(String)	Matches methods that accept a String parameter

Example: Custom Pointcut

```
@Pointcut("execution(* com.example.service.MyService.processData(..))")
public void myPointcut() {}

@Before("myPointcut()")
public void beforeAdvice(JoinPoint joinPoint) {
    System.out.println("Before executing: " + joinPoint.getSignature().getName(
}
```

Explanation

- The @**Pointcut** annotation defines a reusable pointcut named **myPointcut().**
- The @**Before("myPointcut()")** advice uses the pointcut instead of writing the execution expression again.

Difference Between JoinPoint & Pointcut

Feature	JoinPoint	Pointcut
What it represents	A specific execution point	A filtering mechanism for methods
Usage	Provides method details like name, arguments	Defines where aspects should apply
Example	joinPoint.getSignature().getName()	"execution(* com.example.service.MyService.*(..))"

Final Thoughts

- **JoinPoint** helps capture runtime details of intercepted methods.
- **Pointcut** determines **which methods** should be intercepted.
- Together, they make Spring AOP flexible and modular.

❖

Example With SpringBoot:

Here's a simple Spring AOP example that demonstrates logging before and after a method execution.

Steps to Implement Spring AOP

1. Add **Spring Boot Starter AOP** dependency.
2. Create a service class with a method.
3. Define an aspect class to log method execution.
4. Enable AOP in the configuration.

---

1. Add Spring AOP Dependency (for Maven projects)

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

---

2. Create a Simple Service Class

```
package com.example.service;

import org.springframework.stereotype.Service;

@Service
public class MyService {

    public String processData(String input) {
        System.out.println("Processing: " + input);
        return "Processed Data: " + input.toUpperCase();
    }
}
```

---

3. Create an Aspect for Logging

```
package com.example.aspect;

import org.aspectj.lang.annotation.*;
```

```
import org.aspectj.lang.JoinPoint;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

   @Before("execution(* com.example.service.MyService.processData(..))")
   public void beforeAdvice(JoinPoint joinPoint)
   {
            System.out.println("Before executing method: " + joinPoint.getSignature().getName());
   }

   @AfterReturning(pointcut =
"execution(*com.example.service.MyService.processData(..))", returning = "result")
   public void afterReturningAdvice(JoinPoint joinPoint, Object result) {
           System.out.println("After executing method: " + joinPoint.getSignature().getName() +
          ", Result: " + result);
   }
}
```

---

4. Enable AspectJ Support in Configuration

```
package com.example.config;

import org.springframework.context.annotation.*;

@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
}
```

---

5. Run the Application

In a Spring Boot application, simply annotate your main class with @SpringBootApplication and run it.

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.example.service.MyService;

@SpringBootApplication
public class AopExampleApplication {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(AopExampleApplicati

        MyService myService = context.getBean(MyService.class);
        myService.processData("Hello AOP");
    }
}
```

```
n.run(AopExampleApplication.class, args);
```

Expected Output

```
Before executing method: processData
Processing: Hello AOP
After executing method: processData, Result: Processed Data: HELLO AOP
```

This is a basic demonstration of AOP in Spring. The aspect intercepts the method execution and logs messages **before** and **after** processing.

Explanation Of Above Example:

Let's break down each part of your **LoggingAspect** class to understand how it works.

Package Declaration

package com.example.aspect;

- Defines the package com.example.aspect, which logically groups aspect-related classes.

Imports

import org.aspectj.lang.annotation.;*

import org.aspectj.lang.JoinPoint;

import org.springframework.stereotype.Component;

- **org.aspectj.lang.annotation.*** : Imports AOP-related annotations like @Aspect, @Before, and @AfterReturning.
- **org.aspectj.lang.JoinPoint**: Provides access to method details during execution.
- **org.springframework.stereotype.Component**: Marks the class as a Spring-managed component.

Class Definition

@Aspect

@Component

public class *LoggingAspect {*

- @**Aspect**: Declares this class as an aspect, enabling AOP functionality.
- @**Component**: Registers this class as a Spring bean so that the framework manages it.

Before Advice

```
@Before("execution(* com.example.service.MyService.processData(..))")
public void beforeAdvice(JoinPoint joinPoint) {
        System.out.println("Before executing method: " +
    joinPoint.getSignature().getName());
}
```

- @**Before**: Runs **before processData(..)** is executed.

- **"execution(* com.example.service.MyService.processData(..))"**:
    - Targets any method named processData inside MyService.
    - The **(..)** part means it works with any number of parameters.
- **JoinPoint joinPoint**: Captures method execution details.
- **joinPoint.getSignature().getName()**: Retrieves the name of the method being executed.
- The method prints **"Before executing method: processData"** before execution.

After Returning Advice

```
@AfterReturning(pointcut = "execution(* com.example.service.MyService.processData(..))", returning = "result")
public void afterReturningAdvice(JoinPoint joinPoint, Object result) {
    System.out.println("After executing method: " + joinPoint.getSignature().getName() + ", Result: " + result);
}
```

- **@AfterReturning**: Runs **after** the method completes successfully.
- **pointcut = "execution(* com.example.service.MyService.processData(..))"**:
    - Applies to **processData(..)** in MyService.
- **returning = "result"**:
    - Captures the **return value** from **processData(..)**.
- **Object result**: Holds the returned value.
- Prints **"After executing method: processData, Result: Processed Data: XYZ"** after execution.

Summary

1. **Before Execution** → Logs the method name.
2. **After Successful Execution** → Logs method name and returned value.
3. Helps in debugging, auditing, and performance monitoring.

Spring Data:

- Spring Data is a powerful module in the Spring framework designed to simplify data access and persistence in applications. It provides a unified way to interact with databases, eliminating boilerplate code and enabling developers to focus on business logic.

Key Features of Spring Data

1. **Simplified Data Access**: Reduces the amount of code needed for database operations.
2. **Repository Abstraction**: Uses CrudRepository and JpaRepository for easy CRUD operations.
3. **Supports Multiple Data Stores**: Works with relational databases (JPA, JDBC) and NoSQL databases (MongoDB, Redis, Cassandra).
4. **Query Methods**: Allows you to define queries using method names (findByName() instead of writing SQL).
5. **Paging & Sorting**: Supports pagination and sorting with minimal effort.
6. **Transaction Management**: Simplifies transaction handling for databases.

Core Components of Spring Data

1. Spring Data JPA (For Relational Databases)

- Simplifies database interactions using JPA and Hibernate.
- Uses **repositories** (JpaRepository) to handle CRUD operations.

Example:

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);
}
```

This auto-generates a query for retrieving users by name.

---

2. Spring Data JDBC (Lightweight Alternative to JPA)

- Uses repositories without complex ORM features.
- Works well for simple queries with direct database interactions.

Example:

```
@Repository
public class UserRepository {
    private final JdbcTemplate jdbcTemplate;

    public UserRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public List<User> getAllUsers() {
        return jdbcTemplate.query("SELECT * FROM users", new BeanPropertyRowMapp
    }
}
```

```
new BeanPropertyRowMapper<>(User.class));
```

---

3. Spring Data MongoDB (For NoSQL Databases)

- Supports MongoDB with document-based storage.
- Uses MongoRepository for easy data access.

Example:

```
public interface ProductRepository extends MongoRepository<Product, String> {
    List<Product> findByCategory(String category);
}
```

Automatically generates a query to find products by category.

---

4. Paging and Sorting Support

Spring Data provides an easy way to paginate results. Example:

```
Page<User> users = userRepository.findAll(PageRequest.of(0, 10,
Sort.by("name").ascending()));
```

- Fetches **10 users per page** sorted by name in ascending order.

Benefits of Spring Data

✅ **Reduces Boilerplate Code**: No need to manually write SQL queries.
✅ **Works with Different Databases**: Supports relational and NoSQL data stores.
✅ **Built-in Query Mechanism**: Automatically generates queries from method names.
✅ **Transaction Handling**: Provides seamless transaction management.

❖

Instructions While Using Spring Data Module:

When using **Spring Data**, here are some key instructions and best practices to follow to ensure smooth implementation and optimal performance.

1. Choosing the Right Spring Data Module

- If working with **relational databases (MySQL, PostgreSQL, etc.)**, use **Spring Data JPA**.
- If interacting with **NoSQL databases (MongoDB, Redis, etc.)**, use the corresponding **Spring Data NoSQL** module.
- If you prefer a lightweight alternative without JPA's complexity, use **Spring Data JDBC**.

2. Setting Up Dependencies

- Ensure you add the correct dependencies in your pom.xml (Maven) or build.gradle (Gradle).
- **Example** (Spring Data JPA):

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

3. Configuring the Database Connection

- Always configure the correct database **URL, username, password** in application.properties or application.yml.

```
Properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
```

- Use **environment variables** for security instead of hardcoding credentials.

4. Defining Entities Correctly (For Spring Data JPA)

- Always annotate model classes with @Entity, @Table, @Id, @GeneratedValue, etc.
- **Example**:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;
}
```

5. Using Repository Interfaces Effectively

- Extend JpaRepository or MongoRepository to get built-in CRUD methods.

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);
}
```

- Avoid writing unnecessary queries manually when Spring Data can generate them automatically.

6. Handling Transactions Properly

- Use @Transactional in service methods when making multiple database operations.

```
@Service
public class UserService {
    @Transactional
    public void updateUser(Long id, String newName) {
        User user = userRepository.findById(id).orElseThrow();
        user.setName(newName);
        userRepository.save(user);
    }
}
```

- Use **transaction rollback** strategies for error handling.

7. Managing Performance

- Use **pagination & sorting** for large datasets:

```
Page<User> users = userRepository.findAll(PageRequest.of(0, 10,
Sort.by("name").ascending()));
```

- **Avoid fetching unnecessary data**; select only required columns using **DTO projections**.

8. Securing Data Access

- Implement **custom query filtering** to restrict user access to certain data.
- Use **Spring Security** with Spring Data to apply role-based access control.

Final Thoughts

Spring Data makes database interactions **efficient** and **concise**, but following best practices ensures performance, security, and maintainability.

Spring Data JPA with some powerful features:

1. Custom Query Methods

In addition to autogenerated queries, you can define custom queries using @**Query**.

```
@Query("SELECT u FROM User u WHERE u.name LIKE %:keyword%")
List<User> searchByName(@Param("keyword") String keyword);
```

✅ This allows flexible searching while avoiding complex method naming.

---

2. Native Queries

If you need full control over SQL, use **native queries**.

```
@Query(value = "SELECT * FROM users WHERE name = :name", nativeQuery
= true)
List<User> findByNameNative(@Param("name") String name);
```

✅ Ideal for optimizing database performance!

---

3. Specification API for Dynamic Queries

For complex filtering, Spring Data provides the **Specification API**.

```
public class UserSpecification {
  public static Specification<User> hasName(String name) {
    return (root, query, cb) -> cb.equal(root.get("name"), name);
  }
}
```

✅ Useful when dealing with **dynamic search criteria**.

---

4. Soft Deletes with Logical Deletion

Instead of physically removing data, use a **soft delete approach** by adding a deleted flag.

```
@Column(name = "deleted", nullable = false)
private boolean deleted = false;
```

Then filter records accordingly:

```
@Query("SELECT u FROM User u WHERE u.deleted = false")
List<User> findActiveUsers();
```

✅ Helps maintain **audit trails** while preventing accidental deletions.

---

5. Auditing for Automatic Record Tracking

Spring Data provides built-in auditing features using @CreatedDate and @LastModifiedDate.

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class User {
    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;
}
```

✅ Automatically tracks **creation & update timestamps**!

Using **camelCase** for data members in an entity can lead to certain issues, especially when working with databases. Here are some key considerations:

1. Database Column Name Mismatch

- In most databases, column names are case-insensitive, and they are typically stored in **lowercase** or **snake_case** format (first_name).
- If your entity fields use camelCase (firstName), some ORM frameworks may **automatically convert** them to lowercase or snake_case.
- **Example** issue:

```
@Column(name = "firstName")
private String firstName;
```

If the database expects first_name, this can lead to **column not found** errors.

---

2. Inconsistent Naming Across Layers

- Java follows camelCase for variable names (userName), but databases often use snake_case (user_name).
- If auto-mapping isn't properly handled, queries may fail or require manual configuration.

---

3. JSON Serialization/Deserialization Issues

- When working with REST APIs, serialization tools like **Jackson** can automatically convert camelCase to **snake_case** unless properly configured.
- **Example**:

```
{
  "userName": "John Doe"
}
```

If the expected format is user_name, data parsing errors may occur.

- **Solution:**

```
@JsonProperty("user_name")
private String userName;
```

---

4. JPA/Hibernate Naming Strategy

- Hibernate provides different naming strategies that affect how camelCase field names are converted to column names.
- **Example**: **physical_naming_strategy=SpringPhysicalNamingStrategy**
    - Converts **firstName** → **first_name** automatically.
- If a different strategy is used, it may **not convert names correctly**, leading to errors.

---

5. Case Sensitivity Issues in Queries

- Some databases like **PostgreSQL** treat column names as **case-sensitive** when enclosed in double quotes.
- **Example issue**:

```
SELECT * FROM users WHERE "firstName" = 'John';
```

❖ If the database stores first_name, this query will fail.

Best Practices to Avoid Issues

✔ **Explicitly define column names using @Column(name = "first_name").**
✔ **Configure JSON serialization with @JsonProperty** if needed.
✔ **Use a proper Hibernate naming strategy to match the database format.**
✔ **Ensure naming conventions remain consistent across layers (Entity ↔ Database ↔ API response).**

❖

Q) How to Handle this in SpringBoot?

Ans: Here's a practical example showing how to properly handle camelCase naming conventions in a Spring Boot application.

---

1. Define Entity with Explicit Column Mapping

To avoid mismatches between Java field names and database column names, **always specify the exact column name** using **@Column(name = "column_name")**.

```
@Entity
@Table(name = "users")
public class User {

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

    @Column(name = "first_name")  // Explicit mapping to snake_case
    private String firstName;

    @Column(name = "last_name")
    private String lastName;
}
```

✅ **Ensures Spring maps firstName and lastName to first_name and last_name.**

---

2. Handling JSON Serialization (Snake Case)

Spring Boot uses **camelCase by default** for JSON responses. If your API needs **snake_case**, configure Jackson.

```
@JsonProperty("first_name")
private String firstName;
```

Or globally in **application.yml**:

```yaml
spring.jackson.property-naming-strategy: SNAKE_CASE
```

✅ **Converts JSON keys automatically (e.g., firstName → first_name).**

---

3. Configure Hibernate Naming Strategy

Spring Boot **automatically converts camelCase to snake_case**, but you can explicitly set naming strategies.

In **application.properties:**

spring.jpa.hibernate.naming.physical- strategy=
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

Or, **Spring's default naming strategy** (SpringPhysicalNamingStrategy) will **auto-convert**:

```properties
firstName → first_name
lastName → last_name
```

✅ **Prevents case-sensitive mismatches in PostgreSQL and other databases.**

---

4. Query Issues with Case Sensitivity

Some databases (like PostgreSQL) treat column names as **case-sensitive** if enclosed in quotes.

⚠️ **Problem:**

```
SELECT * FROM users WHERE "firstName" = 'John'; -- Error!
```

✅ **Solution: Use lower-case column names in queries**

```
SELECT * FROM users WHERE first_name = 'John';
```

✅ **Ensures compatibility across different DBMS.**

Final Thoughts

- ✔ **Always explicitly define column names using @Column(name = "snake_case").**
- ✔ **Use @JsonProperty or SNAKE_CASE naming strategy for JSON responses.**
- ✔ **Check Hibernate naming strategies to match database conventions.**
- ✔ **Write queries using lowercase column names to avoid case-sensitivity issues.**

❖

*SpringBoot Crud With These CamelCase Notations:

Awesome! Let's go step by step and implement an **end-to-end Spring Boot CRUD (Create, Read, Update, Delete) application** while following best practices for **camelCase naming convention** and database compatibility.

---

1. Setup Spring Boot Project

Ensure you have the correct dependencies in pom.xml:

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

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

<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

✅ **Includes Spring Boot Starter JPA, Web, and MySQL Driver.**

---

2. Configure Database Connection

Add database credentials in application.**properties**:

```
Properties

spring.datasource.url=jdbc:mysql://localhost:3306/userdb
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jackson.property-naming-strategy=SNAKE_CASE
```

✅ **Automatically converts JSON properties from camelCase to snake_case.**

3. Define Entity with Proper Naming

```
@Entity
@Table(name = "users")
public class User {

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

    @Column(name = "first_name")
    @JsonProperty("first_name")
    private String firstName;

    @Column(name = "last_name")
    @JsonProperty("last_name")
    private String lastName;
}
```

🔷 **Naming Convention Fixes**

✔ **@Column(name = "first_name")** ensures correct mapping.

✔ **@JsonProperty("first_name")** makes JSON responses compatible.

---

4. Create Repository Interface

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByFirstName(String firstName);
}
```

✅ **Provides built-in CRUD operations without writing SQL queries.**

---

5. Implement Service Layer

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

```
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    public User updateUser(Long id, User updatedUser) {
        User user = userRepository.findById(id).orElseThrow();
        user.setFirstName(updatedUser.getFirstName());
        user.setLastName(updatedUser.getLastName());
        return userRepository.save(user);
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

✅ **Manages CRUD logic inside a structured service layer.**

---

6. Create REST Controller

```
@RestController
@RequestMapping("/users")
public class UserController {
```

```
@Autowired
private UserService userService;

@GetMapping
public List<User> getAllUsers() {
   return userService.getAllUsers();
}

@PostMapping
public User createUser(@RequestBody User user) {
        return userService.createUser(user);
}

@GetMapping("/{id}")
public ResponseEntity<User> getUserById(@PathVariable Long id) {
            return userService.getUserById(id)
                .map(ResponseEntity::ok)
                .orElseGet(() -> ResponseEntity.notFound().build());
}

@PutMapping("/{id}")
public User updateUser(@PathVariable Long id, @RequestBody User user) {
   return userService.updateUser(id, user);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
```

```
            userService.deleteUser(id);
            return ResponseEntity.noContent().build();
        }
}
```

🔷 **Endpoints**

✔ **GET /users** → Fetch all users
✔ **POST /users** → Create a new user
✔ **GET /users/{id}** → Fetch user by ID
✔ **PUT /users/{id}** → Update user details
✔ **DELETE /users/{id}** → Delete user

✅ **Final Output Example (JSON)**

Request: **GET /users**

```Json
[
    {
        "id": 1,
        "first_name": "John",
        "last_name": "Doe"
    }
]
```

❖

Spring

**Spring MVC Module – Detailed Explanation with Examples

- Spring MVC (Model-View-Controller) is a powerful module in the **Spring Framework** designed for building web applications. It follows the **MVC architecture**, which separates concerns into **Model (data), View (UI), and Controller (business logic)** to make web applications more maintainable and scalable.

---

1. Architecture Overview

Spring MVC works based on the following key components:

Component	Role
DispatcherServlet	Front controller that routes HTTP requests.
Controller	Handles user input and processes requests.
Model	Represents the application's data (DTOs, Entities).
View	Defines the user interface (JSP, Thymeleaf, etc.).

---

2. How Spring MVC Works?

1. A **user sends an HTTP request** (e.g., **GET /home**).
2. The request reaches **DispatcherServlet** (front controller).
3. DispatcherServlet forwards the request to the appropriate **Controller**.
4. The Controller interacts with the **Model** (database/service).
5. The Controller returns a response to the **View**.
6. The View renders the final output and returns it to the user.

---

3. Setting Up a Spring MVC Project

First, ensure you have the correct dependencies in your **pom.xml**:

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

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

✅ Adds **Spring Boot Web** (for MVC) and **Thymeleaf** (for views).

---

4. Creating a Controller

A simple Spring MVC **Controller** handles user requests:

```java
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/home")
public class HomeController {

    @GetMapping
    public String homePage(Model model) {
        model.addAttribute("message", "Welcome to Spring MVC!");
        return "home"; // View name
    }
}
```

Explanation

✔ @Controller marks this class as an MVC controller.
✔ @RequestMapping("/home") maps requests to /home.
✔ @GetMapping handles GET /home requests.
✔ Model model passes data to the **View** (home.html).

5. Creating a View (Thymeleaf)

Spring MVC integrates seamlessly with **Thymeleaf** for rendering views.

Inside **src/main/resources/templates/home.html:**

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Home Page</title>
</head>
<body>
    <h1 th:text="${message}"></h1>
</body>
</html>
```

How It Works?

✔ **th:text="${message}"** dynamically binds data (**Welcome to Spring MVC!**).

✔ Spring automatically finds this template and returns the response.

6. Handling Form Submissions

To process user input in an MVC application, use a **POST request**:

Controller Handling Form Submission

```
@Controller
@RequestMapping("/user")
public class UserController {

    @GetMapping("/form")
    public String showForm(Model model) {
        model.addAttribute("user", new User());
        return "form";
    }

    @PostMapping("/submit")
    public String processForm(@ModelAttribute User user, Model model) {
        model.addAttribute("name", user.getName());
        return "result";
    }
}
```

User Model

```
public class User {
    private String name;
    // Getters & Setters
}
```

Thymeleaf Form

```
<form action="/user/submit" method="post">
    <input type="text" name="name" placeholder="Enter Name">
    <button type="submit">Submit</button>
</form>
```

✅ Allows users to submit input, which is processed by **UserController**.

---

7. Exception Handling in Spring MVC

To handle errors globally in a Spring MVC application, use **@ControllerAdvice**:

Global Exception Handler

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public String handleException(Model model, Exception ex) {
        model.addAttribute("error", ex.getMessage());
        return "error"; // Redirects to error view
    }
}
```

✅ **Catches all exceptions** and redirects users to a custom error page.

8. Spring MVC with REST APIs

Although Spring MVC is commonly used for rendering views, it also allows **RESTful API development**:

```java
@RestController
@RequestMapping("/api")
public class ApiController {

    @GetMapping("/users")
    public List<User> getUsers() {
        return List.of(new User("Alice"), new User("Bob"));
    }
}
```

Response (JSON)

```json
[
    { "name": "Alice" },
    { "name": "Bob" }
]
```

✅ Use **@RestController** when returning **JSON data** instead of views.

Final Thoughts

✔ **Spring MVC** is a robust module for building web applications.
✔ It supports **templating engines like Thymeleaf** for dynamic UI rendering.
✔ Spring MVC can also build **RESTful APIs** with **@RestController**.
✔ It includes **exception handling**, **form processing**, and **database integration**.