Interview Date -12 July 2025 01:00 pm sat  
Duration - 1 hour

1. Brief Intro

2. What have been your contribution to the project ?

"In my current project for a banking client, we're responsible for sending communications (emails, SMS, push, fax, letters) related to transactions and marketing. Initially, the system supported only SMS and email, but our team extended it to support additional channels like push notifications and physical letters."

"My key contributions included designing and developing REST APIs using Spring Boot, implementing microservices for each communication channel, and integrating with third-party providers for email and SMS delivery. I also worked on implementing retry logic, request validation, and writing JUnit test cases using Mockito."

"I was involved in code reviews, daily Scrum meetings.

3. Agile, have used Jira, how long sprint – 2 weeks how many story point and 1 story point is equal to what ? - 8 hours or 1 day.

4. == and equal difference

**✅ == vs .equals() in Java:**

| **Feature** | **==** | **.equals()** |
| --- | --- | --- |
| **Type** | Operator | Method |
| **Compares** | Reference (memory address) | Actual content/values |
| **Default behavior** | Checks if both references point to the **same object** | Defined in Object class (also checks reference unless overridden) |
| **Common use** | Primitive comparison, or reference check | Content/value comparison (especially in String, Integer, custom classes) |

**🔹 Example 1: String**

String s1 = new String("hello");

String s2 = new String("hello");

System.out.println(s1 == s2);       // false → different objects

System.out.println(s1.equals(s2));  // true  → same content

**🔹 Example 2: Primitives**

int a = 5;

int b = 5;

System.out.println(a == b);        // true → compares value directly

**🔹 Example 3: Custom Class**

class Person {

    String name;

    Person(String name) {

        this.name = name;

    }

}

Person p1 = new Person("Sumit");

Person p2 = new Person("Sumit");

System.out.println(p1 == p2);        // false → different objects

System.out.println(p1.equals(p2));   // false → same as == unless overridden

🔧 If you override .equals() in Person, you can make it return true for same name values.

**✅ Summary:**

* Use == for:
  + Primitives
  + Checking if two references point to **same object**
* Use .equals() for:
  + Comparing values/contents of **objects**

**✅ Bonus Tip (interview level):**

Always reme mber: "abc" == "abc" may return true due to **String pool**, but in general, for objects use .equals() to avoid bugs.

He asked me to write in chat not IDE.

5. write thread implementations

**✅ 1. Extending Thread class**

class MyThread extends Thread {

    public void run() {

        // Code to be executed in the new thread

        for (int i = 1; i <= 5; i++) {

            System.out.println("Thread running: " + i);

            try {

                Thread.sleep(1000); // Sleep for 1 second

            } catch (InterruptedException e) {

                System.out.println(e);

            }

        }

    }

}

public class ThreadExample {

    public static void main(String[] args) {

        MyThread t1 = new MyThread();

        t1.start(); // Start the thread

        // Main thread code

        for (int i = 1; i <= 5; i++) {

            System.out.println("Main thread: " + i);

            try {

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                System.out.println(e);

            }

        }

    }

}

**✅ 2. Implementing Runnable interface (preferred in real-world apps)**

class MyRunnable implements Runnable {

    public void run() {

        for (int i = 1; i <= 5; i++) {

            System.out.println("Runnable thread: " + i);

            try {

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                System.out.println(e);

            }

        }

    }

}

public class RunnableExample {

    public static void main(String[] args) {

        MyRunnable myRunnable = new MyRunnable();

        Thread t1 = new Thread(myRunnable); // Wrap in a Thread

        t1.start();

        for (int i = 1; i <= 5; i++) {

            System.out.println("Main thread: " + i);

            try {

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                System.out.println(e);

            }

        }

    }

}

**📝 When to Use What?**

| **Approach** | **Use When** |
| --- | --- |
| extends Thread | You don’t need to extend any other class. |
| implements Runnable | You want to extend another class (Java allows only single inheritance), or follow good design practices. |

We **can** use @Override on the run() method, and in fact, we **should** — but it’s **not mandatory**.

**✅ Why use @Override on run()?**

The @Override annotation tells the compiler that:

* This method is **intended to override** a method in a **superclass** (Thread) or from an **interface** (Runnable).
* It helps **catch errors** at compile time — for example, if you misspell run() or get the method signature wrong.

6. @SpringBoot Annotation

**✅ @SpringBootApplication**

@SpringBootApplication

public class MyApplication {

    public static void main(String[] args) {

        SpringApplication.run(MyApplication.class, args);

    }

}

**🔍 What does @SpringBootApplication do?**

It's a **convenience annotation** that combines three annotations:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan

public @interface SpringBootApplication {

}

**🔹 Breakdown of the Components:**

1. **@SpringBootConfiguration**
   * A specialized form of @Configuration
   * Marks this class as the source of bean definitions for the application context
2. **@EnableAutoConfiguration**
   * Tells Spring Boot to automatically configure beans based on the classpath and dependencies
   * Example: If spring-boot-starter-web is present, it configures **Tomcat**, **DispatcherServlet**, etc.
3. **@ComponentScan**
   * Automatically scans for components (@Component, @Service, @Repository, @Controller, etc.)
   * Scans packages **from the class it's applied on and below**

**📝 Best Practices:**

* Place the main class with @SpringBootApplication at the **root package**, so that all sub-packages get scanned automatically.

**🔧 Optional Customizations:**

You can customize like this if needed:

@SpringBootApplication(scanBasePackages = {"com.example.service", "com.example.controller"})

7. write code in JPA to get employees by Numbers

Below is a **simple Spring Boot + JPA example** to fetch an Employee from a database by **employee number** (e.g., empNo).

**✅ Step-by-Step Example:**

**1. Entity Class: Employee.java**

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name = "employees")

public class Employee {

    @Id

    private Long empNo;

    private String name;

    private String department;

    // Getters and setters

    public Long getEmpNo() {

        return empNo;

    }

    public void setEmpNo(Long empNo) {

        this.empNo = empNo;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getDepartment() {

        return department;

    }

    public void setDepartment(String department) {

        this.department = department;

    }

}

**2. Repository Interface: EmployeeRepository.java**

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

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // Find employee by empNo (if empNo is not the @Id, define custom method)

    Employee findByEmpNo(Long empNo);

}

**3. Service Class: EmployeeService.java**

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

import org.springframework.stereotype.Service;

@Service

public class EmployeeService {

    @Autowired

    private EmployeeRepository employeeRepository;

    public Employee getEmployeeByNumber(Long empNo) {

        return employeeRepository.findByEmpNo(empNo);

    }

}

**4. Controller Class: EmployeeController.java**

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

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

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

    @Autowired

    private EmployeeService employeeService;

    @GetMapping("/{empNo}")

    public Employee getEmployee(@PathVariable Long empNo) {

        return employeeService.getEmployeeByNumber(empNo);

    }

}

**✅ Example API Call:**

GET http://localhost:8080/api/employees/101

**📝 Notes:**

* Replace empNo with whatever column/field you're searching by.
* Make sure your table name and column mappings match your database schema.
* If empNo is **not** the @Id, then define a custom finder in the repository (like above).
* Use Optional<Employee> for null safety (e.g., Optional<Employee> findByEmpNo(Long empNo)).

8. I was given one scenario :  
if i use ui and provide any number multiple number i will get the resuly but everytime it is hitting the db how to optimize without caching and indexing.  
I still nnot got the answer though as he was not satisfied if I said caching and indexing.

if you **don’t want caching or indexing**, but still want to **optimize fetching employees by number**, your options become limited — but here’s what you **can** do at the **code and architecture level**:

**✅ 1. Use Persistence Context (First-Level Cache)**

When using **Spring Data JPA**, Hibernate already provides a **first-level cache** within a **single transaction or session**.

**🔹 How it works:**

If you call the same findByEmpNo() multiple times **within the same request/transaction**, Hibernate won't hit the DB again — it will return from the persistence context.

**✅ Example:**

@Transactional

public void someMethod() {

    Employee emp1 = employeeRepository.findByEmpNo(101L); // DB hit

    Employee emp2 = employeeRepository.findByEmpNo(101L); // NO DB hit — same Hibernate session

}

⚠️ Outside the transaction (like separate HTTP calls), this won’t help.

**✅ 2. Reduce Repeated Calls in UI or Service Layer**

Sometimes performance issues are not with the DB, but because of:

* Unnecessary repeated calls from the **frontend**
* Loops in backend calling the same method multiple times

**✅ 3. Batch Fetching (if fetching many employees)**

If your use case involves fetching multiple employees, use:

List<Employee> findAllByEmpNoIn(List<Long> empNos);

Instead of:

java

CopyEdit

for (Long id : empNos) {

    employeeRepository.findByEmpNo(id); // Multiple DB calls — slow

}

9. Arraylist vs Linkedlist

*"ArrayList is backed by a dynamic array, so it provides faster random access using index (get() is O(1)), but inserting or deleting elements in the middle or beginning is costly as it requires shifting elements."*

*"LinkedList, on the other hand, is backed by a doubly linked list, so it's better for frequent insertions or deletions at the beginning or middle, but slower for random access as you need to traverse nodes (get() is O(n))."*

*"So if your use case involves more reads and index-based access, ArrayList is better. If it involves lots of insertions/deletions from the middle or start, LinkedList performs better."*

10. @Entity vs @Table

**✅ @Entity vs @Table in JPA**

| **Annotation** | **Purpose** | **Required?** |
| --- | --- | --- |
| @Entity | Marks a class as a **JPA entity** (i.e., mapped to a DB table) | ✅ Yes |
| @Table | Specifies the **actual table name and settings** | ❌ Optional (defaults to class name) |

**🔹 @Entity**

import jakarta.persistence.Entity;

@Entity

public class Employee {

    // mapped to table named 'employee' by default

}

* Makes the class a **managed JPA entity**
* Must have a **no-args constructor**
* Must have a **primary key** (annotated with @Id)
* By default, the table name is the **class name** (case-sensitive in some DBs)

**🔹 @Table**

import jakarta.persistence.Entity;

import jakarta.persistence.Table;

@Entity

@Table(name = "employees")

public class Employee {

    // explicitly maps to "employees" table in DB

}

* Used when:
  + You want to specify a **custom table name**
  + The class name and table name don’t match
  + You want to define **indexes, constraints** using annotations

**✅ Sample Interview Answer:**

*"@Entity is used to mark a Java class as a JPA entity, meaning it's mapped to a table in the database. Without it, the class won't be managed by JPA."*

*"@Table is optional and used to specify the actual table name, schema, or constraints. If you don't specify it, the class name is used as the table name by default. I usually use @Table when the DB table name differs from the class name or when I want to define indexes."*

***11. write sql to fech second Highest Employee salary***

**✅ SQL Query to Fetch Second Highest Employee Salary**

**🔹 Approach 1: Using LIMIT with OFFSET (MySQL / PostgreSQL)**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

* LIMIT 1 OFFSET 1 skips the highest and picks the second highest.

**✅ What does LIMIT 1 OFFSET 1 do?**

This clause is used in SQL (especially **MySQL** and **PostgreSQL**) to:

* **Skip the first row** (OFFSET 1)
* **Fetch only the next one row** (LIMIT 1)

**🔹 Query Example: Get Second Highest Salary**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

**✅ How it works:**

1. ORDER BY salary DESC → sorts salaries from highest to lowest
2. OFFSET 1 → skips the highest salary (first row)
3. LIMIT 1 → fetches just **one row**, which is the **second highest**

**📌 Example Table:**

| **emp\_id** | **salary** |
| --- | --- |
| 1 | 70000 |
| 2 | 60000 |
| 3 | 80000 |
| 4 | 60000 |
| 5 | 90000 |

After sorting:

| **salary** |
| --- |
| 90000 |
| 80000 |
| 70000 |
| 60000 |

So:

* OFFSET 1 skips 90000
* LIMIT 1 gives you 80000 → second highest

**⚠️ Caution:**

If **multiple employees** have the **same highest salary**, OFFSET 1 may still return the same value (not unique).

✅ So it’s better to use DISTINCT or a **subquery method** if needed.

**✅ Safer Version:**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

Or use:

SELECT MAX(salary)

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

**🔹 Approach 2: Using Subquery (ANSI SQL – works in most DBs)**

SELECT MAX(salary) AS second\_highest\_salary

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

* This finds the **maximum salary less than the highest salary**.

**✅ Explanation:**

SELECT MAX(salary) AS second\_highest\_salary

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

**🔍 What it does:**

1. **SELECT MAX(salary) FROM employees**
   * Finds the **highest salary** in the table.
2. **WHERE salary < (...)**
   * Filters out the highest salary.
3. **SELECT MAX(salary)**
   * From the remaining salaries (i.e., less than the max), it picks the **next highest**, which is the **second highest salary**.

**📌 Example Table:**

| **emp\_id** | **salary** |
| --- | --- |
| 1 | 50000 |
| 2 | 60000 |
| 3 | 70000 |
| 4 | 80000 |
| 5 | 90000 |

* MAX(salary) = 90000
* salary < 90000 → [80000, 70000, 60000, 50000]
* MAX(...) on this = **80000** → ✅ second highest salary

**✅ Why this query is good for interviews:**

* **Works in all RDBMS** (MySQL, PostgreSQL, Oracle, SQL Server, etc.)
* **Easy to understand**
* Handles **duplicate salaries** correctly (e.g., two employees with highest salary)

**🧠 Bonus Interview Tip:**

If they ask for the **employee name with second highest salary**, you can use:

SELECT \*

FROM employees

WHERE salary = (

    SELECT MAX(salary)

    FROM employees

    WHERE salary < (

        SELECT MAX(salary) FROM employees

    )

);

**You said:**

it will just give the salary but how to know whos

**ChatGPT said:**

Exactly! You’ve spotted the key limitation of:

SELECT MAX(salary)

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

This gives only the **salary**, not **who** earns it.

**✅ To get the employee(s) with the second highest salary:**

You need to **wrap that query** and filter the employees whose salary matches it.

**🔹 Query: Get Employee(s) with Second Highest Salary**

SELECT \*

FROM employees

WHERE salary = (

    SELECT MAX(salary)

    FROM employees

    WHERE salary < (SELECT MAX(salary) FROM employees)

);

**✅ What it does:**

* Inner subquery gets **second highest salary**
* Outer query fetches **all employees** with that salary

**📌 Example Table:**

| **emp\_id** | **name** | **salary** |
| --- | --- | --- |
| 1 | John | 70000 |
| 2 | Alice | 80000 |
| 3 | Bob | 90000 |
| 4 | Charlie | 80000 |

→ MAX(salary) = 90000  
→ salary < 90000 → [80000, 70000] → MAX() = 80000  
→ Final result: **Alice and Charlie** (both earning 80000)

**🧠 Bonus: Want Only One (if ties exist)?**

If you only want **one employee** with the second highest salary, you can use:

SELECT \*

FROM employees

WHERE salary = (

    SELECT DISTINCT salary

    FROM employees

    ORDER BY salary DESC

    LIMIT 1 OFFSET 1

)

LIMIT 1;

Add LIMIT 1 at the end to break the tie arbitrarily (not recommended unless you’re sure).

Nth highest salary

**✅ SQL: Get Nth Highest Salary Using LIMIT and OFFSET**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET N - 1;

**🧠 Just replace N with the number you want.**

For example:

* **2nd highest salary**:

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

* **3rd highest salary**:

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 2;

**⚠️ Important Notes:**

1. Use \*\*DISTINCT\*\* to avoid duplicate salaries affecting the result.
2. This gives **only the salary**, not the employee details.
3. **If fewer than N distinct salaries exist**, it will return **empty result**.

**✅ To Get Employee(s) with Nth Highest Salary:**

SELECT \*

FROM employees

WHERE salary = (

    SELECT DISTINCT salary

    FROM employees

    ORDER BY salary DESC

    LIMIT 1 OFFSET N - 1

);

**Example: Get Employee(s) with 3rd Highest Salary**

SELECT \*

FROM employees

WHERE salary = (

    SELECT DISTINCT salary

    FROM employees

    ORDER BY salary DESC

    LIMIT 1 OFFSET 2

);

12. He asked me to write Lambda expression over the chat

13. what is Heap memory ?

14. What is Stream API ?

**✅ What is Stream API in Java?**

The **Stream API** in Java 8 is used to **process collections of data** (like List, Set, etc.) in a **declarative**, **functional-style** way.

15. where do you save your code – bitbucket

16. asked about any cloud experience

17. how would you write REST API

this is a **core question** for Java + Spring Boot developers.

Interviewers asking **"How would you write a REST API?"** want to assess:

1. Your understanding of REST principles
2. How you structure Spring Boot code (Controller, Service, Repository)
3. Whether you follow best practices (DTOs, validations, status codes, etc.)

**✅ Interview-Ready Answer (High-Level)**

*"To write a REST API in Spring Boot, I follow a layered approach:*

1. *I define a @RestController to expose HTTP endpoints.*
2. *I create a @Service class to hold business logic.*
3. *I use a JPA Repository for DB access.*
4. *I define DTOs for request/response, and use validations like @Valid and @NotNull.*  
   *I also handle errors using @ControllerAdvice and return proper HTTP status codes."*

**✅ Example: REST API to Manage Employees**

**1. Entity Class**

@Entity

public class Employee {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    private String department;

    // getters and setters

}

**2. Repository Interface**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

**3. Service Layer**

@Service

public class EmployeeService {

    @Autowired

    private EmployeeRepository repository;

    public List<Employee> getAllEmployees() {

        return repository.findAll();

    }

    public Employee getEmployeeById(Long id) {

        return repository.findById(id)

                .orElseThrow(() -> new RuntimeException("Employee not found"));

    }

    public Employee createEmployee(Employee emp) {

        return repository.save(emp);

    }

    public void deleteEmployee(Long id) {

        repository.deleteById(id);

    }

}

**4. Controller (REST API)**

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

    @Autowired

    private EmployeeService service;

    @GetMapping

    public List<Employee> getAllEmployees() {

        return service.getAllEmployees();

    }

    @GetMapping("/{id}")

    public Employee getEmployeeById(@PathVariable Long id) {

        return service.getEmployeeById(id);

    }

    @PostMapping

    public Employee createEmployee(@RequestBody Employee emp) {

        return service.createEmployee(emp);

    }

    @DeleteMapping("/{id}")

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

        service.deleteEmployee(id);

        return ResponseEntity.ok("Deleted successfully");

    }

}

**✅ Bonus: REST Best Practices**

| **Practice** | **What to Do** |
| --- | --- |
| Proper status codes | Use 200, 201, 404, 400, 500 appropriately |
| Input validation | Use @Valid, @NotNull, etc. |
| DTOs | Avoid exposing entity directly |
| Exception handling | Use @ControllerAdvice |
| Swagger/OpenAPI | For documenting the API |
| Pagination | Use Pageable for large data sets |

Let's enhance the REST API example by adding:

* ✅ DTOs (for cleaner API contracts)
* ✅ Validation annotations (e.g., @NotNull, @Size)
* ✅ Centralized exception handling with @ControllerAdvice
* ✅ PUT (update) and PATCH (partial update)

**📦 Use Case: Employee Management API**

Let’s say each employee has:

* id (auto-generated)
* name (required)
* department (optional)

**✅ 1. Employee Entity**

@Entity

public class Employee {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    private String department;

    // Getters and Setters

}

**✅ 2. DTOs for Request and Response**

**🔹 EmployeeRequestDTO.java**

public class EmployeeRequestDTO {

    @NotBlank(message = "Name is required")

    private String name;

    private String department;

    // Getters and Setters

}

**🔹 EmployeeResponseDTO.java**

public class EmployeeResponseDTO {

    private Long id;

    private String name;

    private String department;

    // Constructor

    public EmployeeResponseDTO(Employee emp) {

        this.id = emp.getId();

        this.name = emp.getName();

        this.department = emp.getDepartment();

    }

    // Getters

}

**✅ 3. Service Layer with Mapping**

@Service

public class EmployeeService {

    @Autowired

    private EmployeeRepository repository;

    public List<EmployeeResponseDTO> getAllEmployees() {

        return repository.findAll()

                .stream()

                .map(EmployeeResponseDTO::new)

                .toList();

    }

    public EmployeeResponseDTO getEmployeeById(Long id) {

        Employee emp = repository.findById(id)

                .orElseThrow(() -> new ResourceNotFoundException("Employee not found with ID: " + id));

        return new EmployeeResponseDTO(emp);

    }

    public EmployeeResponseDTO createEmployee(EmployeeRequestDTO dto) {

        Employee emp = new Employee();

        emp.setName(dto.getName());

        emp.setDepartment(dto.getDepartment());

        return new EmployeeResponseDTO(repository.save(emp));

    }

    public EmployeeResponseDTO updateEmployee(Long id, EmployeeRequestDTO dto) {

        Employee emp = repository.findById(id)

                .orElseThrow(() -> new ResourceNotFoundException("Employee not found with ID: " + id));

        emp.setName(dto.getName());

        emp.setDepartment(dto.getDepartment());

        return new EmployeeResponseDTO(repository.save(emp));

    }

    public void deleteEmployee(Long id) {

        if (!repository.existsById(id)) {

            throw new ResourceNotFoundException("Employee not found with ID: " + id);

        }

        repository.deleteById(id);

    }

}

**✅ 4. REST Controller**

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

    @Autowired

    private EmployeeService service;

    @GetMapping

    public List<EmployeeResponseDTO> getAllEmployees() {

        return service.getAllEmployees();

    }

    @GetMapping("/{id}")

    public EmployeeResponseDTO getEmployeeById(@PathVariable Long id) {

        return service.getEmployeeById(id);

    }

    @PostMapping

    public ResponseEntity<EmployeeResponseDTO> createEmployee(

            @Valid @RequestBody EmployeeRequestDTO dto) {

        return new ResponseEntity<>(service.createEmployee(dto), HttpStatus.CREATED);

    }

    @PutMapping("/{id}")

    public EmployeeResponseDTO updateEmployee(

            @PathVariable Long id,

            @Valid @RequestBody EmployeeRequestDTO dto) {

        return service.updateEmployee(id, dto);

    }

    @DeleteMapping("/{id}")

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

        service.deleteEmployee(id);

        return ResponseEntity.ok("Deleted successfully");

    }

}

**✅ 5. Global Exception Handler**

**🔹 ResourceNotFoundException.java**

public class ResourceNotFoundException extends RuntimeException {

    public ResourceNotFoundException(String message) {

        super(message);

    }

}

**🔹 GlobalExceptionHandler.java**

@RestControllerAdvice

public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)

    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {

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

    }

    @ExceptionHandler(MethodArgumentNotValidException.class)

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

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

        ex.getBindingResult().getFieldErrors().forEach(err ->

                errors.put(err.getField(), err.getDefaultMessage()));

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

    }

}

**🧪 Sample Input for POST/PUT**

{

  "name": "Sumit",

  "department": "Engineering"

}

**🧠 Summary**

* ✅ Clean layering (Controller → Service → Repo)
* ✅ DTOs decouple entity from API
* ✅ Validation via @Valid and @NotBlank
* ✅ Custom exception + global handler
* ✅ Follows REST best practices

18. some questions on Exceptions mainly focused on Finally

19. Which java version you are using – told java 21.

Key takeaways :

* > Grip the basics  
  JPA, controller code and thread code and exception tat too on chat  
  practice on Notepad
* Hold the grip on concepts and syntax.
* Mention project name on resume