# Intermediate Interview Questions

**1. How do you handle exceptions in Spring Boot?**

**Why?** Exception handling is crucial for maintaining application stability and providing meaningful feedback to clients. It ensures the application doesn’t crash and helps in debugging issues.

**How?** In Spring Boot, exceptions can be handled using:

* **@ExceptionHandler**: Handles exceptions at the controller level.
* **@ControllerAdvice**: Provides global exception handling across multiple controllers.
* **ResponseStatusException**: Throws exceptions with specific HTTP status codes.

**Example:** In a **User Management System**, I used @ControllerAdvice to handle user-not-found exceptions globally.

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleGenericException(Exception ex) {

return new ResponseEntity<>("An error occurred: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

In the controller, a custom exception is thrown when a user is not found:

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

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

User user = userService.getUserById(id);

if (user == null) {

throw new UserNotFoundException("User with ID " + id + " not found.");

}

return ResponseEntity.ok(user);

}

}

🔹 **Real-Life Use Case:** In an **e-commerce application**, if a product is not found, instead of displaying a blank page or a generic error, the application returns a 404 status with "Product not found." This enhances the user experience and simplifies debugging for developers.

**2. What is the difference between @Component, @Service, and @Repository annotations?**

**Why?** These annotations define Spring-managed beans but serve different purposes.

| **Annotation** | **Purpose** | **Where Used?** |
| --- | --- | --- |
| **@Component** | Generic stereotype for Spring-managed components | General-purpose classes (e.g., utility classes) |
| **@Service** | Indicates business logic service layer | Service classes (e.g., UserService, ProductService) |
| **@Repository** | Indicates data access layer (DAO) | Repository classes (e.g., UserRepository, ProductRepository) |

**Example:** In a **Banking Application**, these annotations are used as follows:

@Component

public class EmailSender {

public void sendEmail(String to, String message) {

System.out.println("Email sent to " + to);

}

}

@Service

public class TransactionService {

@Autowired

private TransactionRepository transactionRepository;

public List<Transaction> getTransactionsByAccountId(Long accountId) {

return transactionRepository.findByAccountId(accountId);

}

}

@Repository

public interface TransactionRepository extends JpaRepository<Transaction, Long> {

List<Transaction> findByAccountId(Long accountId);

}

🔹 **Real-Life Use Case:** In a **banking app**, @Repository manages database operations, @Service contains business logic, and @Component is used for utilities like sending email notifications.

**3. How do you connect a Spring Boot application to a database using JPA?**

**Why?** JPA simplifies database interactions by using object-relational mapping (ORM), eliminating the need for raw SQL queries.

**Steps:**

1. Add dependencies to pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

1. Configure database settings in application.properties:

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

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

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

1. Create an entity class:

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private Double price;

}

1. Create a repository interface:

@Repository

public interface ProductRepository extends JpaRepository<Product, Long> {

}

🔹 **Real-Life Use Case:** In a **blogging platform**, JPA is used to manage user data and blog posts, mapping entities to database tables and performing CRUD operations through repositories.

**4. What is the purpose of the @Entity annotation in Spring Boot?**

**Why?** The @Entity annotation marks a class as a JPA entity, mapping it to a database table.

**Example:** In an **e-commerce application**, a Product entity is defined as:

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private Double price;

}

🔹 **Real-Life Use Case:** In an **inventory management system**, Product represents a table storing details like name, price, and quantity.

**5. How do you implement caching in Spring Boot?**

**Why?** Caching improves performance by reducing database queries for frequently accessed data.

**How?**

1. Add the caching dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

1. Enable caching in the main application class:

@SpringBootApplication

@EnableCaching

public class MyApp {

}

1. Use @Cacheable to cache method results:

@Service

public class ProductService {

@Cacheable("products")

public Product getProductById(Long id) {

return productRepository.findById(id).orElse(null);

}

}

🔹 **Real-Life Use Case:** In an **e-commerce site**, caching is used to store product details, reducing database queries when multiple users access the same product page.

**6. What is the difference between @RequestParam and @PathVariable?**

| **Annotation** | **Used For** | **Example** |
| --- | --- | --- |
| **@RequestParam** | Extracts query parameters | /users?id=10 |
| **@PathVariable** | Extracts values from the URI path | /users/10 |

**Example:**

@GetMapping("/users")

public String getUserById(@RequestParam Long id) {

return "User ID: " + id;

}

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

public String getUserById(@PathVariable Long id) {

return "User ID: " + id;

}

🔹 **Real-Life Use Case:** @RequestParam is used for filtering users by role (e.g., /users?role=admin), while @PathVariable fetches a specific user by ID (/users/10).

**8. What is the purpose of the @Autowired annotation, and how does it work?**

**Why?** The @Autowired annotation is used for dependency injection, allowing Spring to automatically inject beans into your class.

**Example:**  
In a **Payment Service**, I used @Autowired to inject a PaymentGateway bean:

java

Copy

@Service

public class PaymentService {

@Autowired

private PaymentGateway paymentGateway;

public void processPayment(Payment payment) {

paymentGateway.process(payment);

}

}

🔹 **Real-Life Use Case:** In an **e-commerce app**, I used @Autowired to inject services like PaymentService and OrderService into controllers.

**9. How do you implement logging in Spring Boot?**

**Why?** Logging helps in tracking application events, debugging issues, and monitoring the application.

**How?**

1. Use SLF4J for logging:

java

Copy

private static final Logger logger = LoggerFactory.getLogger(MyClass.class);

logger.info("User logged in successfully");

logger.error("An error occurred");

🔹 **Real-Life Use Case:** In an **online store**, I used logging to track payment failures and user login attempts. This helped in identifying and resolving issues quickly.

**10. What is the difference between @Configuration and @Component?**

| **Annotation** | **Purpose** |
| --- | --- |
| **@Configuration** | Defines beans explicitly using @Bean methods |
| **@Component** | Marks a generic Spring-managed component |

**Example:**  
In a **database configuration**, I used @Configuration to define a DataSource bean:

java

Copy

@Configuration

public class DatabaseConfig {

@Bean

public DataSource dataSource() {

return new HikariDataSource();

}

}

🔹 **Real-Life Use Case:** In a **microservices architecture**, @Configuration was used to define beans for external service clients, while @Component was used for utility classes.

No, **@PathVariable** and **@RequestParam** do **not** work exactly the same. While both extract values from an API URL, they serve **different purposes** and behave differently.

**Key Differences with Examples**

| **Feature** | **@PathVariable** | **@RequestParam** |
| --- | --- | --- |
| **Where it appears?** | In the URL path (/users/10) | In the query string (/users?id=10) |
| **Mandatory?** | Usually required | Can be optional (set required = false) |
| **Type of parameter?** | Used for identifying a resource | Used for filtering, sorting, or optional values |
| **Multiple values?** | Typically single value (e.g., id) | Can handle multiple values (e.g., ?role=admin&sort=asc) |

**1️⃣ @PathVariable Example – Used for Identifying Resources**

**URL:**

GET /users/10

**Spring Boot Code:**

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

public String getUserById(@PathVariable Long id) {

return "User ID: " + id;

}

✅ **Use Case:**

* When fetching a specific user by ID (/users/10).
* The ID is part of the **URI structure**.

**2️⃣ @RequestParam Example – Used for Filtering and Optional Inputs**

**URL:**

GET /users?role=admin&sort=asc

**Spring Boot Code:**

@GetMapping("/users")

public String getUsers(@RequestParam String role, @RequestParam(defaultValue = "desc") String sort) {

return "Role: " + role + ", Sorting: " + sort;

}

✅ **Use Case:**

* When filtering users by role (?role=admin).
* When passing optional sorting (?sort=asc or default to desc).
* role is **not part of the URL structure**, but an **additional filter**.

**🚀 When to Use What?**

| **Scenario** | **Use @PathVariable** | **Use @RequestParam** |
| --- | --- | --- |
| Fetch a user by ID | ✅ /users/10 | ❌ |
| Search users by role | ❌ | ✅ /users?role=admin |
| Fetch a specific product | ✅ /products/123 | ❌ |
| Apply filters (price, category) | ❌ | ✅ /products?category=electronics&price=1000 |

**🔥 Final Answer:**

No, they do **not** work the same way.

* @PathVariable is used for **resource identification** (fixed part of the URL).
* @RequestParam is used for **optional parameters, filtering, and sorting** (query parameters).