In **Spring Boot**, @RestController is an annotation used to create RESTful web services. It is a combination of @Controller and @ResponseBody, meaning that it simplifies the creation of REST APIs by automatically serializing returned Java objects into JSON or XML.

### ****Key Features of**** @RestController****:****

1. **Simplifies REST API development**  
   It eliminates the need to annotate every method with @ResponseBody, as it applies it to all request-handling methods by default.
2. **Handles HTTP requests and responses**  
   It is commonly used with @RequestMapping (or @GetMapping, @PostMapping, etc.) to handle HTTP requests.
3. **Supports JSON and XML responses**  
   It uses **Jackson** (by default) to convert Java objects into JSON and **JAXB** for XML (if configured).

### ****Example Usage of**** @RestController

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api")

public class MyController {

@GetMapping("/hello")

public String sayHello() {

return "Hello, World!";

}

@GetMapping("/user")

public User getUser() {

return new User(1, "Devendra Ghag", "devendra@example.com");

}

}

class User {

private int id;

private String name;

private String email;

// Constructor

public User(int id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters and Setters

public int getId() { return id; }

public void setId(int 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; }

}

### ****Equivalent Code Without**** @RestController

If you don't use @RestController, you'd need to manually annotate each method with @ResponseBody:

@Controller

@RequestMapping("/api")

public class MyController {

@GetMapping("/hello")

@ResponseBody

public String sayHello() {

return "Hello, World!";

}

}

Since @RestController does this automatically, it reduces boilerplate code.

### ****When to Use**** @RestController****?****

* When building REST APIs that return JSON/XML.
* When you want to avoid explicitly using @ResponseBody in every method.
* When you are not dealing with traditional **Thymeleaf/JSP** views.

### ****Common Annotations Used with**** @RestController

| **Annotation** | **Description** |
| --- | --- |
| @GetMapping | Handles HTTP **GET** requests |
| @PostMapping | Handles HTTP **POST** requests |
| @PutMapping | Handles HTTP **PUT** requests |
| @DeleteMapping | Handles HTTP **DELETE** requests |
| @RequestParam | Extracts query parameters from the URL |
| @PathVariable | Extracts values from URL path segments |
| @RequestBody | Maps request body to a Java object |

### ****Conclusion****

@RestController is a core component in **Spring Boot** for building RESTful services efficiently. It automatically converts Java objects into JSON/XML and removes the need for @ResponseBody, making REST API development cleaner and easier.

### @RequestMapping ****in Spring****

@RequestMapping is a key annotation in **Spring MVC** and **Spring Boot** used to map HTTP requests to specific handler methods in a controller. It allows developers to define the **URL patterns**, **HTTP methods**, and other configurations required to process incoming requests.

### ****Basic Usage of**** @RequestMapping

@RequestMapping can be applied at both the **class level** and **method level**:

#### **1. Class-Level Mapping**

When applied at the class level, it sets a base URL for all request mappings inside that controller.

@RestController

@RequestMapping("/api")

public class MyController {

@GetMapping("/hello") // Equivalent to @RequestMapping(value = "/hello", method = RequestMethod.GET)

public String sayHello() {

return "Hello, World!";

}

}

* The base URL for all endpoints inside this controller is **/api**.
* @GetMapping("/hello") maps **GET** requests to /api/hello.

#### **2. Method-Level Mapping**

When applied to a method, it specifies the exact endpoint and HTTP method.

@RequestMapping(value = "/greet", method = RequestMethod.GET)

public String greet() {

return "Welcome to Spring!";

}

* This method will handle **GET** requests to /greet.

### ****Using Different HTTP Methods****

Instead of using @RequestMapping(method = RequestMethod.X), Spring provides **shortcut annotations**:

| **Shortcut Annotation** | **Equivalent @RequestMapping** |
| --- | --- |
| @GetMapping | @RequestMapping(method = RequestMethod.GET) |
| @PostMapping | @RequestMapping(method = RequestMethod.POST) |
| @PutMapping | @RequestMapping(method = RequestMethod.PUT) |
| @DeleteMapping | @RequestMapping(method = RequestMethod.DELETE) |

#### **Example**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public String getUser(@PathVariable int id) {

return "User ID: " + id;

}

@PostMapping

public String createUser(@RequestBody String user) {

return "User created: " + user;

}

@PutMapping("/{id}")

public String updateUser(@PathVariable int id, @RequestBody String user) {

return "User " + id + " updated: " + user;

}

@DeleteMapping("/{id}")

public String deleteUser(@PathVariable int id) {

return "User " + id + " deleted.";

}}

### ****Additional Features of**** @RequestMapping

#### **1. Mapping Multiple HTTP Methods**

You can specify multiple HTTP methods for a single endpoint:

@RequestMapping(value = "/multi", method = {RequestMethod.GET, RequestMethod.POST})

public String handleMultipleMethods() {

return "This endpoint supports GET and POST.";

}

#### **2. Mapping Query Parameters (**params**)**

@RequestMapping(value = "/search", params = "name")

public String search(@RequestParam String name) {

return "Searching for: " + name;

}

* Only handles requests where the query parameter **name** is present, e.g., /search?name=Devendra.

#### **3. Mapping Headers (**headers**)**

@RequestMapping(value = "/header", headers = "X-Custom-Header=MyValue")

public String checkHeader() {

return "Header Matched!";

}

* Only processes requests that include X-Custom-Header: MyValue.

#### **4. Producing and Consuming Specific Content Types**

@RequestMapping(value = "/json", produces = "application/json")

public User getUserJson() {

return new User(1, "Devendra", "devendra@example.com");

}

* The response will be returned in **JSON format**.

@RequestMapping(value = "/xml", consumes = "application/xml")

public String processXml(@RequestBody String xmlData) {

return "Received XML: " + xmlData;

}

* This method only processes **XML requests**.

### ****Difference Between**** @RequestMapping ****and**** @GetMapping****,**** @PostMapping****, etc.****

* @RequestMapping is **generic** and supports all HTTP methods.
* @GetMapping, @PostMapping, @PutMapping, etc., are **specialized** for specific HTTP methods.

✅ **Use @RequestMapping when you need flexibility** (e.g., handling multiple methods, headers, or content types).  
✅ **Use @GetMapping, @PostMapping, etc., for simplicity** when handling a specific method.

### ****Conclusion****

* @RequestMapping is a fundamental annotation in **Spring Boot** for mapping URLs to handler methods.
* It allows specifying HTTP methods, query parameters, headers, and content types.
* For specific HTTP methods, @GetMapping, @PostMapping, etc., are preferred.

### @Autowired ****in Spring****

@Autowired is a Spring framework annotation used for **dependency injection**. It allows Spring to automatically resolve and inject **beans** (i.e., objects managed by the Spring container) into your application.

## **Why Use** @Autowired**?**

Instead of manually creating object instances using new, Spring takes care of injecting dependencies, making the application **loosely coupled** and easier to manage.

### ****Example Without**** @Autowired ****(Manual Dependency Injection)****

public class ServiceA {

private ServiceB serviceB;

public ServiceA() {

this.serviceB = new ServiceB(); // Manually creating an object

}

}

Here, ServiceA is tightly coupled to ServiceB, making testing and modifications difficult.

### ****Example With**** @Autowired ****(Spring Dependency Injection)****

@Component

public class ServiceB {

public String getMessage() {

return "Hello from ServiceB";

}

}

@Service

public class ServiceA {

private final ServiceB serviceB;

@Autowired // Spring will automatically inject the dependency

public ServiceA(ServiceB serviceB) {

this.serviceB = serviceB;

}

public String useServiceB() {

return serviceB.getMessage();

}

}

* @Autowired tells Spring to inject an instance of ServiceB automatically into ServiceA.
* Now, ServiceA doesn't create ServiceB manually, reducing **tight coupling**.

## **Ways to Use** @Autowired

### ****1. Constructor Injection (Recommended)****

@Service

public class ServiceA {

private final ServiceB serviceB;

@Autowired

public ServiceA(ServiceB serviceB) { // Constructor-based injection

this.serviceB = serviceB;

}

}

✅ **Best Practice**

* Ensures immutability (final fields can be used).
* Preferred in Spring Boot since **Spring 4.3**, even if @Autowired is omitted.

### ****2. Field Injection (Not Recommended)****

@Service

public class ServiceA {

@Autowired

private ServiceB serviceB;

}

⚠️ **Avoid** field injection because:

* Harder to test (can't inject mocks easily).
* Can't use final (fields remain mutable).

### ****3. Setter Injection****

@Service

public class ServiceA {

private ServiceB serviceB;

@Autowired

public void setServiceB(ServiceB serviceB) { // Setter-based injection

this.serviceB = serviceB;

}

}

✅ Useful when **optional dependencies** need to be injected.

## @Autowired **with Multiple Beans**

If multiple beans of the same type exist, Spring needs help choosing the correct one.

### ****1. Using**** @Primary

@Component

@Primary // Marks this as the default bean

public class ServiceBImpl1 implements ServiceB {}

@Component

public class ServiceBImpl2 implements ServiceB {}

Now, ServiceBImpl1 will be injected by default.

### ****2. Using**** @Qualifier

If you want to specify which bean to inject:

@Service

public class ServiceA {

private final ServiceB serviceB;

@Autowired

public ServiceA(@Qualifier("serviceBImpl2") ServiceB serviceB) {

this.serviceB = serviceB;

}

}

* @Qualifier("beanName") tells Spring to use ServiceBImpl2.

## @Autowired **in** List **or** Map

Injecting all beans of a type:

@Autowired

private List<ServiceB> services; // Injects all beans of type ServiceB

Spring will inject all ServiceB implementations into this list.

## **Conclusion**

* @Autowired is used for **automatic dependency injection** in Spring.
* **Constructor injection** is recommended.
* Use @Primary or @Qualifier to handle multiple beans.
* Avoid **field injection** due to testing limitations.

### ****HTTP Methods in Spring Boot REST Controller****

Spring Boot provides annotations to handle different HTTP methods in a **REST API**. These methods allow the client to **create**, **read**, **update**, and **delete** resources.

| **HTTP Method** | **Spring Annotation** | **Description** |
| --- | --- | --- |
| **GET** | @GetMapping | Fetches data (Read) |
| **POST** | @PostMapping | Creates new data (Create) |
| **PUT** | @PutMapping | Updates existing data (Update) |
| **PATCH** | @PatchMapping | Partially updates data (Update) |
| **DELETE** | @DeleteMapping | Deletes data (Delete) |

## **1. Handling** GET **Requests (**@GetMapping**)**

Used to retrieve data from the server.

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public String getUser(@PathVariable int id) {

return "Fetching user with ID: " + id;

}

}

📌 **Example Request:**  
GET http://localhost:8080/users/1  
📌 **Response:** "Fetching user with ID: 1"

## **2. Handling** POST **Requests (**@PostMapping**)**

Used to create a new resource.

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping

public String createUser(@RequestBody String user) {

return "User created: " + user;

}

}

📌 **Example Request:**

POST http://localhost:8080/users

{

"name": "Devendra",

"email": "dev@example.com"

}

📌 **Response:** "User created: {name: Devendra, email: dev@example.com}"

## **3. Handling** PUT **Requests (**@PutMapping**)**

Used to update an entire resource.

@RestController

@RequestMapping("/users")

public class UserController {

@PutMapping("/{id}")

public String updateUser(@PathVariable int id, @RequestBody String user) {

return "User " + id + " updated: " + user;

}

}

📌 **Example Request:**

PUT http://localhost:8080/users/1

{

"name": "Devendra Ghag",

"email": "dev.ghag@example.com"

}

📌 **Response:** "User 1 updated: {name: Devendra Ghag, email: dev.ghag@example.com}"

## **4. Handling** PATCH **Requests (**@PatchMapping**)**

Used to update **part** of an existing resource.

@RestController

@RequestMapping("/users")

public class UserController {

@PatchMapping("/{id}")

public String updateUserEmail(@PathVariable int id, @RequestBody String email) {

return "User " + id + " email updated to: " + email;

}

}

📌 **Example Request:**

PATCH http://localhost:8080/users/1

{

"email": "new.email@example.com"

}

📌 **Response:** "User 1 email updated to: new.email@example.com"

## **5. Handling** DELETE **Requests (**@DeleteMapping**)**

Used to delete a resource.

@RestController

@RequestMapping("/users")

public class UserController {

@DeleteMapping("/{id}")

public String deleteUser(@PathVariable int id) {

return "User " + id + " deleted.";

}

} **Example Request:**  
DELETE http://localhost:8080/users/1  
📌 **Response:** "User 1 deleted."

**Conclusion**

* @GetMapping → Retrieves data.
* @PostMapping → Creates new data.
* @PutMapping → Updates entire data.
* @PatchMapping → Partially updates data.
* @DeleteMapping → Deletes data.

### @PathVariable ****in Spring Boot****

@PathVariable is used in Spring Boot to extract values from the **URL path** and use them as method parameters in a RESTful API. It allows dynamic handling of URLs.

## **1. Basic Usage of** @PathVariable

@PathVariable is commonly used in **GET, PUT, DELETE** requests where a unique identifier (e.g., user ID) is passed in the URL.

### ****Example****

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}") // URL: /users/{id}

public String getUserById(@PathVariable int id) {

return "Fetching user with ID: " + id;

}

}

📌 **Example Request:**  
GET http://localhost:8080/users/101  
📌 **Response:** "Fetching user with ID: 101"

## **2. Multiple** @PathVariable **Parameters**

You can extract multiple values from the URL.

### ****Example****

@RestController

@RequestMapping("/orders")

public class OrderController {

@GetMapping("/{userId}/products/{productId}")

public String getUserOrder(@PathVariable int userId, @PathVariable int productId) {

return "User " + userId + " ordered product " + productId;

}

}

📌 **Example Request:**  
GET http://localhost:8080/orders/10/products/500  
📌 **Response:** "User 10 ordered product 500"

## **3.** @PathVariable **with Custom Name**

By default, the parameter name must match the placeholder in {}. But if they differ, you can specify the name explicitly.

### ****Example****

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@GetMapping("/{empId}")

public String getEmployee(@PathVariable("empId") int id) {

return "Employee ID: " + id;

}

}

📌 **Example Request:**  
GET http://localhost:8080/employees/25  
📌 **Response:** "Employee ID: 25"

## **4.** @PathVariable **with Different Data Types**

Spring automatically converts path variables to the required type.

### ****Example with**** String ****and**** UUID

@RestController

@RequestMapping("/products")

public class ProductController {

@GetMapping("/{category}/{productId}")

public String getProduct(@PathVariable String category, @PathVariable UUID productId) {

return "Product " + productId + " in category " + category;

}

}

📌 **Example Request:**  
GET http://localhost:8080/products/electronics/550e8400-e29b-41d4-a716-446655440000  
📌 **Response:** "Product 550e8400-e29b-41d4-a716-446655440000 in category electronics"

## **5. Optional** @PathVariable

If the path variable is **optional**, you can set a default value.

### ****Example****

@RestController

@RequestMapping("/customers")

public class CustomerController {

@GetMapping(value = {"/{id}", "/"}) // Supports both /customers/{id} and /customers/

public String getCustomer(@PathVariable(required = false) Integer id) {

if (id == null) {

return "Fetching all customers";

}

return "Fetching customer with ID: " + id;

}

}

📌 **Example Requests:**

* GET http://localhost:8080/customers/5 → "Fetching customer with ID: 5"
* GET http://localhost:8080/customers/ → "Fetching all customers"

## **Conclusion**

* @PathVariable extracts values from the URL path.
* Supports multiple parameters and custom names.
* Automatically converts to required types (int, UUID, String).
* Can be made **optional** with required = false.

### @RequestBody ****in Spring Boot****

@RequestBody is used in Spring Boot to **map the request body (JSON/XML)** to a **Java object** in RESTful APIs. It allows sending complex data (like user details) from the client to the server.

## **1. Basic Usage of** @RequestBody

It is commonly used with **POST, PUT, and PATCH** requests to receive data.

### ****Example (****POST ****Request)****

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping

public String createUser(@RequestBody User user) {

return "User created: " + user.getName();

}

}

#### **User Model**

public class User {

private String name;

private String email;

// Getters and Setters

}

📌 **Example Request (JSON Body)**

POST http://localhost:8080/users

{

"name": "Devendra",

"email": "dev@example.com"

}

📌 **Response:**  
"User created: Devendra"

## **2. Using** @RequestBody **in** PUT **Requests (Updating Data)**

When updating an existing user:

@PutMapping("/{id}")

public String updateUser(@PathVariable int id, @RequestBody User user) {

return "User " + id + " updated with name: " + user.getName();

}

📌 **Example Request (Update User 1)**

PUT http://localhost:8080/users/1

{

"name": "Devendra Ghag",

"email": "dev.ghag@example.com"

}

📌 **Response:**  
"User 1 updated with name: Devendra Ghag"

## **3. Using** @RequestBody **in** PATCH **Requests (Partial Update)**

For partial updates (e.g., updating only email):

@PatchMapping("/{id}")

public String updateUserEmail(@PathVariable int id, @RequestBody Map<String, String> updates) {

return "User " + id + " email updated to: " + updates.get("email");

}

📌 **Example Request (Update Email Only)**

PATCH http://localhost:8080/users/1

{

"email": "new.email@example.com"

}

📌 **Response:**  
"User 1 email updated to: new.email@example.com"

## **4. Handling** @RequestBody **with Lists**

You can send an array of objects in JSON and map them to a **Java List**.

@PostMapping("/bulk")

public String createUsers(@RequestBody List<User> users) {

return "Created " + users.size() + " users.";

}

📌 **Example Request (JSON List)**

POST http://localhost:8080/users/bulk

[

{"name": "Devendra", "email": "dev@example.com"},

{"name": "Shruti", "email": "shruti@example.com"}

]

📌 **Response:**  
"Created 2 users."

## **5. Validating** @RequestBody **Using** @Valid

If you want to **validate input data**, use @Valid along with @RequestBody.

import jakarta.validation.Valid;

import jakarta.validation.constraints.\*;

public class User {

@NotBlank(message = "Name is required")

private String name;

@Email(message = "Invalid email")

private String email;

// Getters and Setters

}

#### **Controller with Validation**

@PostMapping

public String createUser(@Valid @RequestBody User user) {

return "User created: " + user.getName();

}

If a request has an invalid email:

POST http://localhost:8080/users

{

"name": "Devendra",

"email": "invalid-email"

}

📌 **Response:**

{

"error": "Invalid email"

}

## **Conclusion**

* @RequestBody **binds JSON/XML data** from the request body to a Java object.
* Commonly used in **POST, PUT, PATCH** requests.
* Supports **lists**, **maps**, and **custom objects**.
* Use **validation (@Valid)** to ensure data integrity.
* @Repository **in Spring Boot**

@Repository is a **Spring annotation** used to indicate that a class is responsible for **interacting with the database**. It is mainly used in **DAO (Data Access Object) layers** to handle data persistence.

## **1. Role of** @Repository **in Spring**

* Marks a class as a **database repository (DAO layer)**.
* Helps **Spring manage exceptions** related to database operations.
* Works with **Spring Data JPA** for database interactions.

## **2. Basic Usage of** @Repository

A repository is used to interact with a **database table**.

### ****Example: User Repository****

#### **User Entity (JPA Model)**

import jakarta.persistence.\*;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and Setters

}

#### **User Repository (**@Repository**)**

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

User findByEmail(String email);

}

📌 **Explanation:**

* @Repository marks this as a **database interaction layer**.
* JpaRepository<User, Long> provides **built-in CRUD methods**.
* findByEmail(String email) is a **custom query method**.

## **3. Using** @Repository **in a Service Layer**

A **service class** calls the repository to interact with the database.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User getUserByEmail(String email) {

return userRepository.findByEmail(email);

}

}

## **4. Using** @Repository **in a Controller**

A REST API **calls the service**, which interacts with the repository.

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping

public List<User> getUsers() {

return userService.getAllUsers();

}

@GetMapping("/email/{email}")

public User getUserByEmail(@PathVariable String email) {

return userService.getUserByEmail(email);

}

}

📌 **Example Requests:**

1. GET http://localhost:8080/users → Fetch all users.
2. GET http://localhost:8080/users/email/dev@example.com → Find a user by email.

## **5. Alternative:** @Repository **Without Spring Data JPA**

If you don't use Spring Data JPA, you can implement JdbcTemplate manually.

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public class UserDao {

@Autowired

private JdbcTemplate jdbcTemplate;

public List<User> getAllUsers() {

return jdbcTemplate.query("SELECT \* FROM user", new UserRowMapper());

}

}

📌 **Here:**

* JdbcTemplate runs SQL queries directly.
* UserRowMapper maps database rows to Java objects.

## **Conclusion**

* @Repository is used for **database interaction**.
* Works with **Spring Data JPA** for built-in CRUD methods.
* Helps **handle database exceptions** automatically.
* Can be used with **JdbcTemplate** for manual queries.

### @Service ****in Spring Boot****

@Service is a **Spring annotation** used to indicate that a class contains **business logic**. It is part of the **service layer** in a Spring Boot application and is used to handle business operations.

## **1. Role of** @Service **in Spring**

* Used to **write business logic** separately from the controller and repository layers.
* Helps in **code reusability** and better **separation of concerns**.
* Automatically detected by **Spring’s component scanning**.
* Works with @Autowired to inject dependencies.

## **2. Basic Usage of** @Service

A service **calls the repository** to interact with the database and perform business logic.

### ****Example: User Service****

#### **User Entity (JPA Model)**

import jakarta.persistence.\*;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and Setters

}

#### **User Repository (**@Repository**)**

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

User findByEmail(String email);

}

#### **User Service (**@Service**)**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public Optional<User> getUserById(Long id) {

return userRepository.findById(id);

}

public User saveUser(User user) {

return userRepository.save(user);

}

public void deleteUser(Long id) {

userRepository.deleteById(id);

}

}

## **3. Using** @Service **in a Controller**

The **controller layer** calls the **service layer**, which then interacts with the **repository**.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping

public List<User> getUsers() {

return userService.getAllUsers();

}

@GetMapping("/{id}")

public Optional<User> getUserById(@PathVariable Long id) {

return userService.getUserById(id);

}

@PostMapping

public User createUser(@RequestBody User user) {

return userService.saveUser(user);

}

@DeleteMapping("/{id}")

public String deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

return "User deleted";

}

}

📌 **Example Requests:**

1. GET http://localhost:8080/users → Fetch all users.
2. GET http://localhost:8080/users/1 → Fetch user with ID **1**.
3. POST http://localhost:8080/users → Create a new user.
4. DELETE http://localhost:8080/users/1 → Delete user **1**.

## **4. Difference Between** @Service**,** @Component**, and** @Repository

| **Annotation** | **Purpose** |
| --- | --- |
| **@Component** | Generic annotation for **any Spring-managed bean**. |
| **@Service** | Used for **business logic** (service layer). |
| **@Repository** | Used for **database access** (repository/DAO layer). |

## **5. When to Use** @Service**?**

* When you have **business logic** that should be separate from the controller.
* When calling a **database or external API**.
* When performing **calculations or validations** before saving data.

## **Conclusion**

* @Service is part of the **service layer** in a Spring Boot application.
* It **contains business logic** and calls the repository.
* It improves **code structure** and **separates concerns**.
* Works well with **@Autowired** for dependency injection.

### ****Why Do We Use Controller, Service, and Repository Layers in Spring Boot?****

In a Spring Boot application, we follow the **three-layer architecture** to ensure **separation of concerns**, better maintainability, and scalability. These three layers are:

1. **Controller Layer (@RestController)** → Handles **HTTP requests and responses**.
2. **Service Layer (@Service)** → Contains **business logic**.
3. **Repository Layer (@Repository)** → Interacts with the **database**.

## **1. Controller Layer (**@RestController**)**

* The controller layer **handles incoming API requests** (like GET, POST, PUT, DELETE).
* It interacts with the **Service Layer** to process requests.
* Sends the **response** (JSON or XML) back to the client.

📌 **Example:**

@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.saveUser(user);

}

}

🛠️ **Why use a controller?**  
✔️ Separates **API logic** from business logic.  
✔️ Handles **user input validation and HTTP responses**.

## **2. Service Layer (**@Service**)**

* The service layer contains the **business logic** of the application.
* It interacts with the **Repository Layer** to fetch or store data.
* It **processes data**, applies **rules**, and performs **calculations**.

📌 **Example:**

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User saveUser(User user) {

// Additional business logic can be added here

return userRepository.save(user);

}

}

🛠️ **Why use a service?**  
✔️ Separates **business logic** from API handling.  
✔️ Makes the code **reusable and testable**.  
✔️ Allows easy modification of logic without affecting controllers.

## **3. Repository Layer (**@Repository**)**

* The repository layer is responsible for **database operations**.
* It uses **Spring Data JPA** (or JdbcTemplate) to interact with the database.
* Helps in performing **CRUD (Create, Read, Update, Delete) operations**.

📌 **Example:**

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

User findByEmail(String email);

}

🛠️ **Why use a repository?**  
✔️ **Abstracts database logic** from the business layer.  
✔️ Uses **Spring Data JPA** for easy CRUD operations.  
✔️ Reduces **boilerplate code**.

## **4. Summary: How They Work Together**

| **Layer** | **Purpose** | **Example Code** |
| --- | --- | --- |
| **Controller (@RestController)** | Handles HTTP requests | @GetMapping, @PostMapping |
| **Service (@Service)** | Business logic and processing | public User saveUser(User user) |
| **Repository (@Repository)** | Database interaction | JpaRepository<User, Long> |

## **5. Why Follow This Structure?**

✅ **Separation of Concerns** – Each layer has a clear responsibility.  
✅ **Maintainability** – Easier to modify or debug without affecting the whole system.  
✅ **Reusability** – Services can be reused across multiple controllers.  
✅ **Scalability** – The system can grow and remain manageable.

### ****How Spring Boot Handles a Request – Step by Step**** 🚀

When a client (browser, Postman, or mobile app) sends an HTTP request, Spring Boot **processes it step by step**. Let’s go through the entire request lifecycle in Spring Boot.

## **🔹 Step 1: Client Sends a Request**

A client sends an HTTP request (GET, POST, PUT, DELETE) to a **Spring Boot API**.

📌 **Example Request:**

GET http://localhost:8080/users/1

🔹 The client requests **user data** for id=1.

## **🔹 Step 2: Tomcat Receives the Request**

Spring Boot **runs an embedded Tomcat server** (default on port 8080).

**What happens?**

1. Tomcat **listens for incoming HTTP requests**.
2. It **forwards the request** to **Spring Boot’s Dispatcher Servlet**.

## **🔹 Step 3: Dispatcher Servlet Processes the Request**

Spring Boot has a **Front Controller** called **DispatcherServlet** that:

1. **Receives the request** from Tomcat.
2. **Finds the correct controller** based on the URL.
3. **Calls the appropriate method** in the controller.

📌 **Spring Boot automatically registers DispatcherServlet.**

org.springframework.web.servlet.DispatcherServlet

## **🔹 Step 4: Controller Handles the Request (**@RestController**)**

The **controller** processes the request and calls the **service layer**.

📌 **Example Controller (@RestController)**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

public User getUserById(@PathVariable Long id) {

return userService.getUserById(id);

}

}

🔹 **What happens here?**

* @RestController tells Spring Boot that this class handles **REST API requests**.
* @GetMapping("/{id}") maps **GET requests** like /users/1 to getUserById().
* The controller calls **UserService** to **fetch user data**.

## **🔹 Step 5: Service Layer Processes Business Logic (**@Service**)**

The **service layer** fetches data and applies business logic.

📌 **Example Service (@Service)**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public User getUserById(Long id) {

return userRepository.findById(id).orElseThrow(() -> new RuntimeException("User not found"));

}

}

🔹 **What happens here?**

* @Service marks this class as a **service layer**.
* The method getUserById(id) calls the **repository layer** to **fetch data**.
* If the user is **not found**, it throws an **exception**.

## **🔹 Step 6: Repository Fetches Data from Database (**@Repository**)**

The **repository layer** interacts with the **database** using JPA.

📌 **Example Repository (@Repository)**

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {

}

🔹 **What happens here?**

* JpaRepository<User, Long> provides **database operations** (findById, save, delete).
* Spring Boot **auto-generates SQL queries** behind the scenes.

📌 **Generated SQL Query (internally by JPA)**:

SELECT \* FROM users WHERE id = 1;

## **🔹 Step 7: Service Returns Data to Controller**

* The **repository fetches the user** from the database.
* The **service processes the data** and sends it **back to the controller**.

📌 **Example Response Data (Java Object)**

User user = new User(1L, "John Doe", "john@example.com");

## **🔹 Step 8: Controller Returns JSON Response**

The **controller converts the Java object** into **JSON**.

📌 **Example JSON Response**

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

🔹 **What happens internally?**

* Spring Boot **uses Jackson** (by default) to **convert Java objects to JSON**.
* The response is sent **back to the client**.

## **🔹 Step 9: Client Receives the Response**

* The client (browser, Postman, or mobile app) **receives the JSON response**.
* The requested **data is displayed** to the user.

📌 **Final Output in Postman**

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

## **🎯 Summary: Request Flow in Spring Boot**

| **Step** | **Action** | **What Happens Internally?** |
| --- | --- | --- |
| **1** | Client sends a request | GET /users/1 |
| **2** | Tomcat receives the request | Forwarded to DispatcherServlet |
| **3** | DispatcherServlet processes request | Finds the correct controller |
| **4** | Controller handles request | Calls UserService.getUserById(id) |
| **5** | Service layer processes logic | Calls UserRepository.findById(id) |
| **6** | Repository fetches data from DB | JPA generates SELECT query |
| **7** | Service returns data | Sends User object to controller |
| **8** | Controller returns JSON response | Converts Java object to JSON |
| **9** | Client receives response | {"id":1, "name":"John Doe"} |

## **🔥 Example for Other HTTP Methods**

| **HTTP Method** | **Mapping Annotation** | **Example** |
| --- | --- | --- |
| **GET** | @GetMapping | Fetch data (GET /users/{id}) |
| **POST** | @PostMapping | Create data (POST /users) |
| **PUT** | @PutMapping | Update data (PUT /users/{id}) |
| **DELETE** | @DeleteMapping | Delete data (DELETE /users/{id}) |

📌 **Example: Handling POST Requests**

@PostMapping

public User createUser(@RequestBody User user) {

return userService.saveUser(user);

}

🔹 **What happens?**

* @RequestBody takes **JSON input** and converts it into a **Java object**.
* The **service layer** saves the user in the database.
* A **JSON response** is sent back.

## **🎯 Conclusion**

🔹 **Spring Boot automates the entire request flow:**  
✅ **Receives HTTP requests** → ✅ **Processes request in controllers** → ✅ **Handles business logic in services** → ✅ **Fetches data from the database** → ✅ **Returns JSON response to the client**.

### RequestEntity ****in Spring Boot**** 🚀

RequestEntity<T> is a **more advanced** version of @RequestBody. It **encapsulates**:  
✅ **HTTP Headers**  
✅ **Request Body**  
✅ **HTTP Method & URI**

It is used when you need **full control over the incoming request**.

## **🔹 Basic Usage of** RequestEntity

📌 **Example: Handling a POST request with RequestEntity**

import org.springframework.http.RequestEntity;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping("/create")

public ResponseEntity<String> createUser(@RequestBody RequestEntity<User> requestEntity) {

// Extracting request details

User user = requestEntity.getBody();

System.out.println("Headers: " + requestEntity.getHeaders());

System.out.println("Method: " + requestEntity.getMethod());

return ResponseEntity.ok("User Created: " + user.getName());

}

}

🔹 **What happens here?**

1. RequestEntity<User> captures **headers, body, method, and URI**.
2. We extract the **request body** (requestEntity.getBody()).
3. The response is returned using ResponseEntity.

📌 **Example Request (JSON)**

POST /users/create

Content-Type: application/json

Authorization: Bearer token123

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

📌 **Console Output (Extracted from RequestEntity)**

Headers: [Content-Type:"application/json", Authorization:"Bearer token123"]

Method: POST

📌 **Response:**

"User Created: John Doe"

## **🔹 Why Use** RequestEntity**?**

| **Feature** | **@RequestBody** | **RequestEntity<T>** |
| --- | --- | --- |
| Extract Request Body | ✅ Yes | ✅ Yes |
| Extract Headers | ❌ No | ✅ Yes |
| Extract HTTP Method | ❌ No | ✅ Yes |
| Extract URL Info | ❌ No | ✅ Yes |

## **🔹 Example: Handling** GET **Request with** RequestEntity

📌 **Extract Headers from a GET request**

@GetMapping("/info")

public ResponseEntity<String> getInfo(RequestEntity<Void> requestEntity) {

System.out.println("Headers: " + requestEntity.getHeaders());

System.out.println("Method: " + requestEntity.getMethod());

return ResponseEntity.ok("Request Received");

}

📌 **Example GET Request**

GET /users/info

User-Agent: Postman

Authorization: Bearer token123

📌 **Console Output**

Headers: [User-Agent:"Postman", Authorization:"Bearer token123"]

Method: GET

## **🎯 When to Use** RequestEntity**?**

✅ When you **need to access request headers**.  
✅ When you **need the HTTP method or URI information**.  
✅ When building **REST clients that consume APIs dynamically**.

### ResponseEntity ****in Spring Boot**** 🚀

ResponseEntity<T> represents an **HTTP response** in a Spring Boot REST API. It allows you to control:  
✅ **HTTP Status Codes** (200, 400, 500, etc.)  
✅ **Response Headers**  
✅ **Response Body**

## **🔹 Basic Usage of** ResponseEntity

📌 **Example: Returning a Custom Response**

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public ResponseEntity<String> getUser(@PathVariable Long id) {

if (id == 1) {

return ResponseEntity.ok("User Found: John Doe");

} else {

return ResponseEntity.status(404).body("User Not Found");

}

}

}

📌 **Example Requests & Responses**

✅ **Success (200 OK)**

GET /users/1

📌 **Response:**

"User Found: John Doe"

❌ **Failure (404 Not Found)**

GET /users/2

📌 **Response:**

"User Not Found"

## **🔹 Controlling HTTP Status Codes**

You can return different HTTP status codes based on conditions.

📌 **Example: Returning 201 Created for a New User**

@PostMapping("/create")

public ResponseEntity<String> createUser(@RequestBody User user) {

return ResponseEntity.status(201).body("User Created: " + user.getName());

}

📌 **Response (201 Created)**

"User Created: John Doe"

## **🔹 Adding Response Headers**

You can also **add custom headers**.

📌 **Example: Returning a Custom Header**

@GetMapping("/header")

public ResponseEntity<String> getCustomHeader() {

return ResponseEntity.ok()

.header("Custom-Header", "MyHeaderValue")

.body("Response with Custom Header");

}

📌 **Response Headers:**

HTTP/1.1 200 OK

Custom-Header: MyHeaderValue

## **🔹 Returning JSON Data**

📌 **Example: Returning a JSON Object**

@GetMapping("/{id}")

public ResponseEntity<User> getUserJson(@PathVariable Long id) {

User user = new User(id, "John Doe", "john@example.com");

return ResponseEntity.ok(user);

}

📌 **Response:**

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

## **🔹 Summary**

| **Feature** | **@ResponseBody** | **ResponseEntity<T>** |
| --- | --- | --- |
| Return Response Body | ✅ Yes | ✅ Yes |
| Control HTTP Status | ❌ No | ✅ Yes |
| Add Headers | ❌ No | ✅ Yes |

### @RequestParam ****vs**** @PathVariable ****in Spring Boot**** 🚀

Both @RequestParam and @PathVariable are used to extract values from an HTTP request, but they serve **different purposes**.

## **🔹 1.** @RequestParam **– Extracts Query Parameters**

✅ Used to **extract query parameters** from the URL.  
✅ Used in GET requests when parameters are sent **after ? in the URL**.

📌 **Example:**

@GetMapping("/greet")

public String greet(@RequestParam String name) {

return "Hello, " + name;

}

📌 **Example Request (Query Parameter)**

GET /greet?name=Devendra

📌 **Response**

"Hello, Devendra"

### ****✔ Key Points for**** @RequestParam

* Extracts **query parameters** (?key=value format).
* Can have **default values** and **optional parameters**.
* Mostly used in GET requests.

📌 **Example with Default Value & Optional Parameter**

@GetMapping("/greet")

public String greet(@RequestParam(defaultValue = "Guest") String name) {

return "Hello, " + name;

}

📌 **If no name is provided, it defaults to "Guest".**

## **🔹 2.** @PathVariable **– Extracts Path Parameters**

✅ Used to **extract values from the URL path itself**.  
✅ Typically used in **RESTful APIs** for resource identification.

📌 **Example:**

@GetMapping("/users/{id}")

public String getUserById(@PathVariable Long id) {

return "Fetching user with ID: " + id;

}

📌 **Example Request (Path Parameter)**

GET /users/10

📌 **Response**

"Fetching user with ID: 10"

### ****✔ Key Points for**** @PathVariable

* Extracts **path parameters** from the URL.
* Used in **RESTful API endpoints** (/users/{id}).
* Values **must be included in the URL**.

📌 **Example with Multiple @PathVariable Parameters**

@GetMapping("/users/{userId}/orders/{orderId}")

public String getUserOrder(@PathVariable Long userId, @PathVariable Long orderId) {

return "User " + userId + " ordered " + orderId;

}

📌 **Request**

GET /users/5/orders/100

📌 **Response**

"User 5 ordered 100"

## **🔹 Key Differences Between** @RequestParam **&** @PathVariable

| **Feature** | **@RequestParam** | **@PathVariable** |
| --- | --- | --- |
| Extracts from | **Query Parameters (?key=value)** | **URL Path (/{value})** |
| Example URL | /users?name=John | /users/John |
| Used for | Filters, search queries | Identifying resources |
| Default Values | ✅ Can set a default (defaultValue = "") | ❌ No default support |
| Required | ✅ Can be optional | ✅ Required |

## **🎯 When to Use What?**

✅ Use @RequestParam for **optional values** (e.g., filtering, sorting).  
✅ Use @PathVariable for **mandatory values** (e.g., resource IDs in REST APIs).

### @SpringBootApplication ****in Spring Boot**** 🚀

@SpringBootApplication is the **main entry point** for a Spring Boot application. It is a **meta-annotation** that combines three key annotations:

1️⃣ **@Configuration** – Marks the class as a Spring configuration class.  
2️⃣ **@EnableAutoConfiguration** – Enables Spring Boot's auto-configuration.  
3️⃣ **@ComponentScan** – Scans the package and sub-packages for Spring components.

## **🔹 Example: Using** @SpringBootApplication

📌 **Main Class (Application.java)**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

📌 **What Happens Here?**  
✅ Starts the **Spring Boot application**.  
✅ Enables **auto-configuration** (e.g., embedded Tomcat, DataSource setup).  
✅ Scans for **components (@Component, @Service, @Repository)** in the package.

## **🔹** @SpringBootApplication **=** @Configuration **+** @EnableAutoConfiguration **+** @ComponentScan

Let's break it down:

### ****1️⃣**** @Configuration ****– Defines Beans****

Marks the class as a **Spring configuration class**.

@Configuration

public class MyConfig {

@Bean

public String myBean() {

return "Hello, Spring!";

}

}

### ****2️⃣**** @EnableAutoConfiguration ****– Enables Auto Configuration****

Automatically configures components like **Spring MVC, Hibernate, or DataSource** based on dependencies.

### ****3️⃣**** @ComponentScan ****– Scans for Components****

Automatically detects classes annotated with **@Component, @Service, @Repository, @Controller**.

📌 **Without @SpringBootApplication, we would write:**

@Configuration

@EnableAutoConfiguration

@ComponentScan

public class Application { }

✅ @SpringBootApplication **simplifies this**.

## **🔹 Customizing** @SpringBootApplication

You can **exclude specific auto-configurations**:

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })

public class Application { }

📌 **This disables the default database configuration.**

## **🎯 Key Points About** @SpringBootApplication

✔ **Main annotation** to bootstrap a Spring Boot app.  
✔ Automatically **configures dependencies**.  
✔ Scans components in **current package & sub-packages**.  
✔ Reduces **boilerplate code**.

### @PreAuthorize ****in Spring Security**** 🔒🚀

@PreAuthorize is a Spring Security annotation used to **restrict access** to methods based on user roles, permissions, or conditions. It is applied at the **method level** to enforce security **before the method executes**.

🔹 **Works with Spring Security & Expression-Based Access Control.**  
🔹 **Evaluates SpEL (Spring Expression Language) expressions** to check permissions.

## **🔹 Example: Securing Methods with** @PreAuthorize

📌 **Basic Example – Restricting Access by Role**

import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/admin")

public class AdminController {

@GetMapping("/dashboard")

@PreAuthorize("hasRole('ADMIN')") // Only users with "ADMIN" role can access

public String adminDashboard() {

return "Welcome to the Admin Dashboard!";

}

}

📌 **How It Works?**  
✅ **Users with the role "ADMIN" can access** the /admin/dashboard endpoint.  
✅ If a non-admin tries to access it, **403 Forbidden** error is returned.

## **🔹 More** @PreAuthorize **Examples**

### ****✔ Allow Access to Users with Multiple Roles****

@PreAuthorize("hasRole('ADMIN') or hasRole('MANAGER')")

✅ **Admins & Managers** can access.

### ****✔ Checking User Permissions (****hasAuthority****)****

Instead of roles, we can check **specific permissions**:

@PreAuthorize("hasAuthority('READ\_PRIVILEGE')")

✅ Checks if the user has **"READ\_PRIVILEGE"**.

### ****✔ Checking Multiple Authorities****

@PreAuthorize("hasAuthority('READ\_PRIVILEGE') and hasAuthority('WRITE\_PRIVILEGE')")

✅ **Requires both** read and write privileges.

### ****✔ Access Based on User Identity (****principal****)****

@PreAuthorize("#username == authentication.principal.username")

public String getUserData(String username) {

return "User data for: " + username;

}

✅ **Allows users to access only their own data**.

### ****✔ Using Custom Service Methods in**** @PreAuthorize

We can call **custom security methods** inside @PreAuthorize:

@PreAuthorize("@securityService.isAdmin(authentication)")

✅ Calls a method in SecurityService to check if the user is an admin.

## **🔹** @PreAuthorize **vs** @PostAuthorize

| **Annotation** | **Description** |
| --- | --- |
| @PreAuthorize | Checks authorization **before** method execution. |
| @PostAuthorize | Checks authorization **after** method execution (useful for filtering return values). |

📌 **Example of @PostAuthorize**:

@PostAuthorize("returnObject.owner == authentication.name")

public Order getOrder(Long orderId) {

return orderService.findById(orderId);

}

✅ Ensures **users can only retrieve their own orders**.

## **🎯 Key Takeaways**

✔ @PreAuthorize restricts access **before execution**.  
✔ Supports **roles (hasRole) & permissions (hasAuthority)**.  
✔ Can use **SpEL expressions** for dynamic access control.  
✔ Works with **Spring Security & method-level security**.

### @Valid ****in Spring Boot**** 🚀

@Valid is used for **request body validation** in Spring Boot. It works with **Java Bean Validation API** (javax.validation).

## **🔹 Example: Using** @Valid **for Input Validation**

📌 **DTO with Validation Annotations**

java

import jakarta.validation.constraints.\*;

public class UserDTO {

@NotBlank(message = "Name cannot be empty")

private String name;

@Email(message = "Invalid email format")

private String email;

@Min(value = 18, message = "Age must be at least 18")

private int age;

}

📌 **Controller with @Valid**

java

import org.springframework.web.bind.annotation.\*;

import jakarta.validation.Valid;

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping("/register")

public String registerUser(@Valid @RequestBody UserDTO user) {

return "User registered: " + user.getName();

}

}

## **🔹 What Happens If Input Is Invalid?**

📌 **Request (Invalid Data)**

json

{

"name": "",

"email": "invalid-email",

"age": 15

}

📌 **Response (400 Bad Request)**

json

{

"errors": [

"Name cannot be empty",

"Invalid email format",

"Age must be at least 18"

]

}

## **🔹 Handling Validation Errors Gracefully**

You can **customize error messages** with a global exception handler.

📌 **Example: Handling MethodArgumentNotValidException**

java

import org.springframework.web.bind.annotation.\*;

import org.springframework.http.\*;

import jakarta.validation.Valid;

import org.springframework.validation.\*;

import java.util.\*;

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResponseEntity<Map<String, List<String>>> handleValidationErrors(MethodArgumentNotValidException ex) {

Map<String, List<String>> errors = new HashMap<>();

List<String> errorMessages = new ArrayList<>();

for (FieldError error : ex.getBindingResult().getFieldErrors()) {

errorMessages.add(error.getDefaultMessage());

}

errors.put("errors", errorMessages);

return new ResponseEntity<>(errors, HttpStatus.BAD\_REQUEST);

}

}

## **🎯 When to Use** @Valid**?**

✅ To **validate user input** in REST APIs.  
✅ To enforce **data integrity** before processing requests.  
✅ To automatically **return validation errors** in a structured way.

### @Bean ****in Spring Boot**** 🌱

The @Bean annotation in Spring Boot is used to define **a method that returns a Spring-managed bean**. It is applied inside a class annotated with @Configuration, and Spring automatically registers the returned object in its **application context**.

## **🔹 Why Use** @Bean**?**

✅ **Custom Bean Creation** – When you need to create and manage a bean manually.  
✅ **Third-party Library Integration** – When you can't use @Component, @Service, or @Repository.  
✅ **Configuration & Dependency Injection** – Helps configure beans centrally.

## **🔹 Example: Creating a Bean Using** @Bean

📌 **Example: Defining a Bean for a Custom Service**

java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration // Marks this class as a configuration provider

public class AppConfig {

@Bean // Defines a bean named "myService"

public MyService myService() {

return new MyService();

}

}

📌 **Using the Bean in Another Class**

java

import org.springframework.stereotype.Service;

@Service

public class MyClientService {

private final MyService myService;

// Spring injects the "myService" bean automatically

public MyClientService(MyService myService) {

this.myService = myService;

}

}

✅ Spring **automatically injects the myService bean** into MyClientService.

## **🔹** @Bean **vs** @Component **(or** @Service**,** @Repository**)**

| **Feature** | **@Bean** | **@Component / @Service** |
| --- | --- | --- |
| Used In | @Configuration classes | Any class |
| Object Creation | Custom logic in a method | Spring automatically instantiates |
| Third-Party Libraries | ✅ Yes | ❌ No |
| Explicit Bean Name | ✅ Yes (@Bean("beanName")) | ❌ No (default to class name) |

## **🔹 Example: Customizing a Bean**

📌 **Naming a Bean Manually**

java

@Bean(name = "customService")

public MyService myCustomService() {

return new MyService();

}

📌 **Injecting by Name**

java

@Autowired

@Qualifier("customService")

private MyService myService;

✅ Ensures **we inject the correct bean** when multiple beans exist.

## **🔹 Example: Configuring a** RestTemplate **Bean**

📌 **Defining a RestTemplate Bean for API Calls**

java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class RestConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

📌 **Using the RestTemplate Bean**

java

@Autowired

private RestTemplate restTemplate;

✅ **Allows dependency injection instead of manually creating objects.**

## **🎯 Key Takeaways**

✔ @Bean **defines and manages Spring beans explicitly**.  
✔ Useful when dealing with **third-party classes**.  
✔ Used inside **@Configuration classes**.  
✔ Provides **more control** than @Component-based beans.

### @Configuration ****in Spring Boot**** 🚀

@Configuration is an **annotation in Spring Boot** that marks a class as a **configuration class**. It is used to define **Spring Beans** using @Bean methods.

## **🔹 Why Use** @Configuration**?**

✅ **Centralized Configuration** – Keeps bean definitions in one place.  
✅ **Replaces XML Configurations** – Eliminates applicationContext.xml.  
✅ **Creates & Manages Beans** – Works with @Bean to create Spring-managed beans.

## **🔹 Example: Defining Beans with** @Configuration

📌 **Java-based Configuration**

java

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

@Configuration

public class AppConfig {

@Bean

public String myBean() {

return "Hello, Spring!";

}

}

📌 **Using the Bean in Another Class**

java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class MyService {

private final String myBean;

@Autowired

public MyService(String myBean) {

this.myBean = myBean;

}

public String getBeanMessage() {

return myBean;

}

}

## **🔹 Alternative: Without** @Configuration

Without @Configuration, we would define beans in applicationContext.xml:

xml

<bean id="myBean" class="java.lang.String">

<constructor-arg value="Hello, Spring!"/>

</bean>

✅ @Configuration **eliminates XML-based bean definitions**.

## **🔹 Difference Between** @Component **and** @Configuration

| **Feature** | **@Component** | **@Configuration** |
| --- | --- | --- |
| Purpose | Marks a class as a **Spring-managed component** | Defines **beans** in a centralized way |
| Used with | @Service, @Repository, @Controller | @Bean methods |
| Scanned by @ComponentScan? | ✅ Yes | ✅ Yes |
| Example | @Component public class MyComponent {} | @Configuration public class AppConfig {} |

## **🔹 When to Use** @Configuration**?**

✅ When you **need to define multiple beans** manually.  
✅ When you want to **centralize Spring configurations**.  
✅ When integrating **third-party libraries** that require Spring Beans.

### @Component ****in Spring Boot**** 🚀

@Component is a **Spring annotation** used to mark a class as a **Spring-managed bean**. This means **Spring will automatically detect and register** the class in the **Spring ApplicationContext**.

## **🔹 Why Use** @Component**?**

✅ Automatically detects and registers beans.  
✅ Reduces the need for manual bean configuration.  
✅ Works with **dependency injection** (@Autowired).

## **🔹 Example: Using** @Component

📌 **Creating a Simple Component**

java

import org.springframework.stereotype.Component;

@Component

public class MyComponent {

public String sayHello() {

return "Hello from MyComponent!";

}

}

📌 **Using it in Another Class (Dependency Injection)**

java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class MyService {

@Autowired

private MyComponent myComponent;

public String getMessage() {

return myComponent.sayHello();

}

}

## **🔹**@Component **vs** @Service **vs** @Repository

Although all three **register a class as a Spring Bean**, they serve different purposes.

| **Annotation** | **Purpose** |
| --- | --- |
| @Component | Generic component (can be anything). |
| @Service | Business logic (service layer). |
| @Repository | Data access layer (DAO, database operations). |

📌 **Example:**

java

@Component // General component

class UtilityClass { }

@Service // Business logic

class UserService { }

@Repository // Database operations

class UserRepository { }

## **🔹 How** @ComponentScan **Works**

By default, Spring scans the **package where the main class (@SpringBootApplication) is located**.  
If your components are in **a different package**, you need to **manually specify the package** using @ComponentScan.

📌 **Example: Custom Component Scan**

java

@ComponentScan(basePackages = "com.example.mycomponents")

@SpringBootApplication

public class MyApp { }

## **🔹 Summary**

✔ **@Component** registers a class as a **Spring Bean**.  
✔ Used for **generic components** (utilities, helpers, etc.).  
✔ Works with **dependency injection** (@Autowired).  
✔ @Service and @Repository are **specialized** versions of @Component.

# @Transactional **in Spring Boot - In-Depth Guide** 🚀

### ****What is**** @Transactional****?****

@Transactional is a Spring Boot annotation that **manages database transactions** automatically. It ensures that a method executes within a **transactional boundary**, meaning:

✅ If everything succeeds → **Changes are committed**  
❌ If an exception occurs → **All changes are rolled back**

## **🔹 Why Do We Need** @Transactional**?**

Without transactions, partial updates can lead to **data inconsistency**.

📌 **Example of Inconsistent Data Without @Transactional**

public void createUser(User user, Address address) {

userRepository.save(user); // ✅ User is saved

addressRepository.save(address); // ❌ Fails, but User is already saved

}

🔴 **Problem:** If addressRepository.save(address) fails, the user is still saved → **Inconsistent Data!**

📌 **With @Transactional, this issue is prevented** ✅

@Transactional

public void createUser(User user, Address address) {

userRepository.save(user);

addressRepository.save(address);

}

✔ **If an exception occurs, both operations are rolled back.**

# **1️⃣ How** @Transactional **Works**

When a method is annotated with @Transactional, Spring:  
🔹 Starts a **new transaction** before method execution.  
🔹 Executes the method inside a **transactional scope**.  
🔹 **Commits** changes if successful, or **rolls back** if an exception occurs.

📌 **Example: @Transactional in Service Layer**

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

@Autowired

private AddressRepository addressRepository;

@Transactional

public void createUser(User user, Address address) {

userRepository.save(user);

addressRepository.save(address);

}

}

✔ **If any operation fails, both user and address are rolled back**.

# **2️⃣ Transaction Rollback Behavior**

By default, **Spring rolls back only on RuntimeException (unchecked exceptions)**.

📌 **Example: @Transactional Default Behavior**

@Transactional

public void createUser(User user, Address address) {

userRepository.save(user);

throw new RuntimeException("Something went wrong!"); // ✅ Will trigger rollback

}

✔ **Rollback happens because it's a RuntimeException**.

## **🔹 Rollback on Checked Exceptions (**Exception**)**

Spring **does NOT** roll back for Exception (checked exception) by default.

📌 **Example: No Rollback on Exception (Checked Exception)**

@Transactional

public void createUser(User user, Address address) throws Exception {

userRepository.save(user);

throw new Exception("Something went wrong!"); // ❌ Won't trigger rollback

}

✔ **User is still saved because Exception is a checked exception!**

📌 **Solution: Force rollback for Exception using rollbackFor**

@Transactional(rollbackFor = Exception.class)

public void createUser(User user, Address address) throws Exception {

userRepository.save(user);

throw new Exception("Something went wrong!"); // ✅ Will trigger rollback

}

✔ **Now rollback happens for both RuntimeException and Exception.** ✅

# **3️⃣ Propagation in Transactions**

Spring provides **different transaction propagation strategies** to define how transactions behave when calling other transactional methods.

### ****🔹 Propagation Types****

| **Propagation Type** | **Behavior** |
| --- | --- |
| REQUIRED (Default) | Uses an existing transaction or creates a new one if none exists. |
| REQUIRES\_NEW | Always creates a **new** transaction. |
| SUPPORTS | Runs within a transaction **if one exists**, else runs normally. |
| NOT\_SUPPORTED | Runs **without** a transaction. |
| MANDATORY | Requires an existing transaction (throws an error if none exists). |
| NEVER | Throws an error if called within a transaction. |
| NESTED | Creates a **nested transaction** within the main transaction. |

📌 **Example: REQUIRES\_NEW (Independent Transaction)**

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void logUserActivity(User user) {

activityRepository.save(user.getActivity());

}

✔ **Even if the main transaction fails, logs are still saved.** ✅

## **4️⃣ Transaction Isolation Levels**

Isolation levels prevent **dirty reads, non-repeatable reads, and phantom reads**.

| **Isolation Level** | **Description** |
| --- | --- |
| READ\_UNCOMMITTED | Allows dirty reads (read uncommitted changes). |
| READ\_COMMITTED (Default) | Prevents dirty reads, but allows non-repeatable reads. |
| REPEATABLE\_READ | Prevents dirty and non-repeatable reads, but allows phantom reads. |
| SERIALIZABLE | Strictest level (no dirty, non-repeatable, or phantom reads). |

📌 **Example: Setting Isolation Level**

@Transactional(isolation = Isolation.SERIALIZABLE)

public void updateAccountBalance(Account account) {

accountRepository.save(account);

}

✔ **Ensures strict transaction control to avoid conflicts.** ✅

## **5️⃣ Read-Only Transactions**

If a method only **fetches data (no updates),** use @Transactional(readOnly = true) for better performance.

📌 **Example: Read-Only Transaction**

@Transactional(readOnly = true)

public List<User> getAllUsers() {

return userRepository.findAll();

}

✔ **Improves performance by optimizing database locks**. ✅

## **6️⃣** @Transactional **on Different Layers**

| **Layer** | **@Transactional Usage** |
| --- | --- |
| **Controller (@RestController)** | ❌ Not recommended |
| **Service (@Service)** | ✅ Recommended |
| **Repository (@Repository)** | ✅ (Already transactional by Spring Data JPA) |

📌 **Best Practice:** Apply @Transactional at the **Service layer**, NOT in the Controller.

# **🛠 Summary**

✔ **Ensures all database operations in a method succeed or fail together.**  
✔ **Automatically rolls back on RuntimeException, but not on Exception (unless configured).**  
✔ **Supports different propagation types (REQUIRED, REQUIRES\_NEW, etc.).**  
✔ **Can be used with different isolation levels (READ\_COMMITTED, SERIALIZABLE, etc.).**  
✔ **Use @Transactional(readOnly = true) for read-only queries to optimize performance.**

The difference between **using JPA only** and **using JPA with Hibernate** lies in the role of **JPA (Java Persistence API)** and **Hibernate** in handling database operations. Let’s break it down clearly:

## **1️⃣ JPA Only (Java Persistence API)**

* **JPA is just a specification**, meaning it provides a set of **interfaces** (like EntityManager, Query, etc.) but does not provide an implementation.
* You need an **implementation** of JPA (such as Hibernate, EclipseLink, or OpenJPA) to actually perform persistence operations.
* If you use **JPA alone**, you have to configure and provide an implementation manually.

### ****Example of JPA Without Hibernate****

java

import jakarta.persistence.\*;

@Entity

@Table(name = "products")

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private double price;

}

* Here, we use **pure JPA annotations**, but this won’t work unless we provide an **implementation like Hibernate**.

## **2️⃣ JPA with Hibernate**

* Hibernate is the **most commonly used implementation** of JPA.
* JPA provides an **abstracted API**, while Hibernate provides the **actual ORM functionality**.
* Spring Boot uses Hibernate **by default** when you include spring-boot-starter-data-jpa.

### ****Example of JPA + Hibernate****

java

import jakarta.persistence.\*;

import lombok.\*;

@Entity

@Table(name = "products")

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private double price;

}

* This is similar to the **JPA-only example**, but **Spring Boot automatically picks Hibernate** as the implementation.

## **🔍 Key Differences**

| **Feature** | **JPA Only** | **JPA with Hibernate** |
| --- | --- | --- |
| **Definition** | API specification for ORM | Implementation of JPA |
| **Provides ORM?** | No, just an API | Yes, full ORM features |
| **Implementation** | Needs a provider like Hibernate | Hibernate is used as JPA provider |
| **Persistence API** | Uses EntityManager | Uses EntityManager via Hibernate |
| **Query Language** | Uses JPQL | Supports JPQL & HQL |
| **Caching** | No built-in caching | Has first-level caching |
| **Performance** | Depends on implementation | Optimized ORM performance |

## **3️⃣ When to Use What?**

* If you're **just learning JPA**, you can try using it **without an implementation** to understand the API.
* In a **real-world Spring Boot application**, **JPA + Hibernate** is **recommended** because Hibernate provides:
  + Caching
  + Lazy loading
  + Advanced query features
  + Performance optimizations

### ****Conclusion****

JPA is like a **universal remote** 🕹️, while Hibernate is a **TV that actually works with the remote** 📺.

So, in most cases, you **don’t use JPA alone**—you **use JPA with Hibernate** to actually interact with the database efficiently. 🚀

## **✅ Summary**

🔹 **JPA provides the API**, but without an implementation, it won't work.  
🔹 **Hibernate is removed, and EclipseLink is used instead** as the JPA provider.  
🔹 **Spring Data JPA still works** but now interacts with EclipseLink instead of Hibernate.

When people say **"JPA helps to change the database in Spring"**, they mean that **JPA (Java Persistence API) abstracts database interactions**, allowing developers to switch databases with minimal or no changes to the code.

### 🔹 ****How Does JPA Enable Database Switching?****

JPA works as an **abstraction layer** between the application and the underlying database by using **ORM (Object-Relational Mapping)** frameworks like **Hibernate**. Instead of writing database-specific queries, developers use **JPA annotations and repository methods**, making it **independent of the database**.

### ✅ ****Process of Changing the Database in Spring Boot with JPA****

#### **Step 1: Use JPA Instead of Raw SQL**

* JPA allows developers to use **Entity classes** and **repositories** instead of writing raw SQL.
* Example: Instead of writing database-specific queries like SELECT \* FROM users;, you use JPA like this:

java

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

}

java

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

}

This code **works regardless of the database type** (H2, MySQL, PostgreSQL, etc.).

#### **Step 2: Use a Database-Specific Dialect in** application.properties

Spring Boot allows changing databases by modifying properties.

🔹 **For H2 (In-memory database for testing)**

properties

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

🔹 **For MySQL**

properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

🔹 **For PostgreSQL**

properties

spring.datasource.url=jdbc:postgresql://localhost:5432/mydb

spring.datasource.username=postgres

spring.datasource.password=root

spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

**🔄 To switch databases**, you only need to change the spring.datasource.\* and spring.jpa.database-platform properties. The **code remains unchanged**.

#### **Step 3: Spring Boot Automatically Adapts**

* Since JPA manages entity relationships and Hibernate translates them into database-specific SQL, no code changes are required.
* Example:
  + **If using MySQL**, Hibernate will generate MySQL-compatible SQL queries.
  + **If using PostgreSQL**, Hibernate will generate PostgreSQL-compatible queries.

### 🎯 ****Key Benefits of Using JPA for Database Switching****

1. **No need to modify queries** – JPA translates the queries for the new database.
2. **Portability** – The same code works with multiple databases.
3. **Flexibility** – Change databases just by modifying configuration files.
4. **Less vendor lock-in** – You are not tied to a single database provider.

### ****🔚 Conclusion****

JPA helps switch databases **by abstracting SQL queries and using database-specific dialects**. Instead of writing database-dependent code, you define **entities and repositories**, and Spring Boot automatically adjusts based on the **configured database**

### ****Which Imports Should You Prefer: JPA or Hibernate?****

When working with **Spring Boot and database persistence**, the general recommendation is to **prefer JPA imports (javax.persistence.\* or jakarta.persistence.\*) over Hibernate imports (org.hibernate.\*)**.

### ✅ ****Prefer JPA Imports (****javax.persistence.\* ****or**** jakarta.persistence.\*****)****

* **JPA is a standard** (Java Persistence API) and is vendor-independent.
* **It provides flexibility** to switch between different ORM frameworks (like Hibernate, EclipseLink, TopLink, etc.).
* **Spring Boot and Spring Data JPA work best with JPA annotations**.

#### **Example using JPA imports:**

java

import jakarta.persistence.\*;

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false)

private String name;

@Column(unique = true, nullable = false)

private String email;

}

* This code works with **any JPA-compliant provider**, not just Hibernate.
* In **Spring Boot 3+, javax.persistence is replaced with jakarta.persistence**.

### ❌ ****Avoid Direct Hibernate Imports (****org.hibernate.\*****) Unless Necessary****

* Hibernate is an **implementation** of JPA, not a standard.
* Using org.hibernate.\* makes your code **tied to Hibernate**, making it harder to switch to another ORM provider.
* **Exception**: Some advanced Hibernate features (like @BatchSize, @LazyCollection, Session) require direct Hibernate imports.

#### **Example using Hibernate-specific imports (Not recommended unless necessary)**

java

import org.hibernate.annotations.BatchSize;

import org.hibernate.annotations.LazyCollection;

import org.hibernate.annotations.LazyCollectionOption;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false)

private String name;

@BatchSize(size = 10)

@LazyCollection(LazyCollectionOption.EXTRA)

private List<Order> orders;

}

* This will **only work with Hibernate** and may cause issues if switching to another JPA provider.

### 🎯 ****When to Use Hibernate Imports?****

✅ Use JPA imports **for standard ORM features** (annotations like @Entity, @Table, @Column, @ManyToOne, etc.).  
✅ Use Hibernate imports **only for Hibernate-specific optimizations** (e.g., @BatchSize, @LazyCollection, Session).

### ****Conclusion****

* **For general use:** ✅ **Prefer JPA (jakarta.persistence.\*)**
* **For Hibernate-specific features:** ⚠️ **Use org.hibernate.\* only when needed**
* **For maximum flexibility:** ✅ **Write JPA-compliant code to make switching ORM providers easier.**

## **1. ApplicationContextAware**

**Purpose:** Allows a bean to access the ApplicationContext, which manages all the beans in the Spring container.

### ****Example:****

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

import org.springframework.stereotype.Component;

@Component

public class MyApplicationContextAware implements ApplicationContextAware {

private static ApplicationContext context;

@Override

public void setApplicationContext(ApplicationContext applicationContext) {

context = applicationContext;

}

public static <T> T getBean(Class<T> beanClass) {

return context.getBean(beanClass);

}

}

🔹 **Use case**: This can be used to fetch any bean dynamically by class type.

## **2. BeanNameAware**

**Purpose:** Allows a bean to be aware of its own name within the Spring container.

### ****Example:****

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.stereotype.Component;

@Component

public class MyBeanNameAware implements BeanNameAware {

private String beanName;

@Override

public void setBeanName(String name) {

this.beanName = name;

}

public String getBeanName() {

return beanName;

}

}

🔹 **Use case**: Useful when debugging or logging the bean’s name in a large application.

## **3. BeanFactoryAware**

**Purpose:** Allows a bean to access the BeanFactory, which is responsible for creating and managing beans.

### ****Example:****

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.BeanFactoryAware;

import org.springframework.stereotype.Component;

@Component

public class MyBeanFactoryAware implements BeanFactoryAware {

private static BeanFactory beanFactory;

@Override

public void setBeanFactory(BeanFactory factory) {

beanFactory = factory;

}

public static <T> T getBean(Class<T> beanClass) {

return beanFactory.getBean(beanClass);

}}

🔹 **Use case**: Allows retrieving beans lazily when needed.

## **4. EnvironmentAware**

**Purpose:** Allows a bean to access the environment variables and properties.

### ****Example:****

java

import org.springframework.context.EnvironmentAware;

import org.springframework.core.env.Environment;

import org.springframework.stereotype.Component;

@Component

public class MyEnvironmentAware implements EnvironmentAware {

private static Environment environment;

@Override

public void setEnvironment(Environment env) {

environment = env;

}

public static String getProperty(String key) {

return environment.getProperty(key);

}

}

🔹 **Use case**: Fetching configuration properties dynamically.

## **5. MessageSourceAware**

**Purpose:** Allows a bean to access internationalization (i18n) messages.

### ****Example:****

java

import org.springframework.context.MessageSource;

import org.springframework.context.MessageSourceAware;

import org.springframework.stereotype.Component;

import java.util.Locale;

@Component

public class MyMessageSourceAware implements MessageSourceAware {

private static MessageSource messageSource;

@Override

public void setMessageSource(MessageSource source) {

messageSource = source;

}

public static String getMessage(String key, Object[] args, Locale locale) {

return messageSource.getMessage(key, args, locale);

}

}

🔹 **Use case**: Useful in applications with multiple language support.

## **6. ResourceLoaderAware**

**Purpose:** Allows a bean to load resources such as files, images, etc.

### ****Example:****

java

import org.springframework.context.ResourceLoaderAware;

import org.springframework.core.io.Resource;

import org.springframework.core.io.ResourceLoader;

import org.springframework.stereotype.Component;

@Component

public class MyResourceLoaderAware implements ResourceLoaderAware {

private static ResourceLoader resourceLoader;

@Override

public void setResourceLoader(ResourceLoader loader) {

resourceLoader = loader;

}

public static Resource getResource(String location) {

return resourceLoader.getResource(location);

}

}

🔹 **Use case**: Reading external files dynamically.

## **Conclusion**

Each of these interfaces helps a bean interact with the Spring container:

| **Aware Interface** | **Purpose** |
| --- | --- |
| ApplicationContextAware | Access to ApplicationContext (fetch beans dynamically) |
| BeanNameAware | Access the bean name |
| BeanFactoryAware | Access BeanFactory (manage beans programmatically) |
| EnvironmentAware | Access environment variables & properties |
| MessageSourceAware | Access message sources for internationalization |
| ResourceLoaderAware | Load resources dynamically |

The annotations @Component, @Service, and @Repository are all used for defining Spring-managed beans, but they serve different purposes based on their intended use in an application.

### ****1.**** @Component ****(Generic Bean)****

* **Purpose:** A generic annotation used to mark any Spring-managed bean.
* **Usage:** It is the parent annotation of @Service and @Repository.
* **Example:**

java

@Component

public class MyComponent {

public void doSomething() {

System.out.println("Doing something...");

}

}

### ****2.**** @Service ****(Business Logic Layer)****

* **Purpose:** A specialized form of @Component used to define service-layer beans.
* **Usage:** It is used for business logic and service-related operations.
* **Example:**

java

@Service

public class UserService {

public String getUser() {

return "User details";

}

}

### ****3.**** @Repository ****(Data Access Layer)****

* **Purpose:** A specialization of @Component used to define DAO (Data Access Object) beans.
* **Usage:** It is used for database interaction and includes automatic exception translation in Spring Data JPA.
* **Example:**

java

@Repository

public class UserRepository {

public String getUserFromDB() {

return "Fetching user from DB";

}

}

### ****Key Differences:****

| **Annotation** | **Purpose** | **Layer Used** |
| --- | --- | --- |
| @Component | Generic Spring-managed bean | Any Layer |
| @Service | Business logic processing | Service Layer |
| @Repository | Database interaction, exception handling | DAO Layer |

Though functionally they are similar, using the appropriate annotation improves code readability and maintainability. 🚀

### ****How Does Spring Boot Auto-Configuration Work?****

Spring Boot **auto-configuration** automatically configures application beans based on the classpath dependencies and environment settings. It eliminates the need for manual bean configuration by detecting available libraries and providing sensible defaults.

#### **Key Components of Auto-Configuration:**

1. **@SpringBootApplication**
   * This annotation includes @EnableAutoConfiguration, which triggers auto-configuration.
   * Example: @SpringBootApplication

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

1. **Spring Boot Starters**
   * If a dependency (e.g., spring-boot-starter-web) is found, Spring Boot configures necessary beans (e.g., DispatcherServlet, Jackson, etc.).
2. **@ConditionalOnMissingBean & @ConditionalOnClass**
   * Auto-configurations define conditions for loading beans based on the presence of certain classes or missing beans.
   * Example from Spring Boot: @Configuration

@ConditionalOnClass(DataSource.class)

public class DataSourceAutoConfiguration {

@Bean

public DataSource dataSource() {

return new HikariDataSource();

}

}

1. **Auto-Configuration Classes**
   * Defined in META-INF/spring.factories (Spring Boot <3) or META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports (Spring Boot 3+).

### ****How to Disable Auto-Configuration?****

You can disable auto-configuration in multiple ways:

#### **1. Using** @SpringBootApplication(exclude = ...)

* Example: Disabling DataSourceAutoConfiguration
* SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class MyApplication { }

#### **2. Using** @EnableAutoConfiguration(exclude = ...)

* Example:

java

@EnableAutoConfiguration(exclude = {JpaRepositoriesAutoConfiguration.class})

public class MyApplication { }

#### **3. Using** spring.autoconfigure.exclude **in** application.properties

* Example:

properties

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

#### **4. Excluding Dependencies in** spring.factories **(Spring Boot <3)**

* Example (META-INF/spring.factories):

org.springframework.boot.autoconfigure.EnableAutoConfiguration=

com.example.MyCustomAutoConfiguration

By disabling unnecessary auto-configurations, you can improve application performance and avoid conflicts. 🚀

### ****How to Create a Custom Starter in Spring Boot?****

A **Spring Boot Starter** is a set of pre-configured dependencies that help developers easily integrate specific functionalities. Creating a **custom starter** involves the following steps:

### ****Step 1: Create a New Maven Project for the Starter****

Create a separate Maven/Gradle module for the starter package.

#### **Example:** custom-starter

📂 **Project Structure**

custom-starter/

├── src/main/java/com/example/autoconfig/

│ ├── MyService.java

│ ├── MyServiceAutoConfiguration.java

├── src/main/resources/

│ ├── META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports (Spring Boot 3+)

├── pom.xml

### ****Step 2: Create the Main Service****

Define the functionality provided by the starter.

📌 **Example: MyService.java**

package com.example.autoconfig;

public class MyService {

public String greet() {

return "Hello from MyService!";

}

}

### ****Step 3: Create an Auto-Configuration Class****

This class ensures that MyService is auto-configured when the starter is added as a dependency.

📌 **Example: MyServiceAutoConfiguration.java**

java

package com.example.autoconfig;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class MyServiceAutoConfiguration {

@Bean

public MyService myService() {

return new MyService();

}

}

### ****Step 4: Register the Auto-Configuration****

For **Spring Boot 3+**, use org.springframework.boot.autoconfigure.AutoConfiguration.imports instead of spring.factories.

📌 **File: META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports**

com.example.autoconfig.MyServiceAutoConfiguration

✅ **For Spring Boot 2.x or lower**, use META-INF/spring.factories:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.example.autoconfig.MyServiceAutoConfiguration

### ****Step 5: Package and Publish the Starter****

* Build the project using:

bash

mvn clean install

* Publish the JAR to a Maven repository (local or remote).

### ****Step 6: Use the Starter in Another Project****

In a separate Spring Boot application, add the custom starter as a dependency.

📌 **Example: pom.xml**

xml

<dependency>

<groupId>com.example</groupId>

<artifactId>custom-starter</artifactId>

<version>1.0.0</version>

</dependency>

📌 **Use the auto-configured service:**

java

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import com.example.autoconfig.MyService;

@Component

public class MyRunner implements CommandLineRunner {

private final MyService myService;

public MyRunner(MyService myService) {

this.myService = myService;

}

@Override

public void run(String... args) throws Exception {

System.out.println(myService.greet());

}

}

### ****Summary****

| **Step** | **Action** |
| --- | --- |
| 1 | Create a new Maven module |
| 2 | Implement the functionality (MyService.java) |
| 3 | Define an auto-configuration class |
| 4 | Register auto-configuration (org.springframework.boot.autoconfigure.AutoConfiguration.imports) |
| 5 | Package and install the starter (mvn install) |
| 6 | Add the starter as a dependency in a Spring Boot project |

Now, whenever you include the custom starter in a project, it will automatically configure MyService.

### ****Difference Between**** @Bean ****and**** @Component ****in Spring Boot****

Both @Bean and @Component are used to define Spring-managed beans, but they work in different ways and are used in different scenarios.

### ****1.**** @Component ****(Class-Level Annotation)****

* **Purpose:** It is a class-level annotation used to mark a class as a Spring-managed bean.
* **Usage:** Typically used for service, repository, or component classes.
* **Component Scanning:** Automatically detected and registered if the package is scanned.
* **Example:**

java

@Component

public class MyComponent {

public void sayHello() {

System.out.println("Hello from MyComponent");

}

}

**Spring automatically registers @Component classes if they are inside a package scanned by @ComponentScan.**

### ****2.**** @Bean ****(Method-Level Annotation)****

* **Purpose:** Declares a bean manually inside a configuration class.
* **Usage:** Used when defining third-party beans or when more control over bean creation is needed.
* **Explicit Declaration:** Not automatically detected; must be inside a @Configuration-annotated class.
* **Example:**

java

@Configuration

public class AppConfig {

@Bean

public MyComponent myComponent() {

return new MyComponent();

}

}

**Beans declared with @Bean are not automatically discovered by Spring unless explicitly defined inside a @Configuration class.**

### ****Key Differences****

| **Feature** | **@Component** | **@Bean** |
| --- | --- | --- |
| **Annotation Type** | Class-level | Method-level |
| **Component Scanning** | Automatically detected | Needs explicit declaration |
| **Use Case** | Used for application components like @Service, @Repository, etc. | Used for third-party classes or complex bean initialization |
| **Definition Location** | Directly on a class | Inside a @Configuration class |
| **Customization** | Less flexible | More control over bean creation |

### ****When to Use What?****

✅ Use @Component when creating standard Spring components like services, repositories, and controllers.  
✅ Use @Bean when you need manual bean creation, such as for external libraries or factory methods.

### ****Different Scopes Available in Spring Boot****

Spring provides different **bean scopes** to control the lifecycle and visibility of beans in a Spring application. The scope determines how and when a new bean instance is created.

### ****1. Singleton Scope (****@Scope("singleton")****)****

* **Default scope in Spring** (if no scope is defined).
* A **single instance** of the bean is created and shared across the entire application context.
* The same instance is used every time the bean is injected.

📌 **Example:**

@Component

@Scope("singleton")

public class SingletonBean {

public SingletonBean() {

System.out.println("SingletonBean instance created");

}

}

🔹 **When to Use?**

* For stateless services or utility classes.

### ****2. Prototype Scope (****@Scope("prototype")****)****

* A new bean instance is created **every time** it is requested.
* Each injection point gets a different instance of the bean.

📌 **Example:**

@Component

@Scope("prototype")

public class PrototypeBean {

public PrototypeBean() {

System.out.println("PrototypeBean instance created");

}

}

🔹 **When to Use?**

* For beans that maintain state and should not be shared.

### ****3. Request Scope (****@Scope("request")****)**** (Only for Web Applications)

* A new bean instance is created **for each HTTP request**.
* Used in web applications to manage per-request state.

📌 **Example:**

@Component

@Scope("request")

public class RequestBean {

public RequestBean() {

System.out.println("RequestBean instance created");

}

}

🔹 **When to Use?**

* For handling request-specific data in web applications.

### ****4. Session Scope (****@Scope("session")****)**** (Only for Web Applications)

* A single instance is created **for each HTTP session**.
* Used to store user session-related data.

📌 **Example:**

@Component

@Scope("session")

public class SessionBean {

public SessionBean() {

System.out.println("SessionBean instance created");

}

}

🔹 **When to Use?** For managing user session information.

### ****5. Application Scope (****@Scope("application")****)**** (Only for Web Applications)

* A single instance is created and **shared across the entire application** (same as Singleton but limited to web applications).

📌 **Example:**

java

@Component

@Scope("application")

public class ApplicationBean {

public ApplicationBean() {

System.out.println("ApplicationBean instance created");

}

}

🔹 **When to Use?**

* For application-wide shared data.

### ****6. WebSocket Scope (****@Scope("websocket")****)**** (Only for WebSocket-based Apps)

* A single instance is created and shared for **each WebSocket session**.

📌 **Example:**

java

@Component

@Scope("websocket")

public class WebSocketBean {

public WebSocketBean() {

System.out.println("WebSocketBean instance created");

}

}

🔹 **When to Use?**

* For WebSocket-based applications where you need session-based data.

### ****Comparison of Scopes****

| **Scope** | **Bean Lifecycle** | **Use Case** |
| --- | --- | --- |
| **Singleton** | One instance per application context | Shared stateless service beans |
| **Prototype** | New instance every time requested | Stateful beans |
| **Request** | New instance per HTTP request | Web request-specific beans |
| **Session** | One instance per HTTP session | Session-based user data |
| **Application** | One instance per application | Application-wide shared beans |
| **WebSocket** | One instance per WebSocket session | WebSocket-based applications |

### ****Conclusion****

* **Use singleton (default) for stateless services.**
* **Use prototype for beans that need a new instance every time.**
* **Use request, session, and application for web applications.**
* **Use websocket for WebSocket-based apps.**

## **Interceptor in Spring Boot**

### ****📌 What is an Interceptor in Spring Boot?****

In **Spring Boot**, an **Interceptor** is used to **process HTTP requests before and after** they reach the controller. It acts like a **middleware** that intercepts requests to perform logging, authentication, or modifications.

## **1️⃣ Interceptor vs. Filter vs. AOP**

| **Feature** | **Interceptor** | **Filter** | **AOP (Aspect-Oriented Programming)** |
| --- | --- | --- | --- |
| **Position** | Between DispatcherServlet & Controller | Before DispatcherServlet | Works at method level |
| **Scope** | Works on specific routes | Works on all requests | Works on business logic |
| **Use Case** | Logging, authentication, request modification | Security, CORS, character encoding | Transaction management, performance monitoring |

## **2️⃣ How to Create an Interceptor?**

A Spring Boot **interceptor** implements HandlerInterceptor and overrides its methods.

### ****🛠 Key Methods in**** HandlerInterceptor

| **Method** | **When it Runs** | **Return Type** |
| --- | --- | --- |
| preHandle() | Before controller execution | true (continue) or false (stop request) |
| postHandle() | After controller execution but before response | void |
| afterCompletion() | After response is sent to the client | void |

### ****3️⃣ Implementing a Custom Interceptor****

Let's create a simple logging interceptor in Spring Boot.

### ****Step 1: Create the Interceptor Class****

package com.example.interceptor;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

public class RequestInterceptor implements HandlerInterceptor {

// Runs before the controller method

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

System.out.println("🚀 Pre Handle: Request URL - " + request.getRequestURL());

return true; // Continue request processing

}

// Runs after the controller method but before the response

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

System.out.println("✅ Post Handle: Controller executed");

}

// Runs after the response is sent

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

System.out.println("🎯 After Completion: Response sent to client");

}}

### ****Step 2: Register the Interceptor****

To use the interceptor, we need to **register it** in a configuration class.

java

package com.example.interceptor;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.InterceptorRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

public class InterceptorConfig implements WebMvcConfigurer {

@Override

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(new RequestInterceptor());

}

}

### ****Step 3: Create a Simple Controller****

This controller will allow us to test our interceptor.

java

package com.example.interceptor;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@GetMapping("/hello")

public String sayHello(@RequestParam(defaultValue = "World") String name) {

return "Hello, " + name + "!";

}

}

### ****Step 4: Run and Test the Application****

Start your Spring Boot application and hit:

bash

http://localhost:8080/hello?name=Devendra

### ****📌 Expected Console Output****

yaml

🚀 Pre Handle: Request URL - http://localhost:8080/hello?name=Devendra

✅ Post Handle: Controller executed

🎯 After Completion: Response sent to client

### ****📌 Expected Response****

json

Hello, Devendra!

## **Advanced Interceptor Use Cases**

### ✅ ****Authentication & Authorization****

java

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

String authHeader = request.getHeader("Authorization");

if (authHeader == null || !authHeader.startsWith("Bearer ")) {

response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

return false;

}

return true; // Continue only if authorized

}

### ✅ ****Logging Request Data****

java

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

System.out.println("Method: " + request.getMethod());

System.out.println("Headers: " + request.getHeaderNames());

return true;

}

### ✅ ****Modify Response Before Sending****

java

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

response.addHeader("Custom-Header", "Interceptor-Added");

}

## **5️⃣ Conclusion**

| **Feature** | **Interceptor Benefits** |
| --- | --- |
| **Flexibility** | Can intercept requests before and after execution |
| **Security** | Used for authentication & authorization |
| **Logging** | Captures request details for debugging |
| **Performance** | Measures request execution time |

💡 **Interceptors are best for handling cross-cutting concerns like logging, security, and monitoring.** 🚀