Java Basics for Spring Boot (Hinglish Me)

1. Java Kya Hai?

Java ek **object-oriented programming (OOP)** language hai.
Iska use mainly backend development, web applications, aur mobile apps (Android) banane ke liye hota hai.
Java platform-independent hai, matlab tum ek baar code likho aur usse kisi bhi platform (Windows, Linux, Mac) pe run kar sakte ho.

2. Java Setup

JDK (Java Development Kit): Java code likhne aur run karne ke liye JDK install karna zaroori hai.
IDE (Integrated Development Environment): IntelliJ IDEA, Eclipse, ya VS Code use kar sakte ho. Spring Boot ke liye IntelliJ IDEA best hai.

3. Java Syntax Basics

Class: Java me har cheez ek class me hoti hai. Class ek blueprint hota hai objects banane ka.

HelloWorld.java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Explanation:

- `public`: Access modifier, matlab yeh class sabko accessible hai.
- `class`: Keyword jo class define karta hai.
- `main`: Yehi se program execution shuru hota hai.
- `System.out.println`: Console pe print karne ke liye.

Variables: Data store karne ke liye.

Variables.java

```java
int age = 25; // Integer
String name = "Rahul"; // String
double price = 99.99; // Decimal
boolean isJavaFun = true; // Boolean
```

Data Types:

- **Primitive:** `int`, `double`, `boolean`, `char`, etc.
- **Non-Primitive:** `String`, `Array`, `Class`, etc.

4. Object-Oriented Programming (OOP)

Java OOP principles follow karta hai. Ye principles hain:

Encapsulation: Data aur methods ko ek unit me bandhna.

Person.java

```
class Person {
    private String name; // Private variable
    public String getName() { // Getter
        return name;
    }
    public void setName(String name) { // Setter
        this.name = name;
    }
}
```

Inheritance: Ek class dusri class ke properties aur methods inherit kar sakti hai.

Animal.java

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}
class Dog extends Animal { // Dog inherits Animal
    void bark() {
        System.out.println("Barking...");
    }
}
```

Polymorphism: Ek hi method ka alag-alag tarike se use karna.

Method Overloading: Same method name but different parameters.

MethodOverloading.java

```
void add(int a, int b) { System.out.println(a + b); }
void add(double a, double b) { System.out.println(a + b); }
```

5. Method Overriding

Meaning: Child class me parent class ke method ko **redefine** karna.
Use: Jab child class me parent class ke method ka **new implementation** chahiye.

Animal.java

```java
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}
```

Explanation:

- `Animal` class ka `sound()` method "Animal sound" print karta hai.
- `Dog` class (child class) ne `sound()` method ko **override** kiya aur "Bark" print karaya.

@Override Annotation:

- **What:** Ye Java me ek **hint** hai compiler ko batane ke liye ki method override ho raha hai.
- **Optional:** Use karna zaruri nahi hai, lekin accha practice hai.
- **Benefit:** Agar method ka naam ya signature galat hai, toh error dega.

Key Points:

- Method Overriding me **method name** aur **signature** same hona chahiye.
- `@Override` use karna safe hai, lekin mandatory nahi hai.
- Child class ka method parent class ke method ko **replace** karta hai.

==================================

Abstraction and Interfaces in Java (Hinglish Me)

Abstraction

Meaning: Complex details ko **hide** karna aur sirf simple interface provide karna.
Example: Car chalane ke liye aapko ye nahi pata ki engine kaise kaam karta hai, bas accelerator dabana hai.

Abstract Class

1. **Keyword:** ‘abstract‘
2. **Use:** Jab aap kuch methods ka implementation child class pe chhodna chahte hain.
3. **Cannot create object:** Abstract class ka object nahi banaya ja sakta.
4. **Can have:**
- Abstract methods (without body).
- Concrete methods (with body).

Vehicle.java

```java
abstract class Vehicle {
    abstract void run(); // Abstract method (no body)
    void stop() {
        System.out.println("Vehicle stopped"); // Concrete method
    }
}

class Car extends Vehicle {
    void run() {
        System.out.println("Car is running");
    }
}
```

Explanation:

- `Vehicle` class ek abstract class hai.
- `run()` method ka implementation `Car` class me diya gaya hai.
- `stop()` method ka implementation already hai, ise directly use kar sakte hain.

Interface

1. **Keyword:** ‘interface‘
2. **Use:** Jab aap pure abstraction chahate hain (sirf method signature, no implementation).
3. **Cannot create object:** Interface ka object nahi banaya ja sakta.
4. **All methods are abstract:** Java 8 se default methods allow hain (method with body).
5. **Keyword for implementation:** ‘implements‘

Vehicle.java

```java
interface Vehicle {
    void run(); // Abstract method (no body)
    default void stop() {
        System.out.println("Vehicle stopped"); // Default method (Java 8+)
    }
}

class Car implements Vehicle {
    public void run() {
        System.out.println("Car is running");
    }
}
```

Explanation:

- `Vehicle` interface me `run()` method ka koi implementation nahi hai.
- `stop()` method ka implementation already hai (Java 8+ me default methods allow hain).
- `Car` class ne `run()` method ko implement kiya hai.

Default Keyword in Interface

Meaning: Java 8 se interfaces me **default methods** allow hain.
Use: Agar aap interface me ek method ka implementation provide karna chahte hain, lekin usko override karne ki zarurat nahi hai.

Vehicle.java

```java
interface Vehicle {
    void run(); // Abstract method (no body)
    default void stop() {
        System.out.println("Vehicle stopped"); // Default method (Java 8+)
    }
}
```

Explanation:

- `stop()` method ka implementation already hai, ise override karne ki zarurat nahi hai.

Abstract Class vs Interface

Abstract Class	Interface
Can have **abstract** + **concrete** methods.	Only **abstract** methods (Java 8 se default methods allow hain).
Single inheritance (1 class extend).	**Multiple inheritance** (1 class multiple interfaces implement kar sakti hai).
‘extends‘ keyword use hota hai.	‘implements‘ keyword use hota hai.

Key Points

- **Abstract Class:** Partial abstraction, ‘extends‘ keyword, object nahi ban sakta.
- **Interface:** Full abstraction, ‘implements‘ keyword, object nahi ban sakta.

Point To Note

- **Static Method:** Object banaye bina call kar sakte hain.

- **Abstract Method:** Must be overridden in child class.

Example in Short

Vehicle.java

```java
abstract class Vehicle {
    abstract void run(); // No body
    void stop() {
        System.out.println("Vehicle stopped");
    }
}
class Car extends Vehicle {
    void run() {
        System.out.println("Car is running");
    }
}
```

Vehicle.java

```java
interface Vehicle {
    void run(); // No body
    default void stop() {
        System.out.println("Vehicle stopped");
    }
}
class Car implements Vehicle {
    public void run() {
        System.out.println("Car is running");
    }
}
```

Abstraction in Web Development (Real-Time Example in Hindi)

Abstraction kya hota hai?

Abstraction ka matlab hota hai **complexity ko hide karna aur sirf zaroori details dikhana**.

- **Car ka example:** Aap car chalate ho to sirf steering, accelerator aur brake ka use karte ho.
- Aapko ye nahi pata hota ki engine andar kaise kaam kar raha hai.
- Car manufacturer ne complexity hide kar di hai aur ek simple interface diya hai.

Real-Time Example: E-Commerce Website

Maan lo **Amazon ka ek product listing page** hai jo user ko products dikhata hai.

- User sirf `/products` API call karta hai aur sirf products ka data dekhta hai.
- **Lekin usko ye nahi pata chalega ki:**
  - Data kaha se aa raha hai?
  - Database kaunsa use ho raha hai?
  - Backend ka structure kaisa hai?

Example

Yahi abstraction hai! User sirf API call karta hai aur response leta hai bina backend ki complexity samjhe.

Spring Boot me Abstraction ka Use

Step 1: Abstract Class (Service Layer)

ProductService.java (Abstract Class)

```java
abstract class ProductService {
    abstract List<String> getAllProducts();
    void logRequest() {
        System.out.println("Request logged for product API");
    }
}
```

Step 2: Implementation Class (Product Service)

```java
class ProductServiceImpl extends ProductService {
    @Override
    List<String> getAllProducts() {
        logRequest();
        return Arrays.asList("Laptop", "Mobile", "Headphones");
    }
}
```

Step 3: API Controller (User Ko Response Dikhana)

```java
@RestController
@RequestMapping("/products")
class ProductController {
    private final ProductService productService = new ProductServiceImpl();

    @GetMapping
    public List<String> getProducts() {
        return productService.getAllProducts();
    }
}
```

Final Output

Agar user **browser ya Postman** me API call karega:

```
GET http://localhost:8080/products

Response:
["Laptop", "Mobile", "Headphones"]
```

Interface ka Real Use in Web Development

Step 4: Interface for Database (Product Repository)

```java
interface ProductRepository extends JpaRepository<Product, Integer> {
    // Spring Boot khud implementation handle karega (Abstraction)
}
```

Conclusion

lightBlue **Layer**	**Role**	**Abstraction ka Use**
Service Layer	Business logic handle karta hai	Abstract class se reusability badhti hai
lightred **Controller Layer**	User ke request ko process karta hai	User ko backend ki complexity nahi dikhai jati

Summary (Key Learnings)

- **Abstract Class** - Common logic reuse karne ke liye.
- **Interface** - Backend complexity hide karne ke liye.
- **Spring Boot** me abstraction ka use **Service Layer & Repository Layer** me hota hai.
- **User ko sirf API ka response milta hai bina backend ki complexity samjhe.**

==================================

Java Collections Framework and More (Hinglish Me)

Collections Framework

Meaning: Java me data structures (jaise list, set, map) ko handle karne ke liye ek framework hai.
Use: Data ko store, retrieve, aur manipulate karne ke liye.

1. List

Meaning: Ordered collection of elements.
Properties:

- Elements can be **duplicate**.
- Elements are stored in **sequence** (order).

Common Implementations:

- `ArrayList`: Dynamic array (fast access, slow insertion/deletion).
- `LinkedList`: Doubly linked list (fast insertion/deletion, slow access).

Main.java

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(); // List banaya
        names.add("Rahul"); // Element add kiya
        names.add("Amit");  // Element add kiya
        System.out.println(names); // Output: [Rahul, Amit]
    }
}
```

Explanation:

- `names.add()` se elements add hote hain.
- `System.out.println(names)` se pure list print hota hai.

2. Set

Meaning: Collection of **unique** elements.
Properties:

- Elements cannot be **duplicate**.
- No specific order (unordered collection).

Common Implementations:

- `HashSet`: Uses hashing for fast access.
- `TreeSet`: Stores elements in sorted order.

Main.java

```java
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<Integer> numbers = new HashSet<>(); // Set banaya
        numbers.add(1); // Element add kiya
        numbers.add(2); // Element add kiya
        numbers.add(1); // Duplicate element (add nahi hoga)
        System.out.println(numbers); // Output: [1, 2]
    }
}
```

Explanation:

– `numbers.add(1)` do baar call kiya, lekin `Set` me duplicate allow nahi hai.

– `System.out.println(numbers)` se unique elements print hote hain.

3. Map

Meaning: Collection of **key-value pairs**.
Properties:

– Keys are **unique**.

– Values can be duplicate.

Common Implementations:

– `HashMap`: Uses hashing for fast access.

– `TreeMap`: Stores keys in sorted order.

Main.java

```java
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>(); // Map banaya
        map.put("Rahul", 25); // Key-Value pair add kiya
        map.put("Amit", 30);  // Key-Value pair add kiya
        System.out.println(map); // Output: {Rahul=25, Amit=30}
    }
}
```

Explanation:

– `map.put("Rahul", 25)` se key-value pair add hota hai.

– `System.out.println(map)` se pure map print hota hai.

Key Differences

Feature **Map**	**List**	**Set**
Order Unordered (no sequence)	Ordered (sequence)	Unordered (no sequence)
Duplicates Keys: Not allowed, Values: Allowed	Allowed	Not allowed
Example `HashMap`, `TreeMap`	`ArrayList`, `LinkedList`	`HashSet`, `TreeSet`

Common Methods

– **List:**
- `add(element)`: Element add karna.
- `get(index)`: Element access karna.
- `remove(index)`: Element remove karna.

– **Set:**
- `add(element)`: Element add karna.
- `contains(element)`: Check karna ki element hai ya nahi.
- `remove(element)`: Element remove karna.

– **Map:**
- `put(key, value)`: Key-value pair add karna.
- `get(key)`: Value access karna.
- `remove(key)`: Key-value pair remove karna.

Example in Short

Main.java

```java
List<String> names = new ArrayList<>();
names.add("Rahul");
names.add("Amit");
System.out.println(names); // Output: [Rahul, Amit]
```

Main.java

```java
Set<Integer> numbers = new HashSet<>();
numbers.add(1);
numbers.add(2);
numbers.add(1); // Duplicate (add nahi hoga)
System.out.println(numbers); // Output: [1, 2]
```

Main.java

```java
Map<String, Integer> map = new HashMap<>();
map.put("Rahul", 25);
map.put("Amit", 30);
System.out.println(map); // Output: {Rahul=25, Amit=30}
```

Key Points

– **List:** Ordered, duplicates allowed.

– **Set:** Unordered, no duplicates.
– **Map:** Key-value pairs, keys unique.

6. Exception Handling

Errors ko handle karne ke liye `try-catch` block use karte hain.

Main.java

```java
try {
    int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("This will always execute");
}
```

Explanation:

– `try` block me error-prone code likha jata hai.
– `catch` block me error handle kiya jata hai.
– `finally` block hamesha execute hota hai, chahe exception ho ya nahi.

7. Keywords in Java

Point To Note

static:
`static` keyword ka matlab hai ki wo method ya variable class ka part hai, object ka nahi.

–

Example.java

```java
class Example {
    static int count = 0; // Static variable
    static void print() { // Static method
        System.out.println("Hello");
    }
}
```

Explanation:

∗ `count` aur `print` ko object banaye bina use kar sakte hain: `Example.count` ya `Example.print()`.

Point To Note

final:
`final` keyword ka matlab hai ki uski value change nahi ki ja sakti.

∗

Example.java

```java
final int age = 25; // Ab age ki value change nahi kar sakte
final class Animal { // Ab Animal class ko inherit nahi kar sakte
    // Class content
}
```

this:
`this` keyword current object ko refer karta hai.

Person.java

```java
class Person {
    String name;
    Person(String name) {
        this.name = name; // Current object ka name set karo
    }
}
```

Point To Note

8. Constructor

Constructor ek special method hai jo object banate waqt automatically call hota hai.
Iska use object ki initial state set karne ke liye hota hai.
Constructor ka naam class ke naam jaisa hi hota hai, aur yeh kuch return nahi karta.

```java
class Car {
    String name;
    // Constructor
    Car(String name) {
        this.name = name;
    }
}
```

·

Types of Constructors:

- **Default Constructor:** Agar tum koi constructor nahi banate, toh Java apne aap ek default constructor banata hai.
- **Parameterized Constructor:** Jo parameters leta hai.
- **Constructor Overloading:** Ek se zyada constructors hona.

Car.java

```java
class Car {
    String name;
    int speed;
    Car() { // Default constructor
        this.name = "Unknown";
    }
    Car(String name) { // Parameterized constructor
        this.name = name;
    }
    Car(String name, int speed) { // Another parameterized
       constructor
        this.name = name;
        this.speed = speed;
    }
}
```

9. Methods in Java

Methods functions hote hain jo class ke andar define hote hain.

Calculator.java

```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

Method Overloading:
Ek se zyada methods ka naam same hona, lekin parameters alag hona.

Calculator.java

```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
```

10. Access Modifiers

Ye batate hain ki kaun class, method, ya variable ko access kar sakta hai.

Example.java

```java
class Example {
    public int publicVar = 1;
    private int privateVar = 2;
    protected int protectedVar = 3;
    int defaultVar = 4; // Default
}
```

Explanation:

Point To Note

- `public`: Sabko accessible.
- `private`: Sirf class ke andar accessible.
- `protected`: Same package aur child classes ke liye accessible.
- `default`: Sirf same package me accessible.

11. Inheritance

Inheritance ka matlab hai ki ek class dusri class ke properties aur methods ko inherit kar sakti hai.
`extends` keyword ka use hota hai.

Animal.java

```java
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}
class Dog extends Animal { // Dog inherits Animal
    void bark() {
        System.out.println("Barking...");
    }
}
```

12. Polymorphism

Polymorphism ka matlab hai ”ek se zyada forms”.

Method Overloading: Same method name, different parameters.

Example.java

```java
void add(int a, int b) { System.out.println(a + b); }
void add(double a, double b) { System.out.println(a + b); }
```

Method Overriding: Child class me parent class ke method ko redefine karna.

Animal.java

```java
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Bark"); }
}
```

==================================

01em

Spring Boot Project from Scratch (Step-by-Step) in Hinglish (For Beginners)

Your Name February 12, 2025

1 Introduction

This guide provides a step-by-step tutorial for creating a Spring Boot project from scratch. It is designed to be beginner-friendly and includes Hinglish explanations for better understanding.

Hinglish Explanation: Yeh guide Spring Boot project banane ka step-by-step tarika batata hai. Isme folder structure, database setup, CRUD operations, aur IntelliJ IDEA me project setup kaise karna hai, sab kuch cover kiya gaya hai.

2 Step 1: Install Java aur IntelliJ IDEA

2.1 Install Java JDK (Java Development Kit)

Spring Boot project run karne ke liye Java install hona zaroori hai.

Download Java JDK: https://www.oracle.com/java/technologies/javase-jdk17-downloads.htmlOracle Java JDK Download Install hone ke baad check karne ke liye command run karo:

```
java -version
```

3 Step 2: Create New Spring Boot Project

3.1 IntelliJ IDEA me Project Create Karo

Agar IntelliJ IDEA me Spring Initializr ka option nahi hai, toh manually Spring Boot project download karo.

Spring Initializr Website: https://start.spring.io/Spring Initializr Ye details fill karo:

- **Project:** Maven
- **Language:** Java
- **Spring Boot Version:** Latest Stable Version
- **Packaging:** Jar
- **Java Version:** 17

click on ADD Required Dependencies

- **Spring Web** → REST API banane ke liye
- **Spring Boot DevTools** → Auto Restart ke liye
- **Spring Data JPA** → Database ke liye
- **MySQL Driver** → MySQL Database ke liye
- **Lombok** → Code short aur clean banane ke liye

4 Step 3: Project Structure Samjho

```
SpringBootApp/
          src/
                main/
                      java/com/example/springbootapp/
                            controller/      <-- (Express.js ke routes jaise)
                            service/         <-- (Business logic ka layer)
                            repository/      <-- (Database operations, like Mongoose)
                            entity/          <-- (Database Models, like Mongoose Schema)
                            dto/             <-- (Data Transfer Objects)
                            config/          <-- (Configurations, like middleware)
                            SpringBootApp.java  <-- (Main server file, like app.js)
                      resources/
                            application.properties  <-- (Database config)
          pom.xml   <-- (Dependencies list, like package.json)
```

Step-by-Step Guide to Creating a Spring Boot Folder Structure in IntelliJ IDEA

Introduction

This document explains how to create a well-structured Spring Boot project in IntelliJ IDEA with all necessary folders and files.

Step-by-Step Guide

Creating the Folder Structure Manually

If some folders are missing, follow these steps:

1. Go to **src/main/java/com/yourcompany/yourproject** in IntelliJ IDEA.
2. Right-click on **yourproject** → **New** → **Package**.
3. Name the packages:
4. **controller**
5. **service**
6. **repository**
7. **model**

8. Inside each package, create the following files:
9. **controller**: YourController.java
10. **service**: YourService.java
11. **repository**: YourRepository.java
12. **model**: YourEntity.java

4. Purpose of Each Folder

Controller (controller/) - Handles HTTP requests:

```java
@RestController
@RequestMapping("/api")
public class YourController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

Service (service/) - Contains business logic:

```java
@Service
public class YourService {
    public String processData() {
        return "Processed Data";
    }
}
```

Repository (repository/) - Interfaces with the database:

```java
@Repository
public interface YourRepository extends JpaRepository<YourEntity,
    ↪ Long> {
}
```

Model (model/) - Represents the database table:

```java
@Entity
@Table(name = "your_table")
public class YourEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

5. Running the Spring Boot Application

- Open **YourApplication.java** and click the red **Run** button in IntelliJ.

Summary

· Follow the standard package structure.
· Use @Controller, @Service, @Repository, @Entity annotations.
· Run the application from YourApplication.java.
· Test REST API using Postman or browser: http://localhost:8080/api/hello.

5 Step 4: Configure Database (MySQL)

Database Configuration Location: `src/main/resources/application.properties`

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/springboot_db
spring.datasource.username=root
spring.datasource.password=your_password

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

6 Step 5: Run Server in IntelliJ

1. `SpringBootApp.java` **file open karo** 2. **Right Click** → **Run** 3. Console me `"Tomcat started on port(s): 8080"` likha aayega.
Test API using Postman

- **GET Users** → `http://localhost:8080/users`
- **POST User** → `http://localhost:8080/users`

Example JSON Request:

```
{
  "name": "John Doe",
  "email": "john@example.com"
}
```

- **JDK Install** karo.
- **IDE Install** karo (IntelliJ IDEA ya Eclipse).
- **Spring Boot Project** banayo (Spring Initializr se).
- **Controller** banayo aur '@GetMapping' se API banayo.
- **Run** karo aur API test karo.

article xcolor listings tcolorbox hyperref

7 Spring Boot ko MySQL Workbench se Connect Karna

Agar tum **Spring Boot application ko MySQL database (MySQL Workbench) se connect** karna chahte ho, toh `application.properties` file me **database configuration** set karna hoga.

7.1 Configuration File Location:

Ye file **Spring Boot project ke andar hoti hai**: `src/main/resources/application.properties`

Note

Yeh file Express.js ke `config/database.js` **jaisi hoti hai**, jisme **database ka URL, username, password, aur configurations hoti hain**.

8 Step-by-Step Database Connection Setup

8.1 Step 1: MySQL Workbench Me Database Create Karo

Sabse pehle **MySQL Workbench open karo** aur ek **naya database (schema) create karo**:

1. Workbench Open Karo
2. SQL Editor me jao
3. Query likho aur run karo:

```
    CREATE DATABASE springboot_db;
```

4. Refresh karo aur ensure karo ki springboot$_dbcreatehogayahai$.

Point To Note

8.2 Step 2: Spring Boot Ke application.properties Me Database Configure Karo

Ab **Spring Boot ko batana padega ki kis database se connect hona hai**. **Open karo:** `src/main/resources/application.properties` Aur ye configuration likho:

```
# Spring Boot server ka port set karo
server.port=8080

# MySQL Database ka connection URL set karo
spring.datasource.url=jdbc:mysql://localhost:3306/springboot_db

# MySQL ka username aur password set karo
spring.datasource.username=root
spring.datasource.password=your_password

# Hibernate (ORM) ka setup karo
spring.jpa.hibernate.ddl-auto=update

# Console me SQL queries dikhane ke liye
spring.jpa.show-sql=true

# MySQL 8 dialect set karo
tspring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

- `server.port=8080` - Spring Boot ka server kis port pe chalega.
- `spring.datasource.url` - Database connection URL.
- `spring.datasource.username=root` & `spring.datasource.password` - MySQL ka login credentials.
- `spring.jpa.hibernate.ddl-auto=update` - Hibernate ORM ka setup.
- `spring.jpa.show-sql=true` - Console me SQL queries dikhane ke liye.
- `spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect` - MySQL 8 ke liye dialect.

9 Step 3: Server Run Karo aur Check Karo Ki Database Connect Ho Raha Hai

1. IntelliJ IDEA ya VS Code me project open karo.
2. `SpringBootApp.java` file me jao (jo `main` class hai).
3. Run button click karo ya ye command run karo:

```
    mvn spring-boot:run
```

4. Agar sab sahi hai toh console me yeh message aayega:

```
    Tomcat started on port(s): 8080
```

10 Bonus: Check Connection Using Postman

Agar tumne **CRUD API banayi hai** toh check karne ke liye **Postman use kar sakte ho**:

- **GET Users:** `http://localhost:8080/users`
- **POST User:** `http://localhost:8080/users`

```
{
  "name": "John Doe",
  "email": "john@example.com"
}
```

11 Conclusion

- **Step 1:** MySQL Workbench me **database create karo**.
- **Step 2:** `application.properties` me **database URL, username, aur password set karo**.
- **Step 3: Spring Boot server run karo aur ensure karo ki MySQL connect ho raha hai**.
- **Step 4: Postman ya log messages check karo connection verify karne ke liye**.

Ab tumhara Spring Boot application MySQL Workbench se successfully connect ho gaya!

Express.js vs Spring Boot Comparison

Feature	**Express.js**	**Spring Boot**
Language	JavaScript	Java
Framework	Lightweight	Enterprise-level
Routes	‘app.get()‘, ‘app.post()‘	‘@GetMapping‘, ‘@PostMapping‘
Server Start	‘node app.js‘	‘DemoApplication.java‘ run karo
Port	Default: 3000	Default: 8080

Example Code

HelloController.java

```java
package com.example.demo;

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

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

DemoApplication.java

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

==============================

[a4paper,12pt]article

12 Maven Kya Hai? (NPM ki tarah kaam karta hai?)

Maven ek build automation tool hai jo Java projects ke liye use hota hai. Yeh NPM (Node Package Manager) ke jaise kaam karta hai lekin Java ecosystem ke liye hota hai. NPM Node.js ke dependencies aur scripts ko manage karta hai, jabki Maven Java projects me dependencies, project structure, aur build process ko automate karta hai.

12.1 Maven Ke Fayde

- **Dependency Management**: Jaise NPM me `package.json` hota hai, waise hi Maven me `pom.xml` hota hai jisme dependencies define hoti hain.
- **Project Structure Standardization**: Java projects ka structure organize aur standardized rehta hai.
- **Build & Deployment Automation**: Code compile karna, test cases run karna, aur project ka .jar ya .war file generate karna easy hota hai.
- **Plugin Support**: Alag-alag plugins ka use karke testing, packaging, aur deployment automate kiya ja sakta hai.

12.2 Example (Maven vs NPM)

Feature	**NPM (Node.js)**	**Maven (Java)**
Dependency File	`package.json`	`pom.xml`
Install Dependencies	`npm install`	`mvn install`
Run Project	`npm start`	`mvn spring-boot:run`
Build Project	`npm run build`	`mvn package`

13 Maven Install Karna Aur Environment Variable Set Karna

13.1 Step 1: Download Maven

Official Website: https://maven.apache.org/download.cgi
.zip ya .tar.gz file download karo aur extract karo.

13.2 Step 2: Environment Variable Set Karo (Windows ke liye)

Extracted Folder ka Path Copy Karo

Example: `C:\apache-maven-3.8.6\bin`
Environment Variable Me Add Karo:

- Windows Search → ”*EnvironmentVariables*” *searchkaroSystemProperties* → *Advanced* → *EnvironmentVariables*
- System Variables section me Path select karo → *Edit* → *New* → `C:\apache-maven-3.8.6\bin`*pastekaro* → *OK*.

13.3 Check Installation

CMD open karo aur run karo:

```
mvn -version
```

Agar Apache Maven 3.x.x show ho raha hai, toh setup sahi hai.

13.4 Linux/Mac Ke Liye

Terminal Open Karo:

```
sudo apt install maven  # (Ubuntu)
brew install maven  # (Mac)
```

Check Version:

```
mvn -version
```

14 Spring Boot Java 17 Pe Kyun Chal Raha Hai, Java 23 Pe Nahi?

Spring Boot projects ke liye Java ka supported version important hota hai. Spring Boot ka latest stable version Java 17 LTS (Long-Term Support) ko officially support karta hai, lekin Java 23 ek non-LTS version hai jo backward compatibility issues create kar sakta hai.

14.1 Java 17 vs Java 23 in Spring Boot

Feature	**Java 17 (LTS)**	**Java 23 (Non-LTS)**
Stability	Stable & widely used	Frequent updates, no long-term support
Spring Boot Compatibility	Officially supported	Not officially supported
Backward Compatibility	Ensured	Breaking changes expected

14.2 Kaise Check Kare Ki Kaunsi Java Version Use Ho Rahi Hai?

Command Line Check Karo:

```
java -version
```

Agar `java 17.x.x` show ho raha hai, toh aap Java 17 use kar rahe ho.

14.3 Spring Boot Configuration Me Java Version Set Karo

Maven Project (`pom.xml`) me ensure karo ki yeh lines included ho:

```
<properties>
    <java.version>17</java.version>
</properties>
```

14.4 Agar aap Java 23 use karna chahte ho, toh aapko manually dependencies aur compatibility issues fix karne padenge.

Spring Boot Me Debugging (Django ke breakpoint() jaise)

Breakpoint Kya Hai?

Breakpoint ek aisa point hota hai jahan aapka code execution ruk jata hai aur aap uss point pe variables, expressions, aur program flow ko inspect kar sakte ho. Breakpoint set karne ke baad, aap code ko step-by-step execute kar sakte ho aur dekhte ho ki kya ho raha hai.

Breakpoint Set Karne Ka Tarika

IntelliJ IDEA/Eclipse Me Breakpoint Set Karo

Code editor me line number ke left side pe click karo. Ek red dot show hoga jo breakpoint ko represent karta hai.
Example:

```java
public void greet() {
    String name = "John";  // Yahan breakpoint set karo
    System.out.println("Hello, " + name);
}
```

Debug Mode Me Run Karo

IntelliJ: Run → Debug 'Application'
Eclipse: Right Click on Application → Debug As → Java Application

Point To Note

Step Over, Step Into, Aur Step Out Kya Hai?

Step Over (F8)

Current line ko execute karo aur next line pe move karo. Agar current line me function call hai, toh uss function ke andar nahi jayega.
Example:

```java
public void greet() {
    String name = "John";  // Step Over karne pe next line pe move
    ↪ hoga
    System.out.println("Hello, " + name);
}
```

Step Into (F7)

Current line ko execute karo aur agar uss line me function call hai, toh uss function ke andar jayega.
Example:

```java
public void greet() {
    String name = "John";
    printName(name);  // Step Into karne pe printName function ke
    ↪ andar jayega
}

public void printName(String name) {
    System.out.println("Name: " + name);
}
```

Step Out (Shift + F8)

Agar aap kisi function ke andar ho aur uss function se bahar aana chahte ho, toh Step Out ka use karo. Yeh aapko function ke end tak execute karke wapas caller function pe le jayega.

Debug Mode Me Server Start Karo

Command line se debug mode enable karne ke liye:

```
mvn spring-boot:run -Ddebug
```

.

Ya manually application.properties file me add karo:

```
debug=true
```

Isse detailed logs console me show honge.

Remote Debugging Enable Karo

Agar aap server remotely debug karna chahte ho:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005 -
    ↪ jar yourapp.jar
```

Phir IntelliJ ya Eclipse me Remote Debug Configuration set karke Port 5005 pe connect karo.
================================

Spring Boot Folder Structure

Spring Boot Folder Structure

Express.js me hum generally ye structure follow karte hain:

```
project/
          routes/       (API routes)
          controllers/  (Business logic)
          models/       (Database models)
          middleware/   (Middleware functions)
          app.js        (Main server file)
```

Spring Boot me iska equivalent structure kuch aisa hota hai:

```
src/
          main/
                java/
                      com/
                          example/
                              demo/
                                  controller/   (API endpoints)
                                  service/      (Business logic)
                                  repository/   (Database
    operations)
                                  model/        (Database models)
                                  DemoApplication.java  (Main
    server file)
                resources/
                    static/   (Static files like CSS, JS)
                    templates/ (HTML templates)
                    application.properties  (Configuration file)
          test/
              java/
                  com/
                      example/
                          demo/
                              DemoApplicationTests.java  (Test file
    )
```

1. ‘controller/‘ Folder

Express.js Comparison:

- Express.js me ‘routes/‘ folder me API routes define kiye jate hain.
- Example:

Express.js Example

```
app.get('/hello', (req, res) => {
    res.send('Hello, Express!');
});
```

Spring Boot me:

- ‘controller/‘ folder me API endpoints define kiye jate hain.
- Ye folder **REST APIs** handle karta hai.
- Example:

- **Point To Note**

  'controller/' folder me API endpoints define kiye jate hain.
- Ye folder **REST APIs** handle karta hai.

HelloController.java

```java
package com.example.demo.controller;

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

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

2. 'service/' Folder

Express.js Comparison:

- Express.js me business logic 'controllers/' ya alag modules me likha jata hai.
- Example:

Express.js Example

```javascript
const getData = () => {
    return "Some data";
};
```

Spring Boot me:

- **Point To Note**

  'service/' folder me **business logic** likha jata hai.
- **Point To Note**

  Ye folder **reusable code** aur complex logic handle karta hai.
- Example:

HelloService.java

```java
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class HelloService {

    public String getMessage() {
        return "Hello from Service!";
    }
}
```

3. ‘repository/‘ Folder

Express.js Comparison:

- Express.js me database operations ‘models/‘ folder me likhe jate hain.
- Example:

Express.js Example

```javascript
const User = require('./models/User');
User.find().then(users => console.log(users));
```

141em
article tcolorbox
article tcolorbox

Point To Note

15 Step 1: Repository Folder Ka Role

Spring Boot me `repository` folder database ke saath interaction handle karta hai. Ye Express.js ke `Mongoose Model` jaisa kaam karta hai, jo CRUD (Create, Read, Update, Delete) operations perform karta hai.

Point To Note

Hinglish Explanation: Repository folder ka kaam database se interact karna hota hai. Ye Express.js ke `models/User.js` (Mongoose Model) jaisa hota hai.

black

Folder	**Express.js Equivalent**	**Role**
`repository/`	`models/User.js` (Mongoose Model)	Database se CRUD operations handle karta hai.
`entity/`	`models/User.js` (Schema)	Database table ka structure define karta hai.

Table 1: Spring Boot Repository vs Express.js Mongoose Model

Spring Boot me:

- ‘repository/‘ folder me **database operations** likhe jate hain.
- Ye folder **database se data fetch ya save** karne ka kaam karta hai.
- Example:

UserRepository.java

```java
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User,
    ↪ Long> {
    // Custom queries yahan likh sakte hain
}
```

16 Repository Ki Zaroorat Kyun Hai?

Spring Boot me **Repository** ek important component hai jo database aur application ke beech ka bridge ka kaam karta hai. Agar repository na ho, toh aap directly database queries nahi likh sakte, aur `.findById()`, `.save()`, `.deleteById()` jaise methods use nahi kar sakte.

Example

Without Repository, You Cannot Use These Methods **Agar repository na ho, toh ye methods direct use nahi ho sakte:**

- `findById(id)` - ID ke basis pe data fetch karne ke liye.
- `save(entity)` - Data save/update karne ke liye.
- `deleteById(id)` - Record delete karne ke liye.
- `findByEmail(email)` - Email ke basis pe search karne ke liye.

16.1 Repository Ki Importance

- **Bridge Between Database and Service Layer:** Repository database aur `Service` / `Controller` layer ke beech mediator ka kaam karta hai.
- **Auto-Generated SQL Queries:** Spring Boot ka `JpaRepository` automatically SQL queries generate karta hai, based on method names. Iska matlab hai ki aapko manually queries likhne ki zaroorat nahi hoti.
- **Less Boilerplate Code:** Agar aap `JpaRepository` use karte hain, toh CRUD operations likhne ki zaroorat nahi hoti, kyunki Spring Boot khud inhe generate kar leta hai.

4. ‘model/‘ Folder

Express.js Comparison:

- Express.js me ‘models/‘ folder me database schemas define kiye jate hain.
- Example:

Express.js Example

```javascript
const userSchema = new mongoose.Schema({
    name: String,
    age: Number
});
```

Spring Boot me:

- ‘model/‘ folder me **database entities** define kiye jate hain.
- Ye folder **database tables** ko represent karta hai.
- Example:

User.java

```java
package com.example.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;

    // Getters and Setters
}
```

5. ‘resources/‘ Folder

Express.js Comparison:

- Express.js me static files (CSS, JS, images) ko root folder me rakha jata hai.
- Example:

Express.js Example

```javascript
app.use(express.static('public'));
```

Spring Boot me:

- ‘resources/‘ folder me **static files** aur **configuration files** rakhe jate hain.
- Ye folder **static content** aur **application settings** handle karta hai.
- Sub-folders:
- ‘static/‘: CSS, JS, images rakhe jate hain.
- ‘templates/‘: HTML templates rakhe jate hain.
- ‘application.properties‘: Configuration settings rakhe jate hain.

6. ‘DemoApplication.java‘

Express.js Comparison:
article tcolorbox

Point To Note

Express.js me ‘app.js‘ file server start karta hai same wahi kaam DemoApplication.java file v karta hai.

-

· Example:

Express.js Example

```
const express = require('express');
const app = express();
app.listen(3000, () => console.log('Server started on port
    ↪ 3000'));
```

Spring Boot me:
article tcolorbox

Point To Note

'DemoApplication.java' file server start karta hai.

·

Point To Note

Ye file **Spring Boot application ka entry point** hai.

·

· Example:

DemoApplication.java

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.
    ↪ SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Summary

Folder/File	Express.js Comparison	Spring Boot Role
'controller/'	'routes/'	API endpoints handle karta hai.
'service/'	'controllers/' ya modules	Business logic handle karta hai.
'repository/'	'models/' (database operations)	Database operations handle karta hai.
'model/'	'models/' (database schemas)	Database entities define karta hai.
'resources/'	Static files (CSS, JS, images)	Static files aur configuration rakhta hai.
'DemoApplication.java'	'app.js'	Server start karta hai.

Example Project Structure

```
src/
          main/
                java/
                      com/
                          example/
```

```
                              demo/
                                  controller/
                                        HelloController.java
                                  service/
                                        HelloService.java
                                  repository/
                                        UserRepository.java
                                  model/
                                        User.java
                                  DemoApplication.java
                resources/
                    static/
                    templates/
                    application.properties
          test/
              java/
                  com/
                      example/
                          demo/
                              DemoApplicationTests.java
```

================================

Spring Boot Explained Line-by-Line

1. Main Application File

File Name: ‘DemoApplication.java‘
Location: ‘src/main/java/com/example/demo/‘

DemoApplication.java

```java
package com.example.demo;  // Package declaration (folder structure ke hisab se)

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication  // Ye batata hai ki yeh Spring Boot application hai
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args); // Application start karta hai
    }
}
```

Explanation:

- `@SpringBootApplication`: Ye annotation Spring Boot ko batata hai ki yeh application ka main class hai. Isme **3 annotations** hote hain:
- `@SpringBootConfiguration`: Configuration define karta hai.
- `@EnableAutoConfiguration`: Automatic configuration enable karta hai.
- `@ComponentScan`: Package me components (controllers, services, etc.) ko scan karta hai.
- `SpringApplication.run()`: Ye method Spring Boot application ko start karta hai.

Express.js Me Relate Karo:
Express.js me `app.listen(3000)` ka kaam Spring Boot me `SpringApplication.run()` karta hai.

2. Creating a REST Controller

File Name: ‘HelloController.java‘
Location: ‘src/main/java/com/example/demo/controller/‘

HelloController.java

```java
package com.example.demo.controller;  // Package declaration

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

@RestController  // Ye batata hai ki yeh class ek REST
    ↪ controller hai
public class HelloController {

    @GetMapping("/hello")  // GET request ke liye endpoint
    public String sayHello() {
        return "Hello, Spring Boot!";  // Response return
    ↪ karta hai
    }
}
```

Explanation: article tcolorbox xcolor

Point To Note

- `@RestController`: Ye annotation batata hai ki yeh class ek REST controller hai. Yeh `@Controller` aur `@ResponseBody` ka combination hai.
- `@GetMapping("/hello")`: Ye batata hai ki `/hello` endpoint pe GET request handle karna hai.
- `sayHello()`: Ye method `/hello` endpoint pe request aane par call hota hai aur response return karta hai.

Point To Note

Express.js Me Relate Karo:
Express.js me `app.get('/hello')` ka kaam Spring Boot me `@GetMapping("/hello")` karta hai.

3. Adding a Service Layer

File Name: ‘HelloService.java‘
Location: ‘src/main/java/com/example/demo/service/‘

HelloService.java

```java
package com.example.demo.service;  // Package declaration

import org.springframework.stereotype.Service;  // Service
    ↪ annotation

@Service  // Ye batata hai ki yeh class ek service hai
public class HelloService {
    public String getHelloMessage() {
        return "Hello from Service!";  // Business logic
    }
}
```

Explanation:

Point To Note

- `@Service`: Ye annotation batata hai ki yeh class ek service hai. Isme business logic likha jata hai.
  Express.js Me Relate Karo:
  Express.js me tum `services` folder me business logic likhte ho, wahi kaam Spring Boot me `@Service` karta hai.

4. Using Service in Controller

File Name: 'HelloController.java'
Location: 'src/main/java/com/example/demo/controller/'

HelloController.java

```java
package com.example.demo.controller;

import com.example.demo.service.HelloService;  // Service
    ↪ import karo
import org.springframework.beans.factory.annotation.Autowired;
    ↪   // Dependency injection
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @Autowired  // Service ko inject karo
    private HelloService helloService;

    @GetMapping("/hello")
    public String sayHello() {
        return helloService.getHelloMessage();  // Service ka
    ↪ method call karo
    }
}
```

Explanation:

Point To Note

- `@Autowired`: Ye annotation Spring Boot ko batata hai ki `HelloService` ko automatically inject kare. Isse hume manually object banane ki zaroorat nahi hoti.
  Express.js Me Relate Karo:
  Express.js me tum manually `require('./services/HelloService')` karte ho, wahi kaam Spring Boot me `@Autowired` karta hai.

5. Database Connection

File Name: 'application.properties'
Location: 'src/main/resources/'

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb  # Database URL
spring.datasource.username=root  # Database username
spring.datasource.password=password  # Database password
spring.jpa.hibernate.ddl-auto=update  # Automatically update database schema
```

File Name: 'User.java' (Entity Class)
Location: 'src/main/java/com/example/demo/model/'

User.java

```java
package com.example.demo.model;

import javax.persistence.Entity;  // Ye batata hai ki yeh class database table ke saath map hogi
import javax.persistence.GeneratedValue;  // Auto-increment ke liye
import javax.persistence.GenerationType;  // Generation strategy ke liye
import javax.persistence.Id;  // Primary key ke liye

@Entity  // Ye batata hai ki yeh class database table ke saath map hogi
public class User {
    @Id  // Primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY)  // Auto-increment
    private Long id;
    private String name;

    // Getters and Setters
}
```

File Name: 'UserRepository.java' (Repository Interface)
Location: 'src/main/java/com/example/demo/repository/'

UserRepository.java

```java
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
    ↪  // JpaRepository provide karta hai CRUD operations

public interface UserRepository extends JpaRepository<User,
    ↪ Long> {
    // Yaha pe custom methods likh sakte hain
}
```

Explanation:

Point To Note

- `@Entity`: Ye annotation batata hai ki yeh class database table ke saath map hogi.
- `@Id`: Ye annotation batata hai ki yeh field primary key hai.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Ye batata hai ki primary key auto-increment hogi.
- `JpaRepository`: Ye interface provide karta hai CRUD operations (Create, Read, Update, Delete).
  Express.js Me Relate Karo:
  Express.js me tum `mongoose.model()` aur `User.find()` ka use karte ho, wahi kaam Spring Boot me `@Entity` aur `JpaRepository` karte hain.

6. Testing the Application

File Name: ‘DemoApplicationTests.java‘
Location: ‘src/test/java/com/example/demo/‘

DemoApplicationTests.java

```java
package com.example.demo;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest  // Ye annotation Spring Boot context load
    ↪ karta hai testing ke liye
class DemoApplicationTests {
    @Test
    void contextLoads() {
    }
}
```

Explanation:

- `@SpringBootTest`: Ye annotation Spring Boot context load karta hai testing ke liye.

Express.js Me Relate Karo:
Express.js me tum `Jest` ya `Mocha` use karte ho, wahi kaam Spring Boot me `@SpringBootTest` karta hai.
==================================

161em

Spring Boot Interceptors or Middleware: A Hinglish Guide

17 Introduction

Agar aap Express.js se Spring Boot aa rahe hain aur middleware ka concept samajhna chahte hain, toh ye guide aapke liye hai. Spring Boot me middleware ka kaam **Interceptors** karte hain. Ye guide aapko step-by-step interceptor banana aur use karne ka tarika samjhayega.

—

18 Step 1: Middleware (Interceptor) ka Concept

Point To Note

- **Express.js me middleware** ka kaam hota hai request aur response ke beech me kuch kaam karna, jaise logging, authentication, ya error handling. Ye ‘app.use()‘ se implement hota hai.
- **Spring Boot me** yehi kaam **Interceptor** karta hai. Interceptor bhi request aur response ke beech me kuch logic execute kar sakta hai.

—

19 Step 2: Express.js Middleware vs Spring Boot Interceptor

Feature	**Express.js Middleware**	**Spring Boot Interceptor**
Definition	‘app.use()‘ function	‘HandlerInterceptor‘ interface
Purpose	Request ko modify ya filter karna	Request ko modify ya filter karna
Execution	Request ke pehle ya response ke baad	Request ke pehle ya response ke baad
Example Use	Logging, Authentication, JWT Validation	Logging, Authentication, JWT Validation

—

20 Step 3: Middleware (Interceptor) Banana

Spring Boot me middleware banana **‘HandlerInterceptor‘ interface** ke through hota hai. Isme **3 methods** hote hain:

Point To Note

- **‘preHandle()‘** → Request ke process hone se pehle chalega (Express.js ke ‘app.use()‘ jaisa).
- **‘postHandle()‘** → Controller ka kaam hone ke baad chalega.
- **‘afterCompletion()‘** → Response bhejne ke baad chalega.

—

21 Step 4: LoggingInterceptor Banana

Yeh ek simple middleware hai jo har request ko log karega.

Example

LoggingInterceptor.java

```java
package com.example.project.interceptor;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component  // Ye Spring Boot ko batata hai ki ye ek component hai
public class LoggingInterceptor implements HandlerInterceptor {

    //      Step 1: preHandle()      Request aane se pehle chalega
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        System.out.println("       Incoming Request: " + request.getMethod() + " " + request.getRequestURI());
        return true;  // \textcolor{red}{Agar false return kare to request abort ho jayegi}
    }

    //      Step 2: postHandle()      Controller ka kaam hone ke baad chalega
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, org.springframework.web.servlet.ModelAndView modelAndView) {
        System.out.println("    Response Status: " + response.getStatus());
    }

    //      Step 3: afterCompletion()      Response bhejne ke baad chalega
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception exception) {
        System.out.println("       Request Completed!");
    }
}
```

—

Point To Note

22 Step 5: Middleware ko Register Karna

Middleware ko use karne ke liye use **register** karna padta hai. Ye kaam **‘WebMvcConfigurer‘** interface karta hai.

Example

WebConfig.java

```java
package com.example.project.config;

import com.example.project.interceptor.LoggingInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration  // Ye batata hai ki ye ek configuration file hai
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private LoggingInterceptor loggingInterceptor;  // Middleware ko inject kar rahe hain

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(loggingInterceptor);  // Middleware ko register kar rahe hain
    }
}
```

—

23 Step 6: Middleware ko Test Karna

Ab middleware kaam kar raha hai ya nahi, ye check karne ke liye ek simple API banate hain.

Example

UserController.java

```java
package com.example.project.controller;

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

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

    @GetMapping("/test")
    public String testEndpoint() {
        return "Middleware is working!";
    }
}
```

—

24 Step 7: JWT Authentication Middleware Banana

Agar tum **JWT Token validation** ka middleware banana chahte ho, to ‘preHandle()‘ me JWT check kar sakte ho.

Example

JwtInterceptor.java

```java
package com.example.project.interceptor;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component
public class JwtInterceptor implements HandlerInterceptor {

    private static final String SECRET_KEY = "mySecretKey"; // JWT Secret Key

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        String token = request.getHeader("Authorization");

        if (token == null || !token.startsWith("Bearer ")) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return false;  // {    Unauthorized request abort ho jayegi}
        }

        try {
            token = token.substring(7);  // "Bearer " hata rahe hain
            Claims claims = Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
            request.setAttribute("userId", claims.getSubject());
        } catch (Exception e) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return false;
        }

        return true;
    }
}
```

—

25 Final Summary

- **Express.js ka ‘app.use()‘ = Spring Boot ka ‘Interceptor‘.**
- **Interceptor ke 3 methods hote hain:**
- ‘preHandle()‘ → Request se pehle.

- ‘postHandle()‘ → Controller ke baad.
- ‘afterCompletion()‘ → Response ke baad.
- **Middleware ko register karne ke liye ‘WebMvcConfigurer‘ use karte hain.**
- **JWT validation jaise advanced tasks ke liye bhi interceptor use kar sakte hain.**

—

==================================

CRUD Based Spring Boot Application

Abhi main tumhe **complete CRUD-based Spring Boot application** ka code dunga, aur saath hi **JWT-based middleware (Interceptor)** bhi implement karunga. Har ek line ko comment me explain karunga, aur har keyword ka meaning bataunga. Saath hi, **file and folder structure** bhi dunga. Tumhe kuch assume nahi karna padega, kyuki main sab kuch zero se samjhaunga. Chalo shuru karte hain!

File and Folder Structure

```
src/
          main/
                java/
                      com/
                          example/
                              demo/
                                  config/  // Configuration files
                                        WebConfig.java
                                  controller/  // API endpoints
                                        UserController.java
                                  dto/  // Data Transfer Objects
                                        UserDTO.java
                                  exception/  // Custom exceptions
                                        GlobalExceptionHandler.java
                                  interceptor/  // JWT Interceptor
                                        JwtInterceptor.java
                                  model/  // Database entities
                                        User.java
                                  repository/  // Database operations
                                        UserRepository.java
                                  service/  // Business logic
                                        UserService.java
                                  DemoApplication.java  // Main application class
                resources/
                    application.properties  // Configuration file
          test/  // Test cases
```

Step 1: Add Dependencies

File Name: ‘pom.xml‘
Location: Project root folder

pom.xml

```xml
<dependencies>
    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 Database (In-memory database for testing) -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- JWT Library -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.9.1</version>
    </dependency>

    <!-- Lombok (Optional, for reducing boilerplate code) -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Explanation:

- `spring-boot-starter-web`: REST APIs banane ke liye.
- `spring-boot-starter-data-jpa`: Database operations ke liye.
- `h2`: In-memory database for testing.
- `jjwt`: JWT generate aur validate karne ke liye.
- `lombok`: Boilerplate code kam karne ke liye (optional).

Step 2: Configure Application Properties

File Name: ‘application.properties‘
Location: ‘src/main/resources/‘

application.properties

```
server.port=8080  # Server port
spring.datasource.url=jdbc:h2:mem:testdb  # H2 database URL
spring.datasource.driverClassName=org.h2.Driver  # H2 database driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect # H2 dialect
spring.h2.console.enabled=true  # H2 console enable karo
jwt.secret=mySecretKey  # JWT secret key
```

Explanation:

- `server.port`: Server ka port set karo.
- `spring.datasource.url`: H2 database ka URL.
- `jwt.secret`: JWT generate aur validate karne ke liye secret key.

Step 3: Create User Entity

File Name: ‘User.java‘
Location: ‘src/main/java/com/example/demo/model/‘

User.java

```java
package com.example.demo.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity  // Marks this class as a JPA entity, meaning it will be mapped to a database table
public class User {
    @Id  // Defines this field as the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY)  // Specifies that the ID should auto-increment
    private Long id;

    private String name; // Field for storing user name
    private String email; // Field for storing user email

    // Default Constructor (Required by JPA)
    public User() {}

    // Parameterized Constructor
    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    // toString() Method (For Debugging and Logging)
    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                '}';
    }
}
```

Explanation:

Point To Note

- `@Entity`: This annotation marks the class as a database entity, allowing it to be mapped to a table.
- `@Id`: Specifies that the field is the primary key.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Ensures that the primary key auto-increments with each new record.
- **Default Constructor**: Required by JPA for entity initialization.
- **Parameterized Constructor**: Allows creating a 'User' object with specific values.
- **Getters and Setters**: Used to access and modify private fields.
- **toString() Method**: Useful for debugging and logging user information.

Step 4: Create User Repository

File Name: 'UserRepository.java'
Location: 'src/main/java/com/example/demo/repository/'

UserRepository.java

```java
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User,
    ↪ Long> {
    // Yaha pe custom methods likh sakte hain
}
```

Explanation:

- `JpaRepository`: Ye interface provide karta hai CRUD operations (Create, Read, Update, Delete).

Step 5: Create User Service

File Name: 'UserService.java'
Location: 'src/main/java/com/example/demo/service/'

UserService.java

```java
package com.example.demo.service;

import com.example.demo.model.User;
import com.example.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service  // Ye batata hai ki yeh class ek service hai jo business logic handle karegi
public class UserService {

    @Autowired  // Ye automatically UserRepository ka instance inject karega
    private UserRepository userRepository;

    // Naya user create karne ke liye method
    public User createUser(User user) {
        return userRepository.save(user);  // User ko database me save karta hai
    }

    // Sabhi users ko retrieve karne ke liye method
    public List<User> getAllUsers() {
        return userRepository.findAll();  // Database se sabhi users fetch karega
    }

    // Specific user ko ID ke basis pe retrieve karne ka method
    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);  // Agar user milta hai to return karega, nahi to empty Optional
    }

    // User ko update karne ka method
    public User updateUser(Long id, User userDetails) {
        User user = userRepository.findById(id).orElseThrow(() -> new RuntimeException("User not found"));
        user.setName(userDetails.getName());
        user.setEmail(userDetails.getEmail());
        return userRepository.save(user);  // Updated user ko database me save karega
    }

    // User ko delete karne ka method
    public void deleteUser(Long id) {
        userRepository.deleteById(id);  // ID ke basis pe user ko delete karega
    }
}
```

Explanation:

Point To Note

- `@Service`: Ye annotation batata hai ki yeh class ek service component hai jo application ki business logic handle karegi.
- `@Autowired`: Ye annotation Spring Boot ko batata hai ki 'UserRepository' ka object automatically inject kiya jaye.
- `User user`: Ye ek 'User' class ka object hai jo ek particular user ka data store karega.
- `public List<User> getAllUsers()`: Ye method database se sabhi 'User' objects ki list return karega.
- `public Optional<User> getUserById(Long id)`: Ye method 'Optional¡User¿' return karega, jo ya to ek user object hoga agar user mil gaya, ya phir empty hoga agar user nahi mila.
- `Long id`: Ye 'Long' datatype ka ek variable hai jo user ki unique ID store karega.
- `User userDetails`: Ye ek 'User' object hai jo naye details store karega jab user update hoga.
- `userRepository.save(user)`: Ye method user object ko database me save karega.
- `userRepository.findById(id).orElseThrow()`: Ye database me ID ke basis pe user dhoondta hai. Agar nahi milta to exception throw karega.
- `userRepository.deleteById(id)`: Ye method user ko ID ke basis pe delete karega.

Step 6: Create User Controller

File Name: 'UserController.java'
Location: 'src/main/java/com/example/demo/controller/'

UserController.java

```java
package com.example.demo.controller;

import com.example.demo.model.User;
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController  // Ye batata hai ki yeh class ek REST controller hai
@RequestMapping("/users")  // Base URL for all endpoints in this controller
public class UserController {

    @Autowired  // UserService ko inject karo
    private UserService userService;

    // Create user
    @PostMapping  // POST request ke liye endpoint
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);  // User create karo
    }

    // Get all users
    @GetMapping  // GET request ke liye endpoint
    public List<User> getAllUsers() {
        return userService.getAllUsers();  // Sabhi users ko fetch karo
    }

    // Get user by ID
    @GetMapping("/{id}")  // GET request ke liye endpoint with ID
    public Optional<User> getUserById(@PathVariable Long id) {
        return userService.getUserById(id);  // User ko ID se fetch karo
    }

    // Update user
    @PutMapping("/{id}")  // PUT request ke liye endpoint with ID
    public User updateUser(@PathVariable Long id, @RequestBody User userDetails) {
        return userService.updateUser(id, userDetails);  // User ko update karo
    }

    // Delete user
    @DeleteMapping("/{id}")  // DELETE request ke liye endpoint with ID
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);  // User ko delete karo
    }
}
```

- `@RestController`: **Ye annotation batata hai ki yeh class ek REST controller hai jo API endpoints handle karegi.**
- `@RequestMapping("/users")`: **Iska matlab hai ki is class ke sare endpoints '/users' URL path se start honge.**
- `@Autowired`: **Ye automatically 'UserService' ka instance inject karega, jisse hum database operations perform kar sake.**
- `@PostMapping`, `@GetMapping`, `@PutMapping`, `@DeleteMapping`: **Yeh annotations define karte hain ki kis HTTP method se request aayegi.**
- `@PathVariable Long id`: **Iska use hota hai URL me diye gaye 'id' ko method ke parameter me fetch karne ke liye.**
- `@RequestBody User user`: **Iska use hota hai incoming JSON data ko 'User' object me convert karne ke liye.**

Step 7: Create JWT Interceptor

File Name: 'JwtInterceptor.java'
Location: 'src/main/java/com/example/demo/interceptor/'

JwtInterceptor.java

```java
package com.example.demo.interceptor;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component  // Ye batata hai ki yeh class ek Spring component hai
public class JwtInterceptor implements HandlerInterceptor {

    private static final String SECRET_KEY = "mySecretKey"; // JWT secret key

    @Override  // Ye method har request ke pehle call hota hai
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String token = request.getHeader("Authorization");  // Request se token fetch karo

        if (token == null || !token.startsWith("Bearer ")) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid token");  // Invalid token
            return false;
        }

        token = token.substring(7);  // "Bearer " ko remove karo
        try {
            Claims claims = Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();  // Token validate karo
            request.setAttribute("userId", claims.getSubject());  // User ID ko request me set karo
        } catch (Exception e) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid token");  // Invalid token
            return false;
        }

        return true;  // Request ko agle interceptor ya controller ko pass karo
    }
}
```

Explanation:

- `@Component`: Ye annotation batata hai ki yeh class ek Spring component hai.
- `HandlerInterceptor`: Ye interface provide karta hai methods jo har request ke pehle, baad me, ya completion pe call hote hain.
- `preHandle()`: Ye method har request ke pehle call hota hai. Agar ye `true` return kare, toh request agle interceptor ya controller tak jayegi. Agar `false` return kare, toh request ruk jayegi.

Step 8: Register JWT Interceptor

File Name: ‘WebConfig.java‘
Location: ‘src/main/java/com/example/demo/config/‘

WebConfig.java

```java
package com.example.demo.config;

import com.example.demo.interceptor.JwtInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.
    ↪ InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.
    ↪ WebMvcConfigurer;

@Configuration  // Ye batata hai ki yeh class ek configuration
    ↪  hai
public class WebConfig implements WebMvcConfigurer {

    @Autowired  // JwtInterceptor ko inject karo
    private JwtInterceptor jwtInterceptor;

    @Override  // Ye method interceptors ko register karta hai
    public void addInterceptors(InterceptorRegistry registry)
    ↪ {
        registry.addInterceptor(jwtInterceptor).
    ↪ addPathPatterns("/users/**");  // JWT Interceptor ko
    ↪ register karo
    }
}
```

Explanation:

- `@Configuration`: Ye annotation batata hai ki yeh class ek configuration hai.
- `WebMvcConfigurer`: Ye interface provide karta hai methods jo web configuration ke liye use hote hain.
- `addInterceptors()`: Ye method interceptors ko register karta hai.

Step 9: Create Global Exception Handler

File Name: ‘GlobalExceptionHandler.java‘
Location: ‘src/main/java/com/example/demo/exception/‘

GlobalExceptionHandler.java

```java
package com.example.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice  // Ye annotation batata hai ki yeh class global exception handling ke liye hai
public class GlobalExceptionHandler {

    @ExceptionHandler(RuntimeException.class)  // Ye method RuntimeException handle karega
    public ResponseEntity<String> handleRuntimeException(RuntimeException e) {
        return new ResponseEntity<>(e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);  // Error message return karo
    }
}
```

Explanation:

- `@ControllerAdvice`: Ye annotation batata hai ki yeh class global exception handling ke liye hai.
- `@ExceptionHandler`: Ye annotation batata hai ki yeh method specific exception handle karega.

Step 10: Run the Application

File Name: ‘DemoApplication.java‘
Location: ‘src/main/java/com/example/demo/‘

DemoApplication.java

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication  // Ye batata hai ki yeh Spring Boot application hai
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);  // Application start karo
    }
}
```

Explanation:

Point To Note

- `@SpringBootApplication`: Ye annotation Spring Boot ko batata hai ki yeh application ka main class hai.
- `SpringApplication.run()`: Ye method Spring Boot application ko start karta hai.

=================================

Spring Boot Project Structure (With Line-by-Line Code Explanation)

Spring Boot Project Structure

```
src/main
          java/api/aidiph/com
                client
                config
                controller
                dto
                entity
                enums
                exception
                mapper
                repository
                scheduler
                service
                utils
          resources
```

1 Feign client (Express.js me External API calls ka kaam)

Yeh folder doosre microservices se baat karne ke liye hota hai using `@FeignClient`
Express.js Comparison: Jaise `axios` ka use hota hai doosre services ko call karne ke liye

Spring Boot Code (With Comments)

```java
// Foreign Client ka use karke doosre service se baat karne ke liye
    ↪ interface banaya
@FeignClient(name = "user-service", url = "http://localhost:8081")
public interface UserClient {

    // Yeh endpoint `/users/{id}` ko call karega aur user ka data
    ↪ return karega
    @GetMapping("/users/{id}")
    UserResponse getUserById(@PathVariable Long id);
}
```

Express.js (Axios ke saath)

```javascript
const axios = require("axios");

// Function jo doosre service se data lega
async function getUserById(id) {
    const response = await axios.get(‘http://localhost:8081/users/${id}‘);
    return response.data;
}
```

Microservices vs Monolithic & Feign Client vs Axios (Microservices Communication)

Part 1: Microservices vs Monolithic (Comparison)

Feature	Monolithic App	Microservices
Architecture	Ek badi app hoti hai jisme saare modules ek saath hote hain.	Chhoti-chhoti independent servic hain jo ek doosre se API ke th baat karti hain.
Scalability	Slow, kyunki pura system ek saath scale karna padta hai.	Fast & Easy, kyunki sirf jo s chahiye wahi scale hoti hai.
Failure Impact	Agar ek module fail ho jaye to pura system down ho sakta hai.	Agar ek service fail ho jaye to baa vices kaam karti rahengi.
Technology Freedom	Sirf ek hi technology use hoti hai.	Har service alag technology us sakti hai (Jaise ek service Node doosri Python me ho sakti hai).
Development Speed	Slow, kyunki ek hi codebase me sab kuch likhna padta hai.	Fast, kyunki alag-alag teams ala vices develop kar sakti hain.

Example:

- **Monolithic:** Ek single app jo ‘User‘, ‘Order‘, aur ‘Payment‘ system ko ek saath manage karti hai.
- **Microservices:** ‘User Service‘, ‘Order Service‘, aur ‘Payment Service‘ alag hote hain aur API ke through baat karte hain.

Part 2: Feign Client vs Axios (Microservices Communication)

Agar **Order Service** ko **User Service** se data lena ho to ye kaise hoga?

1. Feign Client in Spring Boot

Spring Boot me Feign Client use hota hai jo doosre Microservices se baat karta hai bina manually HTTP request likhe.

```
@FeignClient(name = "user-service", url = "http://localhost:8081")
public interface UserClient {
    @GetMapping("/users/{id}")
    UserResponse getUserById(@PathVariable Long id);
}
```

- `@FeignClient(name = "user-service")` automatically request send karega.
- `getUserById()` method `User Service` se user ka data fetch karega.

2. Axios in Express.js

Express.js me Feign Client nahi hota, isliye Axios ka use hota hai API call ke liye.

```
const axios = require("axios");

async function getUserById(id) {
    try {
        const response = await axios.get(`http://localhost:8081/users/${id}`);
        return response.data;
    } catch (error) {
        console.error("Error fetching user data:", error);
        return null;
    }
}
```

- `axios.get()` request bhejta hai aur data fetch karta hai.
- `try-catch` block error handle karne ke liye use hota hai.

Feign Client vs Axios (Comparison Table)

Feature	Feign Client (Spring Boot)	Axios (Express.js)
Communication	Microservices ko easy connect karta hai bina manually HTTP request likhe.	Manually HTTP request likhni padti hai.
Code Simplicity	Code simple aur clean hota hai.	Extra HTTP handling likhni padti hai.
Error Handling	Spring Boot automatically error handle karta hai.	Manually error handle karna padta hai.
Performance	Fast aur optimized.	Extra HTTP request handling overhead hota hai.
Technology	Spring Cloud Feign ka use karta hai.	Axios ek external library hai.

Conclusion

- **Monolithic Apps** simple hoti hain par scale karna mushkil hota hai.
- **Microservices Architecture** independent services banakar scalability aur reliability improve karta hai.
- **Feign Client** (Spring Boot) microservices communication ko easy banata hai bina manually API calls likhe.
- **Axios** (Express.js) manually API request send karta hai, par zyada control deta hai.

Real-Life Example

- **Amazon:** Order Service, Payment Service, aur User Service alag hote hain.
- **Netflix:** Movie Service, User Service, aur Recommendation Service alag hote hain.
- **Uber:** Ride Booking, User Management, aur Payment System alag-alag services hain.

2 config (Express.js me Middleware aur Configurations ka kaam)

Yeh folder Spring Boot me middleware, security, aur settings store karta hai
Express.js Comparison: Jaise `server.js` me middleware aur configs likhte hain

Spring Boot Code (With Comments)

```
// Configuration class banayi jo Spring Boot ko bataegi ki CORS allow karna hai
@Configuration
public class CorsConfig {

    @Bean // Bean register kar raha hai jo ek object return karega
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                // Sare endpoints ke liye CORS allow karega
                registry.addMapping("/**").allowedOrigins("*");
            }
        };
    }
}
```

Express.js (Middleware ke saath)

```
const cors = require("cors");

// Middleware use kiya jo CORS enable karega
app.use(cors());
```

3 controller (Express.js me Routes/Controllers ka kaam)

Yeh request handle karta hai aur REST APIs define karta hai
Express.js Comparison: Jaise `routes/jobRoutes.js` me `app.get()` use karte hain

Spring Boot Code (With Comments)

```
// REST Controller banaya jo "/jobs" endpoint ko handle karega
@RestController
@RequestMapping("/jobs")
public class JobController {

    // GET request ko handle karega jo "/jobs/{id}" par aayegi
    @GetMapping("/{id}")
    public Job getJobById(@PathVariable Long id) {
        // Service se job ka data leke return karega
        return jobService.getJobById(id);
    }
}
```

Express.js (Router)

```
const express = require("express");
const router = express.Router();

// GET route jo "/jobs/:id" ko handle karega
router.get("/jobs/:id", (req, res) => {
    res.json({ id: req.params.id, title: "Software Engineer" });
});

module.exports = router;
```

================================
Understanding and Using DTO (Data Transfer Object) in Spring Boot

Why Use DTO (Data Transfer Object)?

DTO ek special class hoti hai jo sirf data transfer ke liye use hoti hai. Iska main kaam hai ki hum sirf wahi data API response me bhejein jo zaroori hai, bina database ka pura object expose kiye.

Bina DTO ke Problem

Agar aap directly `UserSignupEntity` return karoge toh API response me password jaise sensitive data bhi chala jayega, jo ek bad practice hai.

DTO ke Fayde

- Sirf zaroori fields bhejo (e.g., `success`, `message`).
- Sensitive data hide hota hai, security improve hoti hai.
- Code clean aur maintainable banta hai.

DTO Implementation in Spring Boot

Project Folder Structure

```
src/
          main/java/com/example/demo/
                controller/     <-- API logic
                service/        <-- Business logic
                entity/         <-- Database Entity
                dto/            <-- DTO Classes
                repository/     <-- Database Queries
                DemoApplication.java
```

Step 1: DTO Class Creation

File: `src/main/java/com/example/demo/dto/ApiResponse.java`

```java
package com.example.demo.dto;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ApiResponse {
    private boolean success;
    private String message;
}
```

Step 2: Controller Update

File: `src/main/java/com/example/demo/controller/UserController.java`

```java
package com.example.demo.controller;

import com.example.demo.dto.ApiResponse;
import com.example.demo.entity.UserSignupEntity;
import com.example.demo.service.UserSignupService;
import org.springframework.web.bind.annotation.*;

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

    private final UserSignupService userSignupService;

    public UserController(UserSignupService userSignupService) {
        this.userSignupService = userSignupService;
    }

    @PostMapping("/create")
    public ApiResponse createUser(@RequestBody UserSignupEntity user) {
        try {
            userSignupService.registerUser(user);
            return new ApiResponse(true, "User created successfully.");
        } catch (Exception e) {
            return new ApiResponse(false, "User creation failed: " + e.
    ↪ getMessage());
        }
    }
}
```

Step 3: Service Class Update

File: `src/main/java/com/example/demo/service/UserSignupService.java`

```java
package com.example.demo.service;

import com.example.demo.entity.UserSignupEntity;
import com.example.demo.repository.UserSignupRepository;
import org.springframework.stereotype.Service;

@Service
public class UserSignupService {

    private final UserSignupRepository userSignupRepository;

    public UserSignupService(UserSignupRepository userSignupRepository) {
        this.userSignupRepository = userSignupRepository;
    }

    public UserSignupEntity registerUser(UserSignupEntity user) {
        return userSignupRepository.save(user);
    }
}
```

Step 4: Repository Class

File: `src/main/java/com/example/demo/repository/UserSignupRepository.java`

```java
package com.example.demo.repository;

import com.example.demo.entity.UserSignupEntity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserSignupRepository extends JpaRepository<
    ↪ UserSignupEntity, Long> {
}
```

Step 5: Entity Class Update

File: `src/main/java/com/example/demo/entity/UserSignupEntity.java`

```java
package com.example.demo.entity;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import java.sql.Timestamp;
import java.time.Instant;

```

```java
@Entity
@Table(name = "users")
@Getter
@Setter
public class UserSignupEntity {

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

    @Column(unique = true, nullable = false, length = 100)
    private String email;

    @Column(nullable = false, length = 255)
    private String password;

    @Column(nullable = false, length = 20)
    private String role;

    @Column(name = "created_at", updatable = false)
    private Timestamp createdAt;

    @PrePersist
    protected void onCreate() {
        this.createdAt = Timestamp.from(Instant.now());
    }
}
```

API Testing

Request: POST `/api/users/create`

```json
{
    "email": "test@gmail.com",
    "password": "123456",
    "role": "user"
}
```

Success Response

```json
{
    "success": true,
    "message": "User created successfully."
}
```

Failure Response

```json
{
    "success": false,
    "message": "User creation failed: Duplicate email found."
}
```

Conclusion

- DTO ka use karke API ka response clean aur secure banaya.
- Directly `UserSignupEntity` return karne se bach gaye.
- Service aur Repository classes alag rakhi jo code maintainable banayegi.
- Future me easily modify kar sakte hain (DTO structure badal ke).

5 entity (Express.js me Mongoose Models ya Sequelize ka kaam)

Entity wo class hoti hai jo database table ko represent karti hai
Express.js Comparison: Jaise `Mongoose` models ya `Sequelize` models use karte hain

Spring Boot Code (With Comments)

```
@Entity // Is class ko DB table ke roop me define kar raha hai
@Table(name = "jobs") // Table ka naam "jobs" rakha hai
public class Job {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto
    ↪ Increment ID
    private Long id;

    private String title; // Job ka title
}
```

Express.js (Mongoose Model)

```
const mongoose = require("mongoose");

// Job model banaya jo MongoDB collection ko represent karega
const jobSchema = new mongoose.Schema({
    title: String,
});

const Job = mongoose.model("Job", jobSchema);
```

6 enums (Constant values ko define karne ke liye)

Enums predefined values store karte hain (e.g., Status, Roles)
Express.js Comparison: Jaise `constants.js` file me constants define karte hain

Spring Boot Code (With Comments)

```
public enum JobStatus {
    ACTIVE, // Job active hai
    CLOSED, // Job closed hai
    PENDING; // Job pending hai
}
```

Express.js (Constants File)

```
const JOB_STATUS = {
    ACTIVE: "ACTIVE",
    CLOSED: "CLOSED",
    PENDING: "PENDING",
};
```

7 exception (Express.js me Error Handling ka kaam)

Custom exceptions aur error handling ke liye hota hai
Express.js Comparison: Jaise `app.use((err, req, res, next) => { ... })`

Spring Boot Code (With Comments)

```java
@ResponseStatus(HttpStatus.NOT_FOUND) // 404 error return karega
public class JobNotFoundException extends RuntimeException {
    public JobNotFoundException(String message) {
        super(message); // Error message set karega
    }
}
```

Express.js (Custom Error Middleware)

```javascript
app.use((err, req, res, next) => {
    res.status(500).json({ error: err.message });
});
```

Point To Note

resources (Static Files, .env, Configs)

Yeh folder static files, application properties aur templates store karta hai
Express.js Comparison: Jaise `public/`, `.env`, `views/` Express.js me hota hai

Conclusion

Spring Boot me structure alag hai but kaam same hai jo Express.js me hota hai
================================

8 mapper (Entity ko DTO me convert karna, Express.js me response format change karna)

Mapper ka kaam hota hai database entity ko DTO me convert karna aur wapas
Express.js Comparison: Jaise manually response ka format change karna

Spring Boot Code (With Comments)

```java
// Job entity ko DTO me convert karne ka helper class
public class JobMapper {

    // Static method jo Job entity ko JobResponseDTO me convert
    ↪ karega
    public static JobResponseDTO toDTO(Job job) {
        return new JobResponseDTO(job.getId(), job.getTitle());
    }
}
```

Yeh kaam har request me karne se bachne ke liye mapper banate hain

Express.js (Manually Response Map karna)

```javascript
// Function jo job entity ko response DTO format me change karega
function jobToDTO(job) {
    return { id: job.id, title: job.title };
}
```

```

// API route me manually convert karna hoga
app.get("/jobs/:id", async (req, res) => {
    const job = await Job.findById(req.params.id);
    res.json(jobToDTO(job)); // Response format ko DTO banaya
});
```

Express.js me yeh manually har route me karna padta hai, isliye Spring Boot mapper better hai

9 repository (Database Queries, Express.js me Mongoose ya Sequelize ka kaam)

Repository layer DB queries execute karti hai
Express.js Comparison: Jaise `Mongoose.find()` ya `Sequelize.findAll()` use karte hain

Spring Boot Code (With Comments)

```
// Repository ka kaam DB queries ko handle karna hai
@Repository
public interface JobRepository extends JpaRepository<Job, Long> {
    // JpaRepository CRUD operations provide karega
}
```

Yeh ‘findById()‘, ‘save()‘, ‘delete()‘ jaise methods automatically provide karta hai

Express.js (Mongoose Queries)

```
// Express.js me Mongoose ka use karke DB se data fetch karna
const jobs = await Job.find();
```

Spring Boot ka ‘JpaRepository‘ automatic methods provide karta hai, jabki Express.js me manually likhna padta hai

Point To Note

scheduler (Background me Periodic Tasks, Express.js me Cron Jobs)

Scheduler background me automatically kuch operations run karta hai
Express.js Comparison: Jaise `node-cron` ka use periodic tasks run karne ke liye

Spring Boot Code (With Comments)

```
// Scheduler jo har raat 12 baje purane jobs delete karega
@Scheduled(cron = "0 0 * * * ?") // CRON job format (every day at
    ↪ midnight)
public void cleanOldJobs() {
    jobService.deleteOldJobs(); // Service call karke purane jobs
    ↪ delete karega
}
```

Spring Boot me ‘@Scheduled‘ annotation se background task run kar sakte hain

Express.js (Cron Job)

```
const cron = require("node-cron");

// Har raat 12 baje kaam chalane ke liye cron job setup kiya
cron.schedule("0 0 * * *", () => {
    console.log("Running scheduled task: Deleting old jobs...");
});
```

Express.js me 'node-cron' package ka use hota hai

resources (Express.js me Views, Static Files, .env ka kaam)

Yeh folder static files, application properties aur templates store karta hai
Express.js Comparison: Jaise `public/`, `.env`, `views/`
Spring Boot aur Express.js dono me yeh alag folder me rakha jata hai for better organization

Spring Boot ('resources/application.properties')

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
```

Express.js ('.env' file)

```
PORT=3000
DB_URL=mongodb://localhost:27017/mydb
```

Dono ka kaam same hai, bas syntax alag hai

Conclusion

Spring Boot me 'mapper', 'repository', aur 'scheduler' alag se define hote hain jo Express.js me manually likhna padta hai
Agar aur koi doubt hai ya kisi topic ka aur deep explanation chahiye to batao!
==============================

Spring Boot Annotations Explained

1. '@FeignClient'

- **Spring Boot**: '@FeignClient' ek aisa annotation hai jo aapko dusre microservices ya REST APIs ko easily call karne mein help karta hai. Ye ek tarah ka client banata hai jo dusre service ke saath communicate karta hai.

```
@FeignClient(name = "user-service", url = "http://localhost:8081")
public interface UserServiceClient {
    @GetMapping("/users/{id}")
    User getUserById(@PathVariable Long id);
}
```

- 'name = "user-service"': Ye service ka naam hai, jisse aap identify kar sakte hain.
- 'url = "http://localhost:8081"': Ye uska base URL hai jahan par ye service available hai.
- **Express.js mein relate karein**: Express.js mein aap 'axios' ya 'fetch' use karte hain dusre APIs ko call karne ke liye. '@FeignClient' bhi wahi kaam karta hai, lekin Spring

Boot mein ye ek declarative way hai, matlab aapko manually HTTP calls nahi likhne padte, Spring khud handle karta hai.
article tcolorbox xcolor fontawesome listings

2. ‘@PathVariable‘

Point To Note

Spring Boot: ‘@PathVariable‘ ka use URL se variable values extract karne ke liye hota hai. Jaise agar aapke paas ek endpoint hai ‘/users/id‘, toh ‘id‘ wala part extract karne ke liye ‘@PathVariable‘ use karte hain.

```
@GetMapping("/users/{id}")
public User getUserById(@PathVariable Long id) {
    // id ka use karke user fetch karo
}

```

· **Express.js mein relate karein**: Express.js mein aap ‘req.params‘ use karte hain URL se variables extract karne ke liye. Jaise:

```
app.get('/users/:id', (req, res) => {
    const id = req.params.id;
    // id ka use karke user fetch karo
});

```

3. ‘@GetMapping‘

-

Point To Note

Spring Boot: ‘@GetMapping‘ ka use GET request ke liye endpoint define karne ke liye hota hai. Jaise:

```
@GetMapping("/users")
public List<User> getAllUsers() {
    // sab users return karo
}
```

- Agar aap ‘@GetMapping‘ mein kuch path specify nahi karte, toh ye default route ke liye use hota hai. Matlab agar aapka base URL hai ‘http://localhost:8080‘, toh ‘@GetMapping‘ wala method ‘http://localhost:8080‘ par GET request handle karega.
- **Express.js mein relate karein**: Express.js mein aap ‘app.get()‘ use karte hain GET request handle karne ke liye. Jaise:

```
app.get('/users', (req, res) => {
    // sab users return karo
});
```

4. ‘@PostMapping‘

-

Point To Note

Spring Boot: ‘@PostMapping‘ ka use POST request ke liye endpoint define karne ke liye hota hai. Jaise:

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    // user ko create karo
}
```

- **Express.js mein relate karein**: Express.js mein aap ‘app.post()‘ use karte hain POST request handle karne ke liye. Jaise:

```
app.post('/users', (req, res) => {
    const user = req.body;
    // user ko create karo
});
```

5. ‘@GetMapping‘ without path

-

Point To Note

Spring Boot: Agar aap ‘@GetMapping‘ mein koi specific path specify nahi karte, toh ye default route ke liye use hota hai. Matlab agar aapka base URL hai ‘http://localhost:8080‘, toh ‘@GetMapping‘ wala method ‘http://localhost:8080‘ par GET request handle karega.

```
@GetMapping
public String home() {
    return "Welcome to the home page!";
}
```

- **Express.js mein relate karein**: Express.js mein aap ‘app.get('/')‘ use karte hain root route ke liye. Jaise:

```
app.get('/', (req, res) => {
    res.send('Welcome to the home page!');
});
```

6. ‘@PostMapping‘ without path

-

Point To Note

Spring Boot: Agar aap ‘@PostMapping‘ mein koi specific path specify nahi karte, toh ye default route ke liye use hota hai. Matlab agar aapka base URL hai ‘http://localhost:8080‘, toh ‘@PostMapping‘ wala method ‘http://localhost:8080‘ par POST request handle karega.

```
@PostMapping
public String createSomething(@RequestBody SomeObject obj) {
    // kuch create karo
}
```

- **Express.js mein relate karein**: Express.js mein aap ‘app.post(’/’)‘ use karte hain root route ke liye POST request handle karne ke liye. Jaise:

```
app.post('/', (req, res) => {
    const obj = req.body;
    // kuch create karo
});
```

Summary

- ‘@FeignClient‘ -¿ Express.js mein ‘axios‘ ya ‘fetch‘ ki tarah hai, lekin declarative way mein.
- ‘@PathVariable‘ -¿ Express.js mein ‘req.params‘ ki tarah hai.
- ‘@GetMapping‘ -¿ Express.js mein ‘app.get()‘ ki tarah hai.
- ‘@PostMapping‘ -¿ Express.js mein ‘app.post()‘ ki tarah hai.
- ‘@GetMapping‘ aur ‘@PostMapping‘ without path -¿ Express.js mein root route (‘’/’‘) ke liye use hota hai.

==================================
Spring Boot vs Express.js: Missing Topics in Real-life Projects

Missing Topics in Your Notes

Real-life Spring Boot projects me jo topics zaroori hain, unko Express.js se relate karke samjhaya gaya hai. Har line ka breakdown bhi diya gaya hai.

1. Application Layer Architecture

Spring Boot me projects ek **specific architecture** follow karte hain:

- **MVC (Model-View-Controller)**
- **3-Tier Architecture (Controller → Service → Repository)**
- **Hexagonal Architecture (Port Adapter Model)**

Spring Boot - 3-Tier Architecture

```
@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.getUserById(id);
    }
}
```

Express.js - Same Structure

```javascript
// Controller
app.get("/users/:id", async (req, res) => {
    const user = await userService.getUserById(req.params.id);
    res.json(user);
});
```

2. Security (Spring Security, JWT, OAuth2)

Authentication aur authorization real-world applications me mandatory hoti hai!

Spring Boot - JWT Implementation

```java
@Service
public class JwtUtil {
    private static final String SECRET_KEY = "mySecretKey";

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis()
    ↪ + 86400000))
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
                .compact();
    }
}
```

Express.js - JWT Implementation

```javascript
const jwt = require("jsonwebtoken");
function generateToken(username) {
    return jwt.sign({ username }, "mySecretKey", { expiresIn: "1d"
    ↪ });
}
```

3. Global Exception Handling

Production-level applications me centralized error handling zaroori hota hai!

Spring Boot - Global Exception Handler

```java
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(
    ↪ RuntimeException e) {
        return new ResponseEntity<>(e.getMessage(), HttpStatus.
    ↪ INTERNAL_SERVER_ERROR);
    }
}
```

Express.js - Error Middleware

```
app.use((err, req, res, next) => {
    res.status(500).json({ error: err.message });
});
```

4. Docker & Deployment

Real-world applications ko containerized aur deploy karna ek must-have skill hai!

Spring Boot - Dockerfile

```
FROM openjdk:17
WORKDIR /app
COPY target/myapp.jar myapp.jar
CMD ["java", "-jar", "myapp.jar"]
```

Express.js - Dockerfile

```
FROM node:16
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD ["node", "server.js"]
```

5. Logging & Monitoring

Logs aur monitoring real-time debugging ke liye crucial hote hain!

Spring Boot - Logging

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
public class JobController {
    private static final Logger logger = LoggerFactory.getLogger(
    ↪ JobController.class);

    @GetMapping("/jobs/{id}")
    public Job getJobById(@PathVariable Long id) {
        logger.info("Fetching job with id: " + id);
        return jobService.getJobById(id);
    }
}
```

Express.js - Logging with Morgan

```
const morgan = require("morgan");
app.use(morgan("dev"));
```

==================================

Annotations in Spring Boot

26 1. @Configuration

Kya Hai Ye?

`@Configuration` ek annotation hai jo Spring Boot ko batata hai ki ye class ek configuration file ki tarah kaam karegi jisme hum beans define kar sakte hain.

Agar Hum Na Dein Toh Kya Hoga?

Agar @Configuration nahi diya toh Spring is class ko ek configuration file nahi samjhega aur @Bean methods execute nahi karega.

Example:

```java
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

27 2. @EnableFeignClients

Kya Hai Ye?

Feign ek declarative REST client hai jo external APIs se connect hone ke liye use hota hai.

Agar Hum Na Dein Toh Kya Hoga?

Agar ye annotation na ho toh Feign clients kaam nahi karenge aur API calls fail ho jayengi.

Example:

```java
@EnableFeignClients(basePackages = "com.aidiph.api.central.client")
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

28 3. @EnableWebSecurity aur @EnableMethodSecurity

Kya Hai Ye?

`@EnableWebSecurity` authentication aur authorization handle karta hai. `@EnableMethodSecurity` method-level security enable karta hai.

Agar Hum Na Dein Toh Kya Hoga?

Agar ye annotations nahi diye toh security disable ho jayegi aur endpoints unprotected rahenge.

Example:

```java
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated();
    }
}
```

29 4. @SecurityRequirement

Kya Hai Ye?

Swagger documentation me batata hai ki API ko authorization ki zaroorat hai.

Agar Hum Na Dein Toh Kya Hoga?

Swagger me API ke liye authorization ka option nahi dikhega.

Example:

```java
@SecurityRequirement(name = "Authorization")
public class MyController {
    // API methods
}
```

30 5. @RequestMapping

Kya Hai Ye?

Isse hum ek class ya method ke liye URL define kar sakte hain jo HTTP request handle karega.

Agar Hum Na Dein Toh Kya Hoga?

Agar @RequestMapping nahi diya toh Spring Boot request ko is controller se match nahi karega aur error milega 404 Not Found.

Example:

```java
@RestController
@RequestMapping(value = "central/permissions")
public class PermissionController {
    @GetMapping
    public List<Permission> getPermissions() {
        return List.of(new Permission("READ"), new Permission("WRITE"));
    }
}
```

Miscellaneous Annotations in Spring Boot

31 Miscellaneous Annotations

31.1 @Slf4j

Kya Hai Ye?

`@Slf4j` ek Lombok annotation hai jo **automatic logging setup** karta hai. Isse `log.info()` aur `log.error()` jaise logging methods bina manually `Logger` banaye use ho sakte hain.

Agar Na Dein Toh Kya Hoga?

- `log.info("Message")` ya `log.error("Error")` likhne par **error aayega** kyunki logger define nahi hoga.
- Har class me **manually logger setup** karna padega jo **time-consuming** hoga.

Example:

```java
@Slf4j
public class MyClass {
    public void test() {
        log.info("This is a log message.");
    }
}
```

Agar @Slf4j Na Ho:

```java
public class MyClass {
    private static final Logger log = LoggerFactory.getLogger(
    ↪ MyClass.class);

    public void test() {
        log.info("This is a log message.");
    }
}
```

—

31.2 @Validated

Kya Hai Ye?

Point To Note

`@Validated` annotation **Spring Boot ke validation framework** ko enable karta hai. Isse hum input data ko validate kar sakte hain bina manually `if-else` check likhe.

Agar Na Dein Toh Kya Hoga?

- `@Valid` ya `@NotNull` jaise annotations **kaam nahi karenge**.
- Invalid data ko manually check karna padega using `if-else`.

Example:

```java
@Validated
public class UserController {
    public ResponseEntity<?> createUser(@Valid @RequestBody
    ↪ UserRequest request) {
        // Handle request
    }
}
```

—

31.3 @RestController

Kya Hai Ye?

`@RestController` **Spring Boot me API endpoints create karne** ke liye use hota hai.

Agar Na Dein Toh Kya Hoga?

- Agar sirf `@Controller` likha aur `@ResponseBody` nahi diya toh **method ka return HTML page render karne ki koshish karega**.
- API request ke response me **JSON return nahi hoga**.

Example:

```java
@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping
    public String getUser() {
        return "Hello User";
    }
}
```

—

31.4 @RequiredArgsConstructor

Kya Hai Ye?
Lombok annotation hai jo `final` fields ke liye **automatic constructor generate** karta hai.
Agar Na Dein Toh Kya Hoga?

- **Hume manually constructor likhna padega** jisme `final` fields ka initialization karein.
- Agar **class me multiple dependencies ho**, toh manually constructor likhna **tedious ho sakta hai**.

Example:

```java
@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;
}
```

—

Point To Note

31.5 @CrossOrigin

Kya Hai Ye?
`@CrossOrigin` **CORS (Cross-Origin Resource Sharing)** enable karta hai.
Agar Na Dein Toh Kya Hoga?

- Frontend agar alag server pe chal raha hai toh **CORS error aayega**.
- Browser request ko **block** kar dega.

Example:

```java
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RestController
public class MyController {
    @GetMapping("/data")
    public String getData() {
        return "Hello";
    }
}
```

32 Conclusion

Yeh miscellaneous annotations **Spring Boot applications ko maintainable aur efficient banati hain**.

Point To Note

Agar hum inhe na dein toh:

- Logging **manually setup** karna padega `@Slf4j` na hone par.
- Input validation **manually handle** karni padegi `@Validated` na hone par.
- JSON response **properly return nahi hoga** `@RestController` na hone par.
- **Constructor manually likhna padega** `@RequiredArgsConstructor` na hone par.
- **CORS error aayega** `@CrossOrigin` na hone par.

Spring Boot vs Express.js Annotations

7. @Operation, @ApiResponses, @ApiResponse

Spring Boot Explanation:

- **@Operation**: Ye annotation Swagger/OpenAPI documentation ke liye hoti hai. Isse aap API endpoint ko describe kar sakte ho, jaise ki summary aur description dena.
- **@ApiResponses**: Ye annotation ek list hoti hai jo aapke API ke possible responses ko specify karti hai.
- **@ApiResponse**: Ye annotation ek specific response ko define karti hai, jaise response code aur description.

Express.js Comparison: Express.js mein, jab aap API banate ho, tab aap comments ya Swagger jaise tools ka use karte ho API documentation ke liye.

Code Example:

```
@Operation(summary = "Add User Career Profile", description = "Creates a new user caree
@ApiResponses(value = {
    @ApiResponse(responseCode = "201", description = "User career profile created succe
    @ApiResponse(responseCode = "400", description = "Invalid request body."),
    @ApiResponse(responseCode = "401", description = "Unauthorized access."),
    @ApiResponse(responseCode = "403", description = "Access forbidden.")
})
@PostMapping
public ResponseEntity<?> createCareerProfile(@RequestBody UserCareerProfileRequest requ
    // Career profile create karna ka logic yahan likhenge
}
```

Without these annotations: Agar ye annotations nahi diye to Swagger ya OpenAPI documentation me aapka API describe nahi hoga. Yeh annotations documentation generate karne mein madad karte hain, taaki frontend developer ko samajh aaye ki API kaise use karna hai aur expected responses kya hain.

8. @RequestBody

Spring Boot Explanation:

- Ye annotation HTTP request body ko Java object ke sath bind karne ke liye use hota hai. Jab aap POST request bhejte ho aur body me data send karte ho, tab Spring us data ko object me convert karta hai.

Express.js Comparison: Express.js mein aap `req.body` ka use karte ho, jo ki body data ko fetch karta hai.

Code Example:

```
@PostMapping
public ResponseEntity<?> createCareerProfile(@RequestBody UserCareerProfileRequest requ
    // Career profile create karna ka logic
}
```

Without this annotation: Agar `@RequestBody` annotation nahi use karoge, to Spring Boot request ke body ko object me convert nahi karega, aur aapko manually parsing karna padega, jo kaafi complex ho sakta hai.

9. @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor

Spring Boot Explanation:

- **@Getter**: Ye Lombok annotation automatically class ke fields ke liye getter methods generate karta hai.
- **@Setter**: Ye Lombok annotation automatically setter methods generate karta hai.
- **@NoArgsConstructor**: Ye constructor bina kisi argument ke generate karta hai.
- **@AllArgsConstructor**: Ye saare fields ke liye constructor generate karta hai.

Express.js Comparison: Express.js mein aapko manually getters, setters, aur constructors define karne padte hain.
Code Example:

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserCareerProfileRequest {
    private String userId;
    private String careerDetails;
}
```

Without these annotations: Agar aap ye annotations use nahi karte, to aapko manually getters, setters, aur constructors likhne padenge. Yeh code ko bohot verbose bana deta hai.

10. @Schema(description = "Current page number")

Spring Boot Explanation:

- Ye annotation Swagger/OpenAPI documentation me field ke description ko add karta hai. Isse aap model ki fields ko achhe se document kar sakte ho.

Express.js Comparison: Express.js mein aap comments ya Swagger annotations ka use karte ho field description dene ke liye.
Code Example:

```
public class PaginationRequest {
    @Schema(description = "Current page number")
    private int page;
}
```

Without this annotation: Agar aap ye annotation nahi denge, to Swagger/OpenAPI documentation me us field ka description nahi dikhayi dega, jo ki documentation ko samajhne mein problem create kar sakta hai.

11. @EntityListeners(AuditingEntityListener.class), @CreatedDate, @CreatedBy, @LastModifiedBy, @LastModifiedDate

Spring Boot Explanation:

- Ye annotations auditing ke liye hoti hain, jo ki entity ke creation aur modification time ko automatically track karti hain. Jaise hi koi entity create ya update hoti hai, yeh fields automatically populate ho jaati hain.

Express.js Comparison: Express.js mein aapko manually yeh fields set karne padte hain jab record create ya update hota hai.
Code Example:

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

    @CreatedBy
    private String createdBy;

    @LastModifiedBy
    private String lastModifiedBy;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;
}
```

Without these annotations: Agar aap ye annotations nahi use karte, to aapko manually created date, modified date, created by, aur modified by ko set karna padega har time jab entity create ya update hoti hai.

12. @ControllerAdvice, @ExceptionHandler(ConstraintViolation

Spring Boot Explanation:

- **@ControllerAdvice**: Ye global exception handling ke liye use hota hai. Aap isme ek centralized error handling mechanism define kar sakte ho.
- **@ExceptionHandler**: Ye specific exception ko handle karne ke liye use hota hai.

Express.js Comparison: Express.js mein aap middleware ka use karte ho error handling ke liye.
Code Example:

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<?> handleConstraintViolation(ConstraintViolationException ex)
        return ResponseEntity.badRequest().body(ex.getMessage());
    }
}
```

Without these annotations: Agar yeh annotations nahi use karoge, to aapko har controller mein exception handling manually likhni padegi, jo code ko repetitive aur error-prone bana sakta hai.

13. @Serial

Spring Boot Explanation:

- Ye annotation Java serialization ke liye hota hai, jo ki object ko byte stream me convert karta hai.

Express.js Comparison: Express.js mein aap JSON.stringify aur JSON.parse ka use karte ho serialization ke liye.
Code Example:

```
public class MyClass implements Serializable {
    @Serial
    private static final long serialVersionUID = 1L;
}
```

Without this annotation: Agar aap yeh annotation use nahi karte, to serialization ke liye aapko manual configuration karni padegi.

14. @Component

Spring Boot Explanation:

- **@Component**: Is annotation ka use kisi bhi class ko Spring container ke under ek bean banane ke liye hota hai, jisse Spring automatically detect aur register kar leta hai.

Express.js Comparison: Express.js mein aap manually modules ko `require` karte ho.
Code Example:

```
@Component
public class MyService {
    // Business logic yahan
}
```

Without this annotation: Agar yeh annotation nahi hota, to Spring ko ye class automatically detect nahi hoti, aur aapko manually isse register karna padega.

15. @ImportAutoConfiguration

Spring Boot Explanation:

- Ye annotation Spring Boot ko automatically configuration classes ko import karne ka permission deta hai.

Express.js Comparison: Express.js mein aapko manually middleware aur plugins configure karne padte hain.
Code Example:

```
@ImportAutoConfiguration({MyConfiguration.class})
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Without this annotation: Agar aap yeh annotation nahi use karte, to aapko manually configuration ko handle karna padega, jo development process ko slow kar sakta hai.
==================================

Spring Boot JPA Relationships Explained in Hinglish

Introduction

Spring Boot mein `@ManyToOne`, `@OneToMany`, `@OneToOne`, aur `@ManyToMany` annotations ka use **entity relationships** define karne ke liye hota hai. Agar tum naye ho toh main har ek cheez step by step samjhata hoon.

Hinglish Explanation

Spring Boot mein `JPA` aur `Hibernate` ka use karke hum tables (entities) ko relational database mein connect kar sakte hain. Is document mein hum 4 types ke relationships ko samjhenge.

Spring Boot JPA Relationships Basics

Spring Boot mein agar humein **do tables (entities) ko relational database mein connect** karna ho, toh hum **JPA (Java Persistence API) aur Hibernate** ka use karte hain.
JPA relationships ke 4 types hote hain:

Point To Note

1. **@OneToOne** → Ek table ka ek row doosri table ke ek row se connected hota hai.
2. **@OneToMany** → Ek table ka ek row doosri table ke **multiple rows** se connected hota hai.
3. **@ManyToOne** → Multiple rows ek hi table ke ek row se connected hote hain.
4. **@ManyToMany** → Dono tables ke multiple rows ek doosre se connected hote hain.

Hinglish Explanation

- `@OneToOne`: Ek student ka sirf ek address ho sakta hai.
- `@OneToMany`: Ek school ke bahut saare students ho sakte hain.
- `@ManyToOne`: Bahut saare students ek hi school mein ho sakte hain.
- `@ManyToMany`: Ek student multiple courses le sakta hai aur ek course multiple students ke liye ho sakta hai.

1. Understanding '@ManyToOne' (Many-to-One Relationship)

'@ManyToOne' ka matlab hai **bahut saare students ek hi school ke under aayenge.**

Example: School aur Student Relationship

Example

Java Code: School Entity

```java
import jakarta.persistence.*;

@Entity
public class School {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)  //
    ↪ Auto increment ID
    private Long id;
    private String name;
}
```

Example

Java Code: Student Entity

```java
import jakarta.persistence.*;

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)  //
    ↪ Auto increment ID
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "school_id")  // Foreign key column in
    ↪ Student table
    private School school;
}
```

Generated Database Tables

```sql
CREATE TABLE school (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255)
);

CREATE TABLE student (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    school_id BIGINT,
    FOREIGN KEY (school_id) REFERENCES school(id)
);
```

Hinglish Explanation

Yahan `@ManyToOne` ka matlab hai ki ek student ek school se belong karta hai. `school_id` foreign key banega `student` table mein.

Point To Note

Q: Agar '@ManyToOne' na likhein toh kya hoga?

Agar '@ManyToOne' **nahi likha**, toh 'school' field sirf ek normal variable ban jayega, aur **database mein foreign key nahi banegi.**

Example

Java Code: Without @ManyToOne

```java
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    private School school;  //     Foreign key nahi banegi
}
```

Hinglish Explanation

Is case mein **Hibernate ‘school_id‘ column create nahi karega**, aur relation ka data manually manage karna padega.

2. Understanding ‘@OneToMany‘ (One-to-Many Relationship)

‘@OneToMany‘ ka matlab hai **ek school ke multiple students honge.**

Example

Java Code: School Entity with @OneToMany

```java
import java.util.List;
import jakarta.persistence.*;

@Entity
public class School {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToMany(mappedBy = "school")  // Connect with Student
    ↪ entity
    private List<Student> students;
}
```

Point To Note

Hinglish Explanation

`mappedBy = "school"` ka matlab hai yeh field `Student` table ke `school` field se connected hai. Iska direct foreign key column nahi banta, kyunki ye ek reverse mapping hai.

3. Understanding ‘@OneToOne‘ (One-to-One Relationship)

Ek student ka sirf ek address ho sakta hai.

Example

Java Code: Address Entity

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String city;
}
```

Example

Java Code: Student Entity with @OneToOne

```java
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToOne
    @JoinColumn(name = "address_id")  // Foreign key
    private Address address;
}
```

Point To Note

Hinglish Explanation

`address_id` foreign key banega `student` table mein.

4. Understanding ‘@ManyToMany‘ (Many-to-Many Relationship)

Ek student multiple courses le sakta hai aur ek course multiple students ke liye ho sakta hai.

Example

Java Code: Student Entity with @ManyToMany

```java
import java.util.List;
import jakarta.persistence.*;

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}
```

Example

Java Code: Course Entity

```java
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String courseName;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

Generated SQL Table

```sql
CREATE TABLE student_course (
    student_id BIGINT,
    course_id BIGINT,
    FOREIGN KEY (student_id) REFERENCES student(id),
    FOREIGN KEY (course_id) REFERENCES course(id)
);
```

Hinglish Explanation

`student_course` naam ka ek extra table banega, jo `student_id` aur `course_id` ko connect karega.

Conclusion (Summary)

headingblue!30 **Annotation**	**Description**	**Foreign Key C ated?**
`@OneToOne`	Ek row doosri table ke ek row se connected	Yes
`@OneToMany`	Ek row doosri table ke multiple rows se connected	No
`@ManyToOne`	Multiple rows ek row se connected	Yes
`@ManyToMany`	Multiple rows ek doosre se connected	Yes

Hinglish Explanation

Yeh table batata hai ki konse relationship mein foreign key banegi aur konse mein nahi.

Additional Explanation for Beginners

Point To Note

- **@JoinColumn(name = "school_id")**: Yeh annotation batata hai ki `student` table mein ek column banega jiska naam `school_id` hoga aur yeh `school` table ke `id` column se connect hoga.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Yeh annotation batata hai ki `id` column auto-increment hogi.
- **@Entity**: Yeh annotation batata hai ki yeh class ek database table ko represent karti hai.
- **@Id**: Yeh annotation batata hai ki yeh field table ka primary key hai.
- **@OneToMany(mappedBy = "school")**: Yeh annotation batata hai ki `students` list `Student` entity ke `school` field se connected hai.

Real-Life Example

Imagine you are building a school management system:

- **School** aur **Student** ka `@ManyToOne` relationship hoga.
- **Student** aur **Address** ka `@OneToOne` relationship hoga.
- **Student** aur **Course** ka `@ManyToMany` relationship hoga.

Hinglish Explanation

Is tarah se, hum complex relationships ko easily manage kar sakte hain.

Final Note

Ab Spring Boot ka JPA relationship ka basic concept clear ho gaya? Koi aur doubt hai toh batao!

==================================

Understanding @JoinTable and @JoinColumn in Hinglish

Point To Note

@JoinTable Ka Matlab

`@JoinTable` annotation ka use `@ManyToMany` relationship mein kiya jata hai. Iska matlab hai ki ek extra table banega jo dono tables ke IDs ko connect karega.

Example: @JoinTable Ka Code

Example

Java Code: @JoinTable Example

```java
@ManyToMany
@JoinTable(
    name = "student_course",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id")
)
private List<Course> courses;
```

Hinglish Explanation

- `name = "student_course"`: Yeh batata hai ki extra table ka naam `student_course` hoga.
- `joinColumns = @JoinColumn(name = "student_id")`: Yeh batata hai ki `student_id` column `student` table se connect hoga.
- `inverseJoinColumns = @JoinColumn(name = "course_id")`: Yeh batata hai ki `course_id` column `course` table se connect hoga.

Generated SQL Table

```sql
CREATE TABLE student_course (
    student_id BIGINT,
    course_id BIGINT,
    FOREIGN KEY (student_id) REFERENCES student(id),
    FOREIGN KEY (course_id) REFERENCES course(id)
);
```

Table Representation

Column Name	Description
`student_id`	`student` table ke `id` se connected hai.
`course_id`	`course` table ke `id` se connected hai.

Hinglish Explanation

Yeh extra table (`student_course`) banega jo `student_id` aur `course_id` ko connect karega.

@JoinColumn Mein Default Behavior

Agar hum `@JoinColumn` mein `name` specify nahi karte, toh Hibernate automatically ek default column name generate karta hai.

Point To Note

Example: @JoinColumn Without Name

Example

Java Code: @JoinColumn Without Name

```java
@OneToOne
@JoinColumn // name specify nahi kiya gaya
private Address address;
```

Hinglish Explanation

- Agar `name` specify nahi kiya gaya, toh Hibernate default column name use karega.
- Default column name `address_id` hoga, kyunki field ka naam `address` hai aur `_id` append ho jayega.
- Matlab, agar field ka naam `address` hai, toh default column name `address_id` hoga.

Generated SQL Column

```sql
CREATE TABLE student (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    address_id BIGINT,  // Default column name
    FOREIGN KEY (address_id) REFERENCES address(id)
);
```

Table Representation

Column Name	Description
`id`	Primary key of the `student` table.
`name`	Name of the student.
`address_id`	Foreign key connecting to the `address` table.

Hinglish Explanation

Agar hum `name = "address_id"` specify nahi karte, toh Hibernate automatically `address_id` column banayega.

Summary

- `@JoinTable` ka use `@ManyToMany` relationship mein extra table banane ke liye hota hai.
- `@JoinColumn` mein agar `name` specify nahi kiya gaya, toh Hibernate default column name generate karega.
- Default column name field ke naam ke saath `_id` append karke banaya jata hai (e.g., `address_id`).

Hinglish Explanation

Is tarah se, hum `@JoinTable` aur `@JoinColumn` ka use karke complex relationships ko easily manage kar sakte hain. Agar default behavior samajhna hai, toh `name` specify nahi karna padega.

================================

33 Spring Boot Annotations

33.1 @SpringBootApplication

Overview

`@SpringBootApplication` Spring Boot application ka entry point hota hai. Isme teen annotations combine hote hain: `@Configuration`, `@EnableAutoConfiguration`, aur `@ComponentScan`.

Real-Life Example: Maano aap ek restaurant management system bana rahe hain. `@SpringBootApplication` wali class aapki restaurant ki main gate ki tarah hai, jahan se sab kuch start hota hai.

```
@SpringBootApplication
public class RestaurantApplication {
    public static void main(String[] args) {
        SpringApplication.run(RestaurantApplication.class, args);
    }
}
```

Explanation: `SpringApplication.run()` se aapki restaurant (application) start ho jati hai.

Point To Note

33.2 @RestController vs @Controller

Dono annotations controllers ke liye use hote hain, lekin inka use case alag hai.
Real-Life Example:

- `@Controller`: Maano aap ek website bana rahe hain jahan par aap users ko HTML pages dikhana chahte hain. Jaise ki restaurant ka menu page.

```
@Controller
public class MenuController {
    @GetMapping("/menu")
    public String showMenu() {
        return "menu"; // "menu.html" template render karega
    }
}
```

Point To Note

- `@RestController`: Maano aap ek mobile app ke liye API bana rahe hain jo JSON data return karega. Jaise ki restaurant ka menu data JSON format mein.

```
@RestController
public class MenuApiController {
    @GetMapping("/api/menu")
    public List<MenuItem> getMenu() {
        return menuService.getMenuItems(); // JSON response dega
    }
}
```

Point To Note

Agar aap web pages return kar rahe hain toh `@Controller` use karo, aur agar API bana rahe hain toh `@RestController` use karo.

33.3 @Service, @Repository, @Component

Ye annotations Spring Boot ke components ko identify karne ke liye use hote hain.

Real-Life Example:

- `@Service`: Maano aap restaurant mein waiter ka kaam karne wali service class bana rahe hain.

```java
@Service
public class WaiterService {
    public String takeOrder(String order) {
        return "Order taken: " + order;
    }
}
```

- `@Repository`: Maano aap restaurant ka database handle kar rahe hain, jaise ki orders ko store karna.

```java
@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {
    // Database queries yaha likhte hain
}
```

- `@Component`: Maano aap ek utility class bana rahe hain jo general kaam karegi, jaise ki bill generate karna.

```java
@Component
public class BillGenerator {
    public String generateBill() {
        return "Bill generated";
    }
}
```

Tip

Ye annotations Spring Boot ke "IoC Container" me automatically register ho jate hain.

34 Dependency Injection (DI) aur Inversion of Control (IoC)

Spring Boot me DI kaise kaam karta hai?

Dependency Injection (DI) ka matlab hai ki objects ka creation Spring Boot handle karega, aur hume manually `new` keyword se object banane ki zarurat nahi hoti.

Real-Life Example:

Maano aap restaurant mein waiter aur chef ko manage kar rahe hain. Waiter ko chef ki zarurat hai order complete karne ke liye. Spring Boot automatically chef ka object waiter ko provide karega.

```java
@Service
public class ChefService {
    public String cookFood(String order) {
        return "Cooking: " + order;
    }
}

@RestController
public class WaiterController {
    private final ChefService chefService;

    // Dependency Injection (Constructor-based)
    @Autowired
    public WaiterController(ChefService chefService) {
        this.chefService = chefService;
    }

    @GetMapping("/order")
    public String placeOrder(@RequestParam String order) {
        return chefService.cookFood(order);
    }
}
```

Spring Boot automatically ChefService ka object inject karega!

@Autowired ka deeper explanation

Agar tum `@Autowired` use karte ho toh Spring Boot automatically object inject kar deta hai.

```java
@Component
public class ManagerService {
    @Autowired
    private WaiterService waiterService;

    @Autowired
    private ChefService chefService;

    public String manageRestaurant() {
        return waiterService.takeOrder("Pizza") + " " + chefService.
    ↪ cookFood("Pizza");
    }
}
```

Best Practice: Constructor Injection use karo instead of Field Injection.

Manual DI kaise kar sakte hain (@Bean, @ComponentScan)?

Agar tum manually object create karna chahte ho toh `@Bean` use kar sakte ho.

```java
@Configuration
public class RestaurantConfig {
    @Bean
    public SpecialChef specialChef() {
        return new SpecialChef();
    }
}
```

Spring Boot automatically RestaurantConfig me defined beans ko use karega.

35 Spring Boot Security (Basic Authentication JWT)

Basic Authentication Setup

Spring Security ka default configuration username `"user"` aur randomly generated password deta hai.

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

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
    ↪ throws Exception {
        http.authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated())
            .formLogin();
        return http.build();
    }
}
```

Isse Spring Boot me default login form enable ho jayega.

JWT Authentication Example

JWT (JSON Web Token) authentication kaafi secure aur stateless hota hai.

```java
public String generateToken(String username) {
    return Jwts.builder()
        .setSubject(username)
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 *
    ↪ 60)) // 1 Hour
        .signWith(SignatureAlgorithm.HS256, "secret")
        .compact();
}
```

```java
public Claims validateToken(String token) {
    return Jwts.parser()
        .setSigningKey("secret")
        .parseClaimsJws(token)
        .getBody();
}
```

JWT authentication me token request headers me send hota hai.

36 Spring Boot Exception Handling

@ControllerAdvice ka use

```java
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(OrderNotFoundException.class)
    public ResponseEntity<String> handleOrderNotFoundException(
    ↪ OrderNotFoundException e) {
        return new ResponseEntity<>(e.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```

Yeh har ek exception ko globally handle karega.

37 Spring Boot Logging

```java
@Slf4j
@Service
public class OrderService {
    public void placeOrder(String order) {
        log.info("Order placed: " + order);
        log.error("Error placing order: " + order);
    }
}
```

Yeh logs console pe print honge aur application debugging me help karenge.

38 Deployment of Spring Boot Application

Jar file Generate karna

```
mvn clean package
```

Docker Pe Deploy Karna

```
FROM openjdk:17
COPY target/app.jar app.jar
CMD ["java", "-jar", "app.jar"]
```

Ab isko Kubernetes ya AWS pe deploy kar sakte ho!
article xcolor listings

Point To Note

39 Validation in Spring Boot

Spring Boot mein validation ke liye **@Valid, @NotNull, @Size, @Email** annotations ka use kiya jata hai. Ye annotations ensure karte hain ki input data correct format mein ho.

39.1 Real-Life Example

Maano aap ek restaurant ka user registration form bana rahe hain, jisme har user ka data validate karna hai.

```java
public class UserDTO {
    @NotNull
    private String name;

    @Email
    private String email;
}
```

Listing 1: User DTO with Validation

Point To Note

@NotNull: Ensure karta hai ki field **name** null na ho.
@Email: Ensure karta hai ki **email** valid email format mein ho.

39.2 Additional Validations

Agar aap aur validation lagana chahte hain to **@Size** ka use kar sakte hain:

```java
public class UserDTO {
    @NotNull
    @Size(min = 3, max = 50)
    private String name;

    @Email
    @NotNull
    private String email;
}
```

Listing 2: User DTO with More Validations

@Size(min = 3, max = 50): Ensure karega ki name kam se kam 3 aur maximum 50 characters ka ho.

39.3 Using @Valid in Controller

Agar aap Spring Boot Controller mein request validation karna chahte hain, to **@Valid** ka use kar sakte hain:

```java
@RestController
public class UserController {
    @PostMapping("/register")
    public ResponseEntity<String> registerUser(@Valid
    ↪ @RequestBody UserDTO user) {
        return ResponseEntity.ok("User registered successfully
    ↪ !");
    }
}
```

Listing 3: Using @Valid in a Controller

@Valid: Ensure karta hai ki incoming request ka data valid ho.

==================================

One-to-Many Relation in Spring Boot
Foreign Key Issue – Simple Explanation

40 Introduction

This document explains the One-to-Many relationship in Spring Boot and why foreign keys exist in the child table rather than the parent table.

41 Understanding One-to-Many Relationship

A **School** has multiple **Students**:

- **One School → Many Students**
- Each student belongs to one school, but a school has many students.

Example

ABC School has students: Rahul, Priya, Aman.

42 Foreign Key Placement Issue

Foreign keys are stored in the **child table** (Student), not the **parent table** (School).

```
Student Table:
+----+--------+----------+
| id | name   | schoolId |
+----+--------+----------+
| 1  | Rahul  | 101      |
| 2  | Priya  | 101      |
| 3  | Aman   | 102      |
+----+--------+----------+
```

Why not store students directly in the school table?

- Relational databases cannot store lists in a single column.
- **Foreign keys are always in the child table.**

43 Correct Approach in Spring Boot

Spring Boot handles this using **@OneToMany** and **@ManyToOne** annotations:

```java
@Entity
public class School {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "school", cascade = CascadeType.ALL)
    private List<Student> students;
}
```

```java
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "school_id")
    private School school;
}
```

Point To Note

- `@OneToMany(mappedBy = "school")` means **School has a list of students but does not store a foreign key.**
- `@ManyToOne @JoinColumn(name = "school`$_i d$”)*means*`Student table stores the foreign key.`

Point To Note

44 CascadeType.ALL Explanation

If **CascadeType.ALL** is used:

· Deleting a **School** deletes its **Students**.
· Updating a **School** updates its **Students**.

Example

Deleting School ID 101 will delete all students linked to it.

45 Summary

1. **Foreign key is always in the child table (Student), not the parent table (School).**
2. **Spring Boot uses @OneToMany and @ManyToOne to map relationships.**
3. **CascadeType.ALL helps manage related records automatically.**

==================================

Point To Note

Most Important Point Below all points are very very important and made from my mistakes so please go through each line by line....

Mistake: Spring Boot Controller Not Found Issue

Galti Jo Hui Thi:

1. **Package Structure Galat Tha:**
2. Maine ‘controller‘ aur ‘service‘ folders **‘java‘ folder ke andar** bana diye the.
3. Lekin **Spring Boot sirf ‘@SpringBootApplication‘ wale class ke parent package me scan karta hai**.
4. Mera ‘DemoApplication.java‘ **‘com.example.demo‘** package me tha, par ‘Controller.java‘ **‘com.example.demo.Controller‘** package me likha tha, jo Spring Boot ke scan range ke bahar tha.
5. **Class Name ‘Controller‘ Rakha Tha:**
6. ‘Controller‘ naam dena **galat tha** kyunki Spring ke ‘@Controller‘ annotation ke saath **naming conflict ho raha tha**.
7. Is wajah se Spring Boot ne ‘Controller.java‘ ko properly detect nahi kiya.

Correct Solution Jo Follow Karna Hai:

1. **Package Structure Sahi Karna:**
2. **Controller aur Service folders** ‘DemoApplication‘ ke **bagal me hone chahiye**.
3. **Sahi Structure:**

Correct Folder Structure

```
    src/main/java/com/example/demo/
            DemoApplication.java  <-- Spring Boot ka main
↪  class
            controller/           <-- Yahan controllers
↪ rakhna hai
                ApiController.java  (Rename from
↪ Controller.java)
            service/              <-- Yahan services
↪ rakhni hai
                ServiceClass.java

```

4. Ab ‘DemoApplication.java‘ **automatically ‘controller‘ aur ‘service‘ folder scan karega**.
5. **Class Name Meaningful Rakhna:**
6. ‘Controller.java‘ ka naam **rename karke** ‘ApiController.java‘ rakho taaki **naming conflict na ho**.

⋆ *KeyTakeaways* :

- **Spring Boot sirf ‘@SpringBootApplication‘ ke parent package me controllers aur services ko detect karega.**
- **Controller class ka naam generic (‘Controller.java‘) mat rakho, use something like ‘ApiController.java‘.**
- **Packages aur folders ka structure sahi hona chahiye nahi toh Spring Boot usko detect nahi karega.**

==============================

451em

45.01em

45.0.01em

Point To Note

46 Getter and Setter Methods in Java (Hinglish Explanation)

Getter aur Setter methods ka use **Encapsulation** ke concept ko implement karne ke liye hota hai. Yeh methods **private variables** ko access karne aur modify karne ka safe tareeka provide karte hain.

47 Getter and Setter Methods Kya Hote Hain?

Getter Methods → Private variables ki value **read** karne ke liye. **Setter Methods** → Private variables ki value **update** karne ke liye.

47.1 Example (Java)

```java
public class User {
    private String email; // Private variable

    // Getter Method
    public String getEmail() {
        return email;
    }

    // Setter Method
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Listing 4: Getter and Setter Methods in Java

Example

Getter vs Setter **Getter se sirf value read hoti hai aur Setter se value update hoti hai.**

48 Agar Getter and Setter Use Nahi Karenge to Kya Hoga?

Agar **getter aur setter** methods **nahi use** karoge to:

- **Private variables ko access nahi kar paoge** - Kyunki **private** variables **directly access nahi kiye ja sakte** kisi aur class se.
- **Encapsulation break ho jayegi** - Private data ko **directly modify** karna **unsafe hota hai**.
- **Data Validation nahi kar sakte - Setter method me validation dal sakte hain**, lekin direct access se validation nahi hoga.

48.1 Example Without Getter/Setter

```java
public class User {
    private String email; // Private variable

    public void showEmail() {
        System.out.println(email); // Direct access not possible from
    ↪ another class
    }
}
```

Listing 5: Without Getter/Setter Methods

Yahan `email` variable **private** hai, to isse **direct access nahi kar sakte**. Agar **getter method nahi hoga**, to iski value **bahar se read nahi kar paoge**.

49 Getter/Setter ka Express.js ke saath Comparison

Agar tum **Express.js** jaante ho, to isse **Mongoose Model ke concept** se relate kar sakte ho.

49.1 Express.js Mongoose Model (Getter/Setter jaisa)

```javascript
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
    email: { type: String, required: true }
});

// Getter function jaisa
userSchema.methods.getEmail = function() {
    return this.email;
};

// Setter function jaisa
```

```
userSchema.methods.setEmail = function(newEmail) {
    this.email = newEmail;
};

const User = mongoose.model('User', userSchema);

let user = new User({ email: "test@example.com" });

console.log(user.getEmail()); // Getter ka use
user.setEmail("newemail@example.com"); // Setter ka use
console.log(user.getEmail());
```

Listing 6: Getter and Setter in Express.js Mongoose Model

49.2 Express.js vs Java Getter/Setter

—c—c—c—

headingblue **Feature**	**Express.js (Mongoose)**	**Java (Spring Boot)**
tablerow **Encapsulation**	Schema me define hota hai	Private variables aur methods ka use hota hai
Getter Method	`user.getEmail()`	`user.getEmail()`
tablerow **Setter Method**	`user.setEmail("newemail")`	`user.setEmail("newemail")`
Validation	Schema validation (`required: true`)	Setter me validation likh sakte ho

50 Conclusion (Hinglish Me)

- Getter aur Setter **Encapsulation implement karne** ke liye use hote hain.
- **Getter value ko read karne** ke liye hota hai.
- **Setter value update karne** aur **validation apply karne** ke liye hota hai.
- **Agar getter aur setter nahi likhoge to private variables ko access nahi kar paoge.**
- **Express.js me yeh concept Mongoose ke methods aur schema validation ke through implement hota hai.**

Ab tum Getter aur Setter ko **Mongoose Model ke instance methods** se relate kar sakte ho!
================================

Point To Note

User Authentication in Spring Boot

Overview

In this section, we will implement user authentication using email and password in Spring Boot. We will create methods to:

- Find a user by email
- Check if the password matches
- Return true if authentication is successful, otherwise false

1. Service Layer - UserSignupService.java

This layer handles business logic, including finding users and verifying passwords.

Java Code

```java
//  src/main/java/com/example/demo/Service/UserSignupService.java
package com.example.demo.Service;

import com.example.demo.Entity.UserSignupEntity;
import com.example.demo.Repository.UserSignupRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.Optional;

@Service //  Service Layer Annotation
public class UserSignupService {

    @Autowired //  Spring Boot will inject the repository automatically
    private UserSignupRepository userSignupRepository;

    /**
     *  Method to authenticate user based on email & password
     * @param email - User’s email ID
     * @param password - User’s entered password
     * @return true if user exists and password matches, false otherwise
     */
    public boolean authenticateUser(String email, String password) {
        //  Step 1: Find user by email in the database
        Optional<UserSignupEntity> userOptional = userSignupRepository.findByEmail(email);

        //  Step 2: Check if user exists
        if (userOptional.isPresent()) { //  If user found in DB
            UserSignupEntity user = userOptional.get(); // Extract user object

            //  Step 3: Compare passwords (  Plain-text comparison, not recommended for production)
            return user.getPassword().equals(password); //  Return true if password matches
        }

        //  Step 4: Return false if user does not exist
        return false;
    }
}
```

Repository Layer - UserSignupRepository.java

This interface interacts with the database to find users by email.

Java Code

```java
// src/main/java/com/example/demo/Repository/UserSignupRepository.
    ↪ java
package com.example.demo.Repository;

import com.example.demo.Entity.UserSignupEntity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.Optional;

@Repository // Marks this interface as a Repository
public interface UserSignupRepository extends JpaRepository<
    ↪ UserSignupEntity, Long> {

    /**
     * Custom query method to find a user by email
     * @param email - User's email ID
     * @return Optional containing UserSignupEntity if found,
    ↪ otherwise empty
     */
    Optional<UserSignupEntity> findByEmail(String email);
    // Spring Data JPA will automatically generate SQL: SELECT * FROM
    ↪  users WHERE email = ?
}
```

Code Explanation in Detail

Service Layer (UserSignupService.java)

This handles authentication logic.

public boolean authenticateUser(String email, String password)

This method takes email and password as input and returns true if authentication is successful, otherwise false.

Optional¡UserSignupEntity¿ userOptional = userSignupRepository.findByEmail(email);

findByEmail(email) - Database se email ke basis par user dhoondhta hai.
Optional¡UserSignupEntity¿ - NullPointerException avoid karne ke liye Optional use hota hai.
Agar email exist nahi karta toh Optional.empty() return hoga.

if (userOptional.isPresent())

Check kar raha hai ki user database me mila ya nahi.
Agar mila toh true hoga aur andar ka code execute hoga.

UserSignupEntity user = userOptional.get();

userOptional.get() se actual user object retrieve ho raha hai.

return user.getPassword().equals(password);

User ka password match karta hai ya nahi check kar raha hai.

Security Alert

Plain-text password check safe nahi hota.
Production me password hashing (BCrypt) use karna chahiye.

return false;

Agar email exist nahi karta toh false return karega.

Repository Layer (UserSignupRepository.java)

JpaRepository¡UserSignupEntity, Long¿
UserSignupEntity - Database table ke corresponding entity class.
Long - Primary key type.

Optional¡UserSignupEntity¿ findByEmail(String email);

Spring Data JPA automatically SQL query generate kar dega:

Generated SQL Query

```
SELECT * FROM users WHERE email = ?;
```

Additional Notes

Optional keyword use NullPointerException avoid karne ke liye hota hai.
Agar email match karega toh Optional¡UserSignupEntity¿ return hoga, warna Optional.empty().

Final Summary

Step-by-Step Authentication Process

· **Step 1:** User enters email & password.
User input aata hai authentication ke liye.
· **Step 2:** Database me `findByEmail(email)` se check hota hai.
Repository email match karke user return karta hai.
· **Step 3:** Agar user exist karta hai toh password match hota hai.
Plain-text password compare hota hai (Secure nahi hai).
· **Step 4:** Agar password sahi hai toh `true` return hota hai.
Authentication successful.
· **Step 5:** Agar user nahi mila ya password galat hai toh `false` return hota hai.
Authentication fail ho jata hai.

Security Improvements (Best Practices)

- **Warning:** Abhi hum password ko **plain-text** me check kar rahe hain, jo **secure nahi hai**.
- **Password Hashing ka Use Karo:**
- **BCrypt** ya **Argon2** hashing algorithms use karo.

Example Code (Java)

```java
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
if (encoder.matches(password, user.getPassword())) { ... }
```

Additional Security Measures

- **Database me encrypted password store karo.**
- **Login attempt count limit set karo** (Brute-force attack se bachne ke liye).

Conclusion (Hinglish Me)

Spring Boot me authentication ke liye `findByEmail()` use hota hai.
Agar **email exist** karta hai aur **password match** hota hai toh authentication successful hota hai.
Optional class ka use **NullPointerException** handle karne ke liye kiya jata hai.
Security ke liye password hashing implement karna zaroori hai.

==================================

Point To Note

Spring Boot mai Bean Concepts Explained

February 12, 2025

51 Spring me Bean kya hota hai?

Bean ek object hota hai jo **Spring IoC (Inversion of Control) container** ke dwara manage kiya jata hai. Jab hum '@Bean' annotation ka use karte hain, to Spring ko batate hain ki us object ko create aur manage karna hai.

52 BCryptPasswordEncoder kya hai aur kyun zaroori hai?

BCryptPasswordEncoder ek hashing algorithm hai jo passwords ko securely store karne ke liye use hota hai. Ye passwords ko encrypt karta hai taaki agar koi database hack ho jaye to bhi passwords secure rahe.

53 BCryptPasswordEncoder ko Bean kyun banana padta hai?

53.1 Dependency Injection (DI) ke liye

Agar hum har jagah 'new BCryptPasswordEncoder()' likhenge, to har baar naya object banega. Isse memory aur performance issues ho sakte hain. **Bean** banane se Spring ek hi object ko manage karta hai aur usko jahan bhi zaroorat ho, wahan inject kar deta hai.

53.2 Memory Management

Agar hum har baar 'new BCryptPasswordEncoder()' likhenge, to har baar naya object banega. Isse memory zyada use hogi. **Bean** banane se ek hi object reuse hota hai, jisse memory bachti hai.

53.3 Loose Coupling aur Testability

Loose Coupling ka matlab hai ki hum directly object create nahi karte, balki Spring se usko inject karwate hain. Isse code flexible aur testable ban jata hai. **Unit Testing** mein hum mock objects use kar sakte hain, jo ki easy hota hai jab hum '@Autowired' ka use karte hain.

54 Bean kaise kaam karta hai?

54.1 Bean ke bina (Galat Tarika)

```java
public class UserService {
    private BCryptPasswordEncoder encoder = new
    ↪ BCryptPasswordEncoder(); // Galat Tarika
}
```

Problem: Har baar 'UserService' create hone par naya 'BCryptPasswordEncoder' object banega. **Solution:** Isko Bean bana ke use karo!

54.2 Bean ke saath (Sahi Tarika)

```java
@Configuration
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); //  Ek hi object Spring
    ↪  manage karega
    }
}
```

Ab 'UserService.java' mein isko **autowire** karenge:

```java
@Service
public class UserService {

    @Autowired
    private PasswordEncoder passwordEncoder; // Spring Bean inject
    ↪ karega

    public void registerUser(User user) {
        String hashedPassword = passwordEncoder.encode(user.
    ↪ getPassword());
        user.setPassword(hashedPassword);
        userRepository.save(user);
    }
}
```

Benefit: Spring khud hi ‘BCryptPasswordEncoder‘ ko inject kar dega jab bhi zaroorat hogi.

55 Summary (Hinglish Mein)

- **Spring mein ”Bean” ka matlab hai ki Spring khud us object ko manage karega.**
- **Agar har jagah ‘new BCryptPasswordEncoder()‘ likhoge, to har baar naya object banega, jo memory waste karega.**
- **Agar ‘@Bean‘ use karoge, to ek hi object Spring manage karega, aur hum bas ‘@Autowired‘ se use kar sakte hain.**
- **Loose coupling aur dependency injection se code maintainable aur testable ban jata hai.**

56 Agar hum Bean nahi banayenge to kya hoga?

- **Memory Waste:** Har baar naya object banega, jisse memory zyada use hogi.
- **Code Duplication:** Har jagah ‘new BCryptPasswordEncoder()‘ likhna padega, jo code ko messy banayega.
- **Testing Difficulties:** Unit testing mein mock objects use karna mushkil ho jayega.

57 Kuch aur doubts clear karne ke liye

- **Agar hum ‘@Bean‘ na use karein to kya hoga?** Agar hum ‘@Bean‘ na use karein, to humein har jagah manually ‘new BCryptPasswordEncoder()‘ likhna padega. Isse code maintain karna mushkil ho jayega aur memory bhi zyada use hogi.
- **Kya hum ‘BCryptPasswordEncoder‘ ko directly class mein use kar sakte hain?** Ha, kar sakte hain, lekin ye best practice nahi hai. Isse code tightly coupled ho jayega aur testing mein dikkat aayegi.
- **Kya hum ek se zyada ‘BCryptPasswordEncoder‘ Bean bana sakte hain?** Ha, bana sakte hain, lekin Spring ko batana padega ki konsa Bean use karna hai. Iske liye ‘@Qualifier‘ annotation ka use hota hai.
- **Kya ‘BCryptPasswordEncoder‘ ke alawa aur koi PasswordEncoder use kar sakte hain?** Ha, Spring Security mein aur bhi PasswordEncoder hote hain jaise ‘NoOpPasswordEncoder‘, ‘Pbkdf2PasswordEncoder‘, etc. Lekin ‘BCryptPasswordEncoder‘ sabse secure mana jata hai.

58 Final Thought

- ‘BCryptPasswordEncoder‘ ko ‘@Bean‘ banana ek best practice hai. Isse memory efficient, maintainable, aur testable code milta hai.
- Agar aur koi doubt ho to pooch sakte hain.

59 Recommended Location for ‘SecurityConfig.java‘

59.1 Best Practice (Create a Config Package)

Place it inside a **dedicated configuration package**:

```
src/main/java/com/example/demo/
          Controller/
          Entity/
          Repository/
          Service/
          Config/    Create this package for configurations
                 SecurityConfig.java   Place this file here

          DemoApplication.java  (Main Spring Boot Application)
```

59.2 Why Create a ‘Config/‘ Package?

- **Organized Structure:** Keeps configuration files separate from business logic.
- **Scalability:** You can add more configuration files later (like JWT, CORS, etc.).
- **Easy to Maintain:** If you have multiple security-related configurations, they stay in one place.

59.3 What If You Don’t Create a ‘Config/‘ Package?

If you don’t want to create a ‘Config/‘ package, you can place ‘SecurityConfig.java‘ in the same package as your ‘DemoApplication.java‘ (main class), because Spring Boot automatically scans the **same package and its sub-packages**.

59.4 Alternative Location

```
src/main/java/com/example/demo/
          Controller/
          Entity/
          Repository/
          Service/
          SecurityConfig.java   Place here if no config package

          DemoApplication.java   (Main Spring Boot Application)
```

This works fine, but a separate ‘Config/‘ package is recommended for better structure.

59.5 Final Answer (Best Practice)

- **Create ‘Config/‘ package**
- **Place ‘SecurityConfig.java‘ inside it**
- This keeps your **Spring Boot project clean, modular, and scalable.**

==================================

Point To Note

Step-by-Step Guide to Fix Missing Dependency Issues

Introduction

This document provides a step-by-step guide to resolve missing dependency issues in Maven or Gradle projects using IntelliJ IDEA. It includes detailed instructions, examples, and Hinglish explanations for better understanding.

Step-by-Step Guide

Step 1: Open ‘pom.xml‘ or ‘build.gradle‘ File

Step 2: Maven (‘pom.xml‘): Open the ‘pom.xml‘ file and add the missing dependency in the ‘¡dependencies¿‘ section.

Step 3: Gradle (‘build.gradle‘): Open the ‘build.gradle‘ file and add the missing dependency in the ‘dependencies‘ block.

Step 4: Hinglish Explanation: ‘pom.xml‘ ya ‘build.gradle‘ file ko open karo aur missing dependency ko add karo.

Point To Note

Step 5: Reload Maven/Gradle Project

Step 6: Maven: Right-click on ‘pom.xml‘ and select `Maven > Reload Project`.

Step 7: Gradle: Right-click on ‘build.gradle‘ and select `Reload Gradle Project`.

Step 8: Hinglish Explanation: Project ko reload karo taaki changes apply ho sakein.

Step 9: Force Update Dependencies

Step 10: Maven: Run the following command in the terminal:

```
mvn clean install
```

Step 11: Gradle: Run the following command in the terminal:

```
./gradlew build --refresh-dependencies
```

Step 12: Hinglish Explanation: Dependencies ko force update karo taaki sab kuch download ho jaye.

Step 13: Invalidate Cache and Restart IntelliJ

Step 14: Go to `File > Invalidate Caches / Restart`.

Step 15: Hinglish Explanation: IntelliJ ka cache clear karo aur restart karo.

Step 16: Verify Imports

Step 17: Manually import the missing class in your Java file.

Step 18: Hinglish Explanation: Java file mein missing class ko manually import karo.

Step 19: Restart Application

Step 20: Restart your Spring Boot application.

Step 21: Hinglish Explanation: Application ko restart karo aur check karo ki sab sahi se kaam kar raha hai.

Examples

Example

Example: Adding Spring Security Dependency

```
<!-- Maven Dependency for Spring Security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Step	Action	Hinglish Explanation
Step 1	Open ‘pom.xml‘ or ‘build.gradle‘	File ko open karo aur dependency add karo
Step 2	Reload Project	Project ko reload karo
Step 3	Force Update Dependencies	Dependencies ko force update karo
Step 4	Invalidate Cache	Cache clear karo aur IntelliJ restart karo
Step 5	Verify Imports	Missing class ko manually import karo
Step 6	Restart Application	Application restart karo

Table 2: Step-by-Step Summary

Tables

Summary

- **Check ‘pom.xml‘ or ‘build.gradle‘ for missing dependencies.**
- **Reload the project to apply changes.**
- **Force update dependencies if necessary.**
- **Invalidate cache and restart IntelliJ.**
- **Verify imports and restart the application.**

==============================

Point To Note

Java mein `import` kaise kaam karta hai? (Hinglish Guide)

60 Introduction

Agar aap Java ya Spring Boot seekh rahe ho, toh aapko `import` ka concept samajhna zaroori hai. Ye guide step-by-step aapko `import`, `package`, aur `class` ka basic samjhane wali hai.
—

61 Step 1: `package` kya hota hai?

article xcolor listings

Package ke andar multiple classes ka structure

Suppose you have a package named `com.example.demo`. Inside this package, you can have multiple classes, such as:

```
com/example/demo/
              Main.java
              User.java
              Product.java
              Utils.java
```

Here:

- `Main.java`, `User.java`, `Product.java`, and `Utils.java` are all part of the `com.example.demo` package.
- Each file represents a separate class.
- `package` **ek folder jaisa hota hai**, jo aapki Java classes ko organize karne ke liye use hota hai.
- Ye **namespace** provide karta hai jisse same naam wali classes ek doosre se alag rah sakein.

  Example: Agar aapka project ka structure kuch aisa hai:

```
com/example/demo/Security/JWTUtils.java
```

Toh `JWTUtils.java` `com.example.demo.Security` **package** ke andar hai.

61.1 Package define karna

Jab aap koi Java file (`.java`) likhte ho, toh sabse pehle `package` define karna hota hai:

Example

Package Definition

```java
package com.example.demo.Security; // Batata hai ki ye class
    ↪ kis package mein hai

public class JWTUtils {
    // Methods yaha honge
}
```

—

62 Step 2: `import` kya hota hai?

- **Jab aap ek class ko doosri class mein use karna chahte ho**, toh usko import karna padta hai.
- Import batata hai ki Java ko kahaan se class ya method uthani hai.

—

63 Step 3: Aapka `import` statement galat kyun hai?

Aap likh rahe ho:

```java
import com.example.demo.Security.JWTUtils.*;
```

Lekin ye **incorrect** hai kyunki:

- `JWTUtils` **ek class hai, package nahi** – Isliye `.*` lagana galat hai.
- `import com.example.demo.Security.JWTUtils;` likhna chahiye.

—

64 Step 4: Sahi Tarika: Import ek puri class

Agar `JWTUtils.java` aapke `com.example.demo.Security` folder ke andar hai, toh sahi import hoga:

Example

Correct Import Statement

```java
import com.example.demo.Security.JWTUtils;
```

Ab aap `JWTUtils` class ke methods ko is tarah se use kar sakte ho:

```java
String token = JWTUtils.generateToken("user@example.com");
```

—

65 Step 5: Static Import (Direct Method Use)

Agar aapko **class ka naam likhne se bachna hai**, toh aap **static import** ka use kar sakte ho.

65.1 Example: Static Import

Example

Static Import Example

```java
import static com.example.demo.Security.JWTUtils.generateToken
    ↪ ;
```

Ab aap **directly method use kar sakte ho** bina `JWTUtils.` likhe:

```java
String token = generateToken("user@example.com"); // Ab JWTUtils
    ↪ likhne ki zaroorat nahi
```

—

66 Step 6: Final Steps (Check Fix)

Agar still error aa raha hai, toh **ye cheezein check karo:**

- **Check karo ki** `JWTUtils.java` **sahi package** (`com.example.demo.Security`) **mein hai ya nahi.**
- **Galat import ko hatao aur sahi wala likho:**

```java
    import com.example.demo.Security.JWTUtils;

```

- **Agar method ko direct use karna hai toh** `static import` **ka use karo.**

— ==================================

661em

Point To Note

Complete Explanation of SecurityConfig in Spring Boot

67 Introduction

Spring Boot me **Spring Security** ka use **authentication aur authorization** handle karne ke liye hota hai. Ye ‘SecurityConfig‘ file ek middleware ki tarah kaam karti hai jo requests ko filter karti hai. Yeh ‘Interceptor‘ se different hai kyunki yeh request ko unauthorized hone pe reject bhi kar sakta hai.
Agar tum Express.js se relate karna chahte ho, to: **- Interceptor** ‘app.use(middleware)‘ **- Spring Security Middleware** ‘app.use(authMiddleware)‘
—

68 SecurityConfig File Ka Role

68.1 Spring Boot me Security Middleware (Interceptor se Different)

Feature	**Spring Security (‘SecurityConfig‘)**	**Interceptor (‘LoggingInterceptor‘)**
Purpose	Authentication & Authorization	Logging, Custom Rules
Request Control	Unauthorized request ko reject karta hai	Request ko log ya modify karta hai
Registration	‘SecurityFilterChain‘ in ‘SecurityConfig‘	‘InterceptorRegistry‘ in ‘WebConfig‘

—

69 SecurityConfig.java Code Explanation

src/main/java/com/example/demo/Config/SecurityConfig.java

Example

SecurityConfig.java

```java
package com.example.demo.Config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration  // Batata hai ki yeh ek configuration class hai
public class SecurityConfig {

    // Step 1: Password Encoding (For Storing Hashed Passwords)
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); // Passwords ko hash karne ke liye BCrypt use kar rahe hain
    }

    // Step 2: Security Filter Chain (Authentication & Authorization)
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .csrf(csrf -> csrf.disable())  // CSRF protection disable (sirf testing ke liye)
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/users/login", "/users/signup", "/users/test").permitAll()  // Public APIs
                        .anyRequest().authenticated() // Baki sab requests secure hongi
                )
                .httpBasic(withDefaults())  // Basic Authentication enable
                .formLogin(form -> form.disable()); // Default Spring Boot Login Form ko disable kar rahe hain

        return http.build();
    }
}
```

—

70 SecurityConfig Ka Step-by-Step Breakdown

70.1 Step 1: ‘@Configuration‘ (Spring Boot ko batana ki yeh ek config file hai)

```java
@Configuration
public class SecurityConfig {
```

Isse Spring Boot automatically is class ko load karega.
—

70.2 Step 2: Password Encoder (‘BCryptPasswordEncoder‘)

```java
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Yeh ‘passwordEncoder()‘ function passwords ko hash karne ke liye use hota hai. Agar tum user password database me store kar rahe ho, to yeh plain text ke bajaye encrypted format me store karega.
—

70.3 Step 3: Security Rules (Middleware for Authentication)

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
    ↪ throws Exception {
```

Yeh function ‘SecurityFilterChain‘ ko return karta hai, jo ek middleware jaisa kaam karta hai. Yeh har request ko check karega ki yeh public hai ya authentication required hai.
—

70.4 Step 4: CSRF Protection Disable (Testing Mode)

```java
.csrf(csrf -> csrf.disable())
```

CSRF (Cross-Site Request Forgery) security ko disable kar rahe hain. Production me isko enable rakhna zaroori hai.
—

70.5 Step 5: Public Routes (Permit Some APIs)

```java
.authorizeHttpRequests(auth -> auth
        .requestMatchers("/users/login", "/users/signup", "/users/
    ↪ test").permitAll()
```

Yeh batata hai ki ‘/users/login‘, ‘/users/signup‘, aur ‘/users/test‘ routes sabko access mil sakta hai. Iska matlab yeh APIs bina login ke bhi accessible hongi.
—

70.6 Step 6: Secure Other Routes

```java
.anyRequest().authenticated()
```

Iska matlab hai ki jo routes explicitly allow nahi kiye gaye, wo sabhi authentication ke bina access nahi kiye ja sakte. Agar koi unauthenticated user ‘/users/profile‘ API call karega, to ‘401 Unauthorized‘ error milega.
—

70.7 Step 7: Enable Basic Authentication

```
.httpBasic(withDefaults())
```

Yeh Basic Authentication enable karta hai. Agar tum JWT ya OAuth use kar rahe ho, to isko disable karke custom authentication use kar sakte ho.
—

70.8 Step 8: Disable Default Spring Security Login Form

```
.formLogin(form -> form.disable());
```

By default, Spring Security ek login form dikhata hai. Isko disable kar diya gaya hai kyunki hum sirf API authentication chahte hain.
—

71 SecurityConfig Ko Register Kaise Kiya Gaya Hai?

Spring Boot me '@Configuration' aur '@Bean' use karke automatic registration hoti hai. Agar tum Spring Boot me 'SecurityConfig' likh dete ho, to Spring Boot khud isko middleware jaisa treat karega. Alag se 'WebConfig' me register karne ki zaroorat nahi hai.
—

72 SecurityConfig Test Karne Ke Liye Postman Requests

72.1 Public APIs (Allowed Without Login)

```
GET http://localhost:8080/users/test
```

Response:

```
{
  "message": "This is a public API"
}
```

Ye request bina authentication ke allow hogi.
—

72.2 Secure API (Unauthorized Without Login)

```
GET http://localhost:8080/users/profile
```

Response:

```
{
  "error": "Unauthorized"
}
```

Ye request '401 Unauthorized' dega, kyunki authentication required hai.
—

72.3 Secure API (With Authentication)

```
GET http://localhost:8080/users/profile
Authorization: Basic <base64-encoded-username:password>
```

Agar tum username/password sahi bhejoge to request authorized ho jayegi.
Agar tum JWT use karna chahte ho, to ‘httpBasic(withDefaults())‘ hata ke JWT middleware use kar sakte ho.
—

73 Final Summary

Feature	Function	Role
‘@Configuration‘	SecurityConfig class ko Spring Boot me register karta hai	Middlewa hai
‘PasswordEncoder‘	‘BCryptPasswordEncoder‘	Password ke liye
‘SecurityFilterChain‘	‘http.authorizeHttpRequests()‘	Security hai
‘csrf.disable()‘	CSRF protection disable karta hai	Sirf testin
‘.permitAll()‘	Public APIs allow karta hai	‘/users/lo s/signup‘ tion ke hai
‘.anyRequest().authenticated()‘	Baki sabhi routes secure karta hai	Agar log ‘401 Una
‘.httpBasic(withDefaults())‘	Basic Authentication enable karta hai	Default word se lo
‘.formLogin().disable()‘	Default login form disable karta hai	Sirf API allow kar

—

74 Conclusion

Spring Boot me ‘SecurityConfig‘ ek middleware ki tarah kaam karta hai jo requests ko authenticate karta hai. Agar kisi request ko authentication ki zaroorat hai aur user login nahi hai, to ‘401 Unauthorized‘ error aayega. Agar tum JWT use karna chahte ho to ‘httpBasic()‘ hata kar JWT middleware use kar sakte ho.
Ab tum ‘SecurityConfig‘ ko easily implement kar sakte ho aur APIs ko secure kar sakte ho!
—
================================

Point To Note

Rules for Class Names and Filenames in Java (Spring Boot

Importance of Matching Class Name and Filename

Spring Boot (aur Java me generally) me **class ka naam filename se match hona chahiye**. Agar aisa nahi hota, to aapko **compilation error** milega.

Rules for Class Names and Filenames

Point To Note

1. Class Name = File Name

Agar aapki class ka naam `UserSignupEntity` hai, to file ka naam bhi **UserSignupEntity.java** hona chahiye.

Example:

```java
public class UserSignupEntity {
    // class content
}
```

Filename: `UserSignupEntity.java`

2. Case-Sensitive

Java **case-sensitive** hota hai, to `UserSignupEntity.java` aur `usersignupentity.java` alag files maani jayengi.

3. Public Classes Must Match File Name

Agar class `public` hai, to filename aur class name exactly match karna chahiye.
Wrong Example:

- File: `User.java`
- Class inside:

```java
public class UserSignupEntity { }  //      Will cause compilation error
```

4. Package Naming

Agar class kisi package me hai `package com.example.demo.entity;`, to:

- **File Path:** `src/main/java/com/example/demo/entity/UserSignupEntity.java`

What Happens if Class Name and File Name Don't Match?

Aapko **compilation error** milega:

```
Error:  The public type UserSignupEntity must be defined in its own
file
```

==================================

Point To Note

Spring Boot: Understanding `findByEmail()` and Objects in Simple Terms

1. What is an Object? (Simple Explanation)

Definition:

An **object** is a **real-world instance of a class**. A **class** is like a **blueprint**, and an **object** is an **actual usable item** made from it.

Example: `UserSignupEntity` Class (Blueprint)

```java
public class UserSignupEntity {
    private Long id;
    private String email;
    private String password;
    private String role;

    // Constructor
    public UserSignupEntity(Long id, String email, String password, String role) {
        this.id = id;
        this.email = email;
        this.password = password;
        this.role = role;
    }

    // Getter Method
    public String getEmail() {
        return email;
    }
}
```

This class defines a user, but it's just a structure (blueprint).

Example: Creating Objects from Class

```java
UserSignupEntity user1 = new UserSignupEntity(1L, "satyam@gmail.com", "password123", "USER");
UserSignupEntity user2 = new UserSignupEntity(2L, "john@gmail.com", "johnpass", "ADMIN");
```

Think of a class as a Car Blueprint, and objects as real Honda City, BMW, etc.

2. Understanding `Optional<UserSignupEntity> findByEmail(String email);`

What is This?

```java
Optional<UserSignupEntity> findByEmail(String email);
```

This method helps fetch a user from the database using their email.

Breaking It Down

Keyword	Meaning
`Optional<>`	**Wrapper that may or may not contain a value**
`UserSignupEntity`	**Type of object inside Optional** (user details)
`findByEmail(String email)`	**Method name (Spring Boot converts it into SQL query)**
`String email`	**Input parameter (email we are searching for)**

Example Query That Spring Generates

```sql
SELECT * FROM users WHERE email = 'satyam@gmail.com' LIMIT 1;
```

No need to write SQL manually, Spring JPA does it for us!

3. Where to Define `findByEmail()`?

Correct Repository Code

```java
public interface UserSignupRepository extends JpaRepository<
    ↪ UserSignupEntity, Long> {
    Optional<UserSignupEntity> findByEmail(String email); //     Auto-
    ↪ generates SQL query
}
```

4. How to Use `findByEmail()` in a Service Class?

Correct Usage in Service Class

```java
public UserSignupEntity findByEmail(String email) {
    Optional<UserSignupEntity> userOptional = userSignupRepository.
    ↪ findByEmail(email);
    return userOptional.orElse(null); // If user is not found, return null
}
```

Better Approach: Throw Exception Instead of Returning Null

```java
public UserSignupEntity findByEmail(String email) {
    return userSignupRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found with
    ↪ email: " + email));
}
```

5. Why Do We Define `findByEmail()` in Repository?

Reason	Explanation
Auto Query Generation	Spring JPA **automatically creates SQL**.
Less Code Needed	No need to write complex SQL queries.
Direct Database Access	`findByEmail()` directly calls the DB.
Prevents Errors	Using `Optional` avoids `NullPointerException`.

Final Summary

- **An object** is a real-world instance of a class.
- `findByEmail(String email)` is a method that automatically generates an SQL query.
- We define it inside `UserSignupRepository` to let Spring Boot handle JPA queries.

Now you fully understand Objects, Repositories, and Optional!

==================================

Point To Note

Spring Boot: Understanding ‘findByEmail()‘ & Other Repository Methods

75 Introduction

Ye guide **Objects, Optional, Repository, aur Query Methods** ko Spring Boot me samjhane me madad karegi. Chaliye step-by-step samjhte hain.
—

76 Pre-Built Methods from ‘JpaRepository‘ (No Need to Define)

Spring Boot me ‘JpaRepository‘ **kuch important CRUD methods automatically provide karta hai**, jinhe hume manually likhne ki zaroorat nahi hoti. Agar hum ‘UserSignupRepository‘ ko ‘JpaRepository‘ se extend karte hain, to ye methods automatically available ho jate hain:

Example

Pre-Built Methods

```
userSignupRepository.findById(1L);  //     Find by ID
userSignupRepository.findAll();     //     Get all users
userSignupRepository.save(user);    //     Save a new user
userSignupRepository.deleteById(1L); //     Delete user by ID
```

In methods ko likhne ki koi zaroorat nahi hoti, ye built-in hote hain!
—

77 Custom Query Methods (Auto-Generated by Spring Boot)

‘findByEmail()‘ ke alawa, aap aur bhi **custom query methods** define kar sakte ho **Spring JPA ke naming conventions** ka use karke. Spring Boot **automatically SQL query generate karega**.

77.1 Custom Methods Ka Example

Example

Custom Query Methods

```
public interface UserSignupRepository extends JpaRepository<
    ↪ UserSignupEntity, Long> {

    //     Find user by email
    Optional<UserSignupEntity> findByEmail(String email);

    //     Find users by role
    List<UserSignupEntity> findByRole(String role);

    //     Find users who signed up after a certain date
    List<UserSignupEntity> findByCreatedAtAfter(Timestamp timestamp);

    //     Find active users
    List<UserSignupEntity> findByIsActiveTrue();

    //     Find users whose email contains a specific string (like
    ↪ search)
    List<UserSignupEntity> findByEmailContaining(String keyword);
}
```

Isme hume SQL likhne ki koi zaroorat nahi hai, Spring Boot khud query generate karega!

—

78 Custom Queries with ‘@Query‘ (Jab Naming Convention Kaafi Nahi Hota)

Kabhi kabhi complex queries likhni hoti hain jo **naming convention se possible nahi hoti**. Aise cases me hum ‘@Query‘ annotation ka use kar sakte hain.

78.1 Example: Get Users with Role = ”ADMIN”

Example

Custom Query Example

```
@Query("SELECT u FROM UserSignupEntity u WHERE u.role = 'ADMIN'")
List<UserSignupEntity> findAdmins();
```

78.1.1 When Will This Query Execute?

Jab bhi ‘findAdmins()‘ function ko call kiya jayega, Spring Boot **automatically is query ko execute karega**. Ye function database se **sabhi users jinka role ’ADMIN’ hai unko fetch karega**. Iska use tab hoga jab hume sirf administrators ki list chahiye.

78.1.2 Dynamic Role Fetching (Passing Variable Instead of ’ADMIN’)

Agar hume hardcoded ’ADMIN’ ke jagah kisi bhi role ka data fetch karna hai, toh hum ‘@Query‘ me variable pass kar sakte hain:

Example

Dynamic Role Query

```
@Query("SELECT u FROM UserSignupEntity u WHERE u.role = :role")
List<UserSignupEntity> findUsersByRole(@Param("role") String role);
```

Explanation: - ‘:role‘ → Yeh **named parameter** hai jo query ke andar **dynamically replace** hoga. - ‘@Param(”role”)‘ → Iska kaam hai method ke argument ‘role‘ ko SQL query ke ‘:role‘ placeholder se link karna.

How This Query Gets Executed? Jab bhi ‘findUsersByRole(”ADMIN”)‘ call hoga, Spring Boot **automatically executes** the following SQL query:

```
SELECT * FROM users WHERE role = 'ADMIN';
```

Agar hum ‘findUsersByRole(”USER”)‘ call karte hain, to SQL query kuch aise banegi:

```
SELECT * FROM users WHERE role = 'USER';
```

Usage in Service Layer: Repository method ‘findUsersByRole()‘ ko service class me aise call kiya jata hai:

Example

Calling the Method in Service Class

```java
List<UserSignupEntity> admins = userSignupRepository.findUsersByRole("ADMIN"); // Fetch Admins
List<UserSignupEntity> users = userSignupRepository.findUsersByRole("USER");  // Fetch Normal Users
```

Key Takeaways: - '@Query' ka use tab hota hai jab built-in naming conventions kaafi nahi hote. - '@Param' annotation se hum dynamic values SQL query me inject kar sakte hain. - 'findUsersByRole("ADMIN")' call hone par 'ADMIN' role ke users fetch honge. - 'findUsersByRole("USER")' call hone par 'USER' role ke users fetch honge.
Ab aap kisi bhi role ka data dynamically fetch kar sakte hain!

Ye queries database me directly SQL run karne ke liye kaam aati hain.

—

79 Using '@Modifying' for Update & Delete Queries

Agar aap **koi record update ya delete karna chahte ho bina entity ko pehle fetch kiye**, to '@Modifying' use kar sakte ho.

79.1 Example: Soft Delete a User (Mark as Inactive)

Example

@Modifying Example

```java
@Modifying
@Query("UPDATE UserSignupEntity u SET u.isActive = false WHERE u.email = :email")
void deactivateUser(@Param("email") String email);
```

Iska fayda ye hai ki poori entity ko fetch kiye bina hi 'isActive' field update ho jayegi, jo performance ke liye accha hai.

—

80 Summary: Repository Me Kya Kya Methods Use Kar Sakte Hain?

Method Type	**Example**	**Kya Manually Likhnana Zaroori Hai?**
Basic CRUD Methods	'findById(Long id)', 'findAll()'	No (Built-in)
Find by Field Name	'findByEmail(String email)', 'findByRole(String role)'	Yes (Spring generates query)
Comparison Queries	'findByCreatedAtAfter(Timestamp t)'	Yes
Boolean Queries	'findByIsActiveTrue()'	Yes
Search Queries	'findByEmailContaining(String keyword)'	Yes
Custom SQL Queries	'@Query("SELECT u FROM UserSignupEntity u WHERE u.role = 'ADMIN'")'	Yes
Update Queries	'@Modifying @Query("UPDATE UserSignupEntity u SET u.isActive = false WHERE u.email = :email")'	Yes

—

81 Final Answer: Kya Sirf ‘findByEmail()‘ Use Kar Sakte Hain?

Nahi! Aap bohot saare custom methods use kar sakte ho repository me. Spring Boot method naming conventions ka use karke queries automatically generate kar sakta hai. Agar complex query likhni ho to ‘@Query‘ ka use karo. Agar kisi record ko update/delete karna ho bina fetch kiye, to ‘@Modifying‘ ka use karo.
Ab aap apni repository me aur bhi powerful queries likh sakte ho! Agar aur doubts hain to batao!
—

82 Notes for You

- **Objects:** Java me objects real-world entities ko represent karte hain. Example: ‘UserSignupEntity‘ ek object hai jo database ke table ko represent karta hai.
- **Optional:** Ye ek container object hai jo null values ko handle karta hai. Example: ‘Optional¡UserSignupEntity¿‘ ka matlab hai ki result null ho sakta hai.
- **Repository:** Ye ek interface hai jo database operations ko handle karta hai. Example: ‘UserSignupRepository‘.
- **Query Methods:** Ye methods database se data fetch karne ke liye use hote hain. Example: ‘findByEmail()‘, ‘findByRole()‘.

—

83 Tips for Better Understanding

- **Practice Karo:** Khud se custom methods likhkar dekho aur unhe test karo.
- **Debugging:** Agar query sahi se kaam nahi kar rahi hai, to logs check karo aur SQL query ko database me manually run karke dekho.
- **Documentation Padho:** Spring Data JPA ki official documentation padhne se aur samjh aayegi.

—

84 Frequently Asked Questions (FAQs)

- **Q: Kya hum ek se zyada fields ke basis par query likh sakte hain? A:** Haan! Example: ‘findByEmailAndRole(String email, String role)‘.
- **Q: Agar hume koi custom SQL query likhni hai to kya karein? A:** ‘@Query‘ annotation ka use karo aur query likh do. Example:

```
    @Query("SELECT u FROM UserSignupEntity u WHERE u.email = :email AND u.role = :role")
    Optional<UserSignupEntity> findByEmailAndRole(@Param("email") String email, @Param("role") String role);

```

- **Q: ‘@Modifying‘ ka use kyu karte hain? A:** ‘@Modifying‘ ka use tab karte hain jab hum database me koi update ya delete operation karna chahte hain bina entity ko fetch kiye.

—

85 Conclusion

Spring Boot me 'findByEmail()' jaise methods ka use karke aap easily database se data fetch kar sakte hain. Iske alawa, aap custom methods aur complex queries bhi likh sakte hain. Bas thoda practice karo aur concepts ko samjho, aap khud hi expert ban jaoge! Agar aur kuch samajhna hai ya koi doubt hai to pooch sakte ho! ==================================

Point To Note

Spring Boot: Understanding @ManyToOne, and Foreign key

1 Understanding @ManyToOne and @JoinColumn in Todo Entity

In `Todo` entity, we want to store the user who created the To-Do. Since **one user can have multiple To-Dos**, we define a **Many-To-One relationship** where multiple To-Dos belong to a single user.

Code: @ManyToOne Relationship in Todo Entity

```
@ManyToOne //      Relationship: Many Todos belong to one User
@JoinColumn(name = "user_id", nullable = false) // Foreign Key (
    ↪ Links to User table)
private UserSignupEntity user;
```

Line-by-Line Explanation (Hinglish Style)

1 @ManyToOne - Ek User Ke Multiple To-Dos Ho Sakte Hain

```
@ManyToOne //      Relationship: Many Todos belong to one User
```

Iska Matlab:

- **Ek user ke multiple To-Dos ho sakte hain.**
- **Ek To-Do sirf ek user ka hoga.**
- **Database me, `user_id` foreign key hoga jo `users` table se link karega.**

Example Data in Tables:

—c—c—

User Table (users)	Todo Table (todo)
id: 1 (satyam@gmail.com)	id: 101, user_id: 1, title: "Learn Spring Boot"
id: 1 (satyam@gmail.com)	id: 102, user_id: 1, title: "Build JWT Auth"
id: 2 (john@gmail.com)	id: 103, user_id: 2, title: "Complete Project"

Dekh sakte ho, ek user (id: 1) ke multiple To-Dos ho sakte hain!

2 @JoinColumn(name = "user_id") - Foreign Key Define Karta Hai

```
@JoinColumn(name = "user_id", nullable = false) // Foreign Key (
    ↪ Links to User table)
```

Iska Matlab:

- **`user_id` ek foreign key hoga** jo `users` table ke `id` column se linked rahega.

- `nullable = false` ka matlab hai ki **har To-Do kisi ek user ke under aana zaroori hai**.

Database Table Structure (`todo` Table in SQL):

```sql
CREATE TABLE todo (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    user_id BIGINT NOT NULL, -- Foreign Key
    title VARCHAR(255) NOT NULL,
    description TEXT,
    completed BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) --     Linking to
    ↪ Users table
);
```

Dekh sakte ho ki `user_id` users table ke `id` se connected hai!

3 `UserSignupEntity` Kya Hai?

`UserSignupEntity` ek **model (entity)** hai jo `users` table ko represent karta hai. Is entity ke andar **id, email, password, role, aur createdAt fields hoti hain.**

`UserSignupEntity` Example:

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

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

    @Column(unique = true, nullable = false)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private String role;

    @Column(name = "created_at", updatable = false)
    private Timestamp createdAt;
}
```

Is entity ka `id` field `user_id` ke saath `todo` table me foreign key ke roop me use hoga.

Summary (Hinglish Me)

—c—p10cm—

Concept	**Explanation**
`@ManyToOne`	Ek User ke multiple To-Dos ho sakte hain, lekin ek To-Do sirf ek user ka hoga.
`@JoinColumn(name = "user_id")`	`user_id` field `users` table ke `id` column se link karega (Foreign Key).
`UserSignupEntity`	Ye `users` table ka model hai jo `id, email, password, role, createdAt` fields ko represent karta hai.

==================================

Point To Note

Understanding ‘@RequestAttribute‘ in Spring

86 Introduction

‘@RequestAttribute‘ is a Spring annotation that allows you to access data added as an attribute to the `HttpServletRequest`. This is typically used when an interceptor or middleware adds attributes to the request, and you need to retrieve them in a controller.

87 Hinglish Notes on ‘@RequestAttribute‘

87.1 ‘@RequestAttribute‘ Kya Hai?

‘@RequestAttribute‘ ek Spring annotation hai jo **HTTP request attributes** ko controller me access karne ke liye use hota hai. Yeh un data ko fetch karne ke liye kaam aata hai jo middleware (jaise ki Interceptor) ne request me add kiye hain.

- **Source:** Interceptor, middleware, ya kisi aur layer ne request me attribute add kiya hoga.
- **Purpose:** Request ke andar ke attributes ko controller tak leke jaana.

88 Example in Hinglish

88.1 Interceptor Me Attribute Add Karna

Interceptor ke `preHandle()` method me request attribute set karte hain:

Java Code Example: Interceptor

```java
@Component
public class LoggingInterceptor implements HandlerInterceptor
    ↪ {

    @Override
    public boolean preHandle(HttpServletRequest request,
    ↪ HttpServletResponse response, Object handler) {
        // User ID attribute add karte hain
        request.setAttribute("userId", 101);
        return true; // Proceed with the request
    }
}
```

88.2 Controller Me Attribute Access Karna

Controller me ‘@RequestAttribute‘ annotation ke through yeh attribute access karte hain:

Java Code Example: Controller

```java
@RestController
@RequestMapping("/todo")
public class TodoController {

    @GetMapping("/test")
    public String testAttribute(@RequestAttribute("userId")
    ↪ Integer userId) {
        return "Logged-in User ID: " + userId;
    }
}
```

89 Flow

1. **Interceptor:** ‘userId‘ ko `request.setAttribute("userId", value)` ke through request me set karta hai.
2. **Controller:** ‘@RequestAttribute(”userId”)‘ ke zariye us value ko fetch karta hai.

90 Output

Agar request `/todo/test` API par jati hai, to response me ye milega:

Output

```
Logged-in User ID: 101
```

91 Key Points

- **‘@RequestAttribute‘** un attributes ko fetch karta hai jo middleware ya interceptor me set hote hain.
- Yeh **request-scoped** hota hai (sirf current request ke liye valid).
- Common use cases: JWT token validation, user roles, ya custom headers process karna.

Aise explain karo to ye samajhne aur likhne ke liye easy hoga. Let me know if you'd like further help!
=================================

Point To Note

Hinglish Notes: `ResponseEntity<?>` Simplified with Full Explanation

Problem Without `ResponseEntity<?>`

Pehle yeh samjho ki `ResponseEntity<?>` kyun zaroori hai. Agar hum `ResponseEntity<?>` ka use nahi karte, to:

1. **Default Status Code Problem:**
2. Controller directly object ya string return karega, to Spring hamesha `200 OK` status code bhejta hai.
3. Chahe data na mile ya koi error ho, response ka status `200 OK` hi hoga, jo galat indication deta hai.

4. **Error Handling Mushkil Hota Hai:**
5. Agar koi error aayi, to hum proper status code (`404`, `500`) aur error message bhej nahi sakte.
6. **No Control Over Response:**
7. Hum response me headers (e.g., metadata) add nahi kar sakte, aur body ko customize karna mushkil hota hai.

Example Without `ResponseEntity<?>`

```java
@GetMapping("/todo/{id}")
public TodoEntity getTodoById(@PathVariable Long id) {
    TodoEntity todo = todoService.findById(id);

    if (todo == null) {
        return null; // Default response
    }

    return todo; // Default success response
}
```

Problems:

· Agar `todo` na mile, to `null` return hoga, aur status code `200 OK` hi rahega.

· Client ko pata hi nahi chalega ki data kyun nahi mila ya kya problem hai.

Solution: Using `ResponseEntity<?>`

`ResponseEntity<?>` ka use karke hum response ko **fully customize** kar sakte hain:

1. **Custom Status Codes:** Success ke liye `200 OK`, data na mile to `404 Not Found`, aur error ke liye `500 Internal Server Error`.
2. **Error Message:** Hum ek meaningful error message bhej sakte hain jo client ko help kare.
3. **Control Over Headers and Body:** Response headers aur body ko modify karna possible hota hai.

What Does `<?>` in `ResponseEntity<?>` Mean?

1. **`<?>` Kya Hai?**
2. Yeh ek **wildcard generic** hai, jo batata hai ki `ResponseEntity` ka response body kisi bhi type ka ho sakta hai.
3. **Flexibility Ka Example:**
4. Response body `String`, `Object`, ya `List` kuch bhi ho sakti hai.
5. Example:
6. `ResponseEntity<String>`: Sirf ek string bhejni ho.
7. `ResponseEntity<TodoEntity>`: Ek single `TodoEntity` object bhejna ho.
8. `ResponseEntity<List<TodoEntity>>`: Todo objects ki list bhejni ho.
9. **Agar `<?>` Use Karein:**

```java
ResponseEntity<?> response;
response = ResponseEntity.ok("Hello World"); // String type
response = ResponseEntity.ok(todo); // Object type

```

Example With `ResponseEntity<?>`

```java
@GetMapping("/todo/{id}")
public ResponseEntity<?> getTodoById(@PathVariable Long id) {
    TodoEntity todo = todoService.findById(id);

    if (todo == null) {
        // Todo nahi mila, 404 error ke saath response bhejo
        return ResponseEntity.status(404).body("Todo not found.");
    }

    // Success case: Todo mil gaya, 200 OK response bhejo
    return ResponseEntity.ok(todo);
}
```

Flow:

1. **Agar Todo ID Valid Hai:**
2. `todo` milta hai, aur `ResponseEntity.ok(todo)` ke saath `200 OK` response bheja jata hai.
3. **Agar Todo ID Invalid Hai:**
4. `todo` nahi milta, to `ResponseEntity.status(404).body("Todo not found.")` ke saath error response bhejte hain.

Responses:

Todo Found:

```json
{
    "id": 1,
    "title": "Complete Homework",
    "description": "Finish math homework",
    "completed": false
}
```

Status Code: `200 OK`
Todo Not Found:

```json
{
    "message": "Todo not found."
}
```

Status Code: `404 Not Found`

Why Beginners Should Use `ResponseEntity<?>`

- **Problem Without `ResponseEntity<?>`:**
- **Status Code Always `200 OK`:**
- Data na mile ya error ho, tab bhi `200 OK` milta hai (misleading response).
- **Solution With `ResponseEntity<?>`:**

1. **Custom Status Codes:**
2. Success: `200 OK`
3. Data Not Found: `404 Not Found`
4. System Error: `500 Internal Server Error`
5. **Custom Body:**
6. Hum meaningful messages aur data bhej sakte hain.
7. **Dynamic Responses:**
8. Response ki type dynamically decide hoti hai.

Full Beginner-Friendly Code Example

```java
@GetMapping("/todo/{id}")
public ResponseEntity<?> getTodoById(@PathVariable Long id) {
    TodoEntity todo = todoService.findById(id);

    if (todo == null) {
        // 404 Not Found response
        return ResponseEntity.status(404).body("Todo not found.");
    }

    // 200 OK response
    return ResponseEntity.ok(todo);
}
```

Summary:

1. `ResponseEntity<?>` ka use error handling aur dynamic responses ke liye hota hai.
2. `<?>` batata hai ki response body kisi bhi type ki ho sakti hai (e.g., `String`, `Object`, `List`).
3. Agar hum `ResponseEntity<?>` use nahi karte, to default response `200 OK` hota hai, jo galat indication de sakta hai.

==================================

Point To Note

Hinglish Notes: Handling Password Exposure in API Responses with @JsonIgnore

Problem: Password Field Is Sent in API Response with `@JsonIgnore`

Agar aapka Spring Boot application me `UserSignupEntity` object API response me serialize ho raha hai, to **password field** bhi JSON me send ho jata hai. Yeh security ke liye risky hai kyunki hashed password ko bhi API ke through expose karna achha practice nahi hai.

Example Problem

```json
{
    "id": 6,
    "email": "satyam@gmail.com",
    "password": "$2a$10$q5AF/zqMrheOpu5YGnfP7eNR2dwhZR.k.
  ↪ P64t02rcHEiH6QtGzO/6",
    "role": "user",
    "createdAt": "2025-02-09T16:20:34.792+00:00"
}
```

Listing 7: Example JSON Response with Password Field

Issue:

- **Password field API response me visible hai, jo security breach ka reason ban sakta hai.**

Solution 1: Use `@JsonIgnore` on the Password Field

Agar aapko password field ko API response me se **exclude** karna hai, to `UserSignupEntity` class me password field par `@JsonIgnore` annotation lagao.

Updated `UserSignupEntity` Class

```java
import com.fasterxml.jackson.annotation.JsonIgnore;

@Entity
public class UserSignupEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String email;

    @JsonIgnore // Yeh password field ko JSON me serialize hone se
    ↪ rokega
    private String password;

    private String role;

    private LocalDateTime createdAt;

    // Getters and setters
}
```

Listing 8: Updated UserSignupEntity Class

How It Solves the Problem

- **`@JsonIgnore` Kya Karta Hai?** Jackson library ko batata hai ki **password field ko JSON response me include mat karo.**
- Yeh backend me password ko available rakhta hai (e.g., authentication ke liye), lekin frontend ko expose nahi karta.

Response After Fix

```json
{
    "id": 6,
    "email": "satyam@gmail.com",
    "role": "user",
    "createdAt": "2025-02-09T16:20:34.792+00:00"
}
```

Listing 9: Fixed JSON Response Without Password Field

Password field ab JSON me visible nahi hai.

Why Use `@JsonIgnore`?

- **Easy to Use:** Sirf ek line ka annotation lagane se field exclude ho jata hai.
- **Secures Sensitive Data:** API response me sensitive fields (jaise password) ko expose hone se rokta hai.
- **No Impact on Backend Logic:** Password backend me authentication ke liye available rahta hai.

Limitation of `@JsonIgnore`

- **Static Solution:** Agar kisi field ko dynamically hide karna ho (e.g., kisi specific endpoint ke liye), to `@JsonIgnore` kaam nahi karega. Iske liye DTO ya `@JsonView` use karna better hai.

=================================

911em

Point To Note

Relationships (‘@ManyToOne‘) API Responses Mein Nested Objects Automatically Kyu Include Hote Hain

—
Nested Objects Kya Hote Hain? (Simple Explanation with Example)

91.1 Nested Object Kya Hota Hai?

Ek **nested object** woh object hota hai jo kisi aur object ke andar hota hai. Programming mein, yeh tab hota hai jab ek object ka dusre object ke saath relationship hota hai. For example: - Ek ‘Todo‘ object ka ‘User‘ object ke saath relationship ho sakta hai (jo user ne todo create kiya hai). - Jab aap ‘Todo‘ object fetch karte hain, toh ‘User‘ object automatically response mein include ho jata hai kyuki dono ke beech mein relationship hai.

91.2 Nested Objects Ka Example

Maano aapke paas do entities hain: 1. **‘Todo‘ Entity**: Ek task ko represent karti hai. 2. **‘User‘ Entity**: Us user ko represent karti hai jisne task create kiya hai.
‘Todo‘ entity ka ‘User‘ entity ke saath ‘@ManyToOne‘ relationship hai. Jab aap ‘Todo‘ fetch karte hain, toh ‘User‘ object automatically response mein include ho jata hai.

Example

Todo.java

```java
@Entity
public class Todo {
    @Id
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user; // User ke saath relationship
}
```

Jab aap ‘Todo‘ fetch karte hain, toh API response kuch aisa dikhega:

Example

API Response

```
{
    "id": 1,
    "title": "Complete Spring Boot",
    "user": {
        "id": 6,
        "email": "user@example.com",
        "password": "hashed_password",  // Sensitive data
    ↪ expose ho raha hai
        "role": "admin"
    }
}
```

—
Yeh Kyu Hota Hai?

91.3 Jackson Library

Spring Boot **Jackson library** ka use karta hai JSON serialization ke liye. Jackson automatically sabhi fields ko include karta hai, including nested relationships, API response mein.

91.4 Nested Objects Mein Problem

1. **Sensitive Data Leak**: Fields jaise ‘password‘ response mein expose ho jate hain. 2. **Unwanted Data**: Response mein unnecessary data include ho jata hai, jisse uska size badh jata hai.
—
Nested Object Serialization Ko Kaise Rokhein?

91.5 Solution 1: ‘@JsonIgnore‘ Ka Use Karein

Specific fields ko serialization se exclude karne ke liye ‘@JsonIgnore‘ ka use karein.

Example

User.java

```java
@Entity
public class User {
    @Id
    private Long id;

    private String email;

    @JsonIgnore // Password ko serialization se exclude karein
    private String password;

    private String role;
}
```

91.6 Solution 2: ‘@JsonIgnoreProperties‘ Ka Use Karein

Nested relationships mein specific fields ko exclude karne ke liye ‘@JsonIgnoreProperties‘ ka use karein.

Example

Todo.java

```java
@Entity
public class Todo {
    @Id
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "user_id")
    @JsonIgnoreProperties({"password"}) // User se password ko
    ↪  exclude karein
    private User user;
}
```

91.7 Solution 3: DTOs (Data Transfer Objects) Ka Use Karein

DTOs aapko full control dete hain ki API response mein kya data include karna hai.

Example

DTOs

```java
public class TodoDTO {
    private Long id;
    private String title;
    private UserDTO user; // Sirf required fields include
    ↪ karein
}

public class UserDTO {
    private Long id;
    private String email;
    private String role; // Password ko exclude karein
}
```

—
Key Points for Notes
1. **Problem**: - Relationships jaise '@ManyToOne' ki wajah se nested objects automatically API responses mein include ho jate hain.
2. **Yeh Kyu Hota Hai**: - Jackson, jo Spring Boot ka default JSON serializer hai, sabhi fields ko include karta hai, including nested relationships.
3. **Solutions**: - Specific fields ko exclude karne ke liye '@JsonIgnore' ka use karein. - Nested relationships mein fields ko exclude karne ke liye '@JsonIgnoreProperties' ka use karein. - Response structure par full control pane ke liye DTOs ka use karein.
4. **Best Practice**: - Jab bhi aapko API response par full control chahiye, DTOs ka use karein.
—
Final Takeaway "Spring Boot mein relationships jaise '@ManyToOne' ki wajah se nested objects automatically API responses mein include ho jate hain. Sensitive data expose hone se bachne ke liye '@JsonIgnore', '@JsonIgnoreProperties', ya DTOs ka use karein."
—
==================================

Point To Note

Understanding == vs .equals() in Java **In Java (and Spring Boot), when comparing strings, it's important to understand the difference between == and .equals().**

—

92 == kya karta hai?

- == compares **memory reference** (*yani, dono objects ka address memory mein same hai ya nahi*).
- Agar do strings ka **reference** same hai toh hi `==` `true` return karega, **chahe unka content same ho ya na ho**.

Example:

Java Code Example

```java
String s1 = "admin";
String s2 = "admin";

if (s1 == s2) {
    System.out.println("Matched!"); // Ye chalega, kyunki dono ka
    ↪ reference same hai (interned string).
}

String s3 = new String("admin");

if (s1 == s3) {
    System.out.println("Matched!"); // Ye nahi chalega, kyunki
    ↪ reference alag hai, even though content same hai.
}
```

—

93 `.equals()` kya karta hai?

- `.equals()` compares the **content** of the objects (*yani strings ke andar ka actual data compare karega*).
- Isse koi farak nahi padta ki unka reference same hai ya alag.

Example:

Java Code Example

```java
String s1 = "admin";
String s3 = new String("admin");

if (s1.equals(s3)) {
    System.out.println("Matched!"); // Ye chalega, kyunki content (
    ↪ value) same hai.
}
```

—

94 Problem with == in Strings

- Jab aap Spring Boot mein strings like `role` ya `email` ko compare karte ho (`role == "admin"`), toh yeh dangerous ho sakta hai.
- Kyunki kabhi kabhi string alag memory mein store hoti hai (e.g., database se aaye data, HTTP request body), toh `==` fail karega even if the content is the same.

Example:

Java Code Example

```java
String role = request.getAttribute("role"); // Database ya HTTP
    ↪ request se aaye.
if (role == "admin") {
    System.out.println("Role matched!"); // Ye fail karega, kyunki
    ↪ memory references alag hain.
}
```

—

95 `.equals()` Reliable Hai

- Jab aap `.equals()` use karte ho, toh sirf **content match** hona zaruri hai.
- Isliye Spring Boot ya kisi bhi Java application mein string comparison ke liye **hamesha `.equals()` use karna chahiye.**

Example (Correct Way):

Java Code Example

```java
String role = (String) request.getAttribute("role");
if ("admin".equals(role)) { // Always put "constant" first to avoid
    ↪ NullPointerException
    System.out.println("Role matched!"); // Ye hamesha sahi kaam
    ↪ karega.
}
```

—

96 Why Use `"admin".equals(role)` Instead of `role.equals("admin`

- Jab aap `"admin".equals(role)` likhte ho, toh agar `role` null hai, tab bhi code crash nahi karega.
- Par agar `role.equals("admin")` likha aur `role` null nikla, toh **NullPointerException** aayegi.

Example:

Java Code Example

```java
String role = null;

if ("admin".equals(role)) {
    System.out.println("Matched!"); // Safe hai, null par bhi chalega
    ↪ .
}

if (role.equals("admin")) {
    System.out.println("Matched!"); // NullPointerException throw
    ↪ karega.
}
```

—

97 Conclusion (Samajhne Layak Baat)

- `==`: Use tab karo jab tumhe **reference (address)** check karna hai (bohot rare scenarios mein).
- `.equals()`: Strings ka **actual content compare** karne ke liye hamesha use karo.
- **Safe pattern**: `"value".equals(variable)` likho, taaki null pointer ka risk na ho.

==================================

971em

Point To Note

Understanding @OneToMany and @ManyToOne with Cascade and Orphan-Removal

98 Introduction

Agar aap Spring Boot aur Hibernate me relationships samajhna chahte hain, toh ye guide aapke liye hai. Ye guide aapko **@OneToMany** aur **@ManyToOne** annotations ke saath **Cascade** aur **OrphanRemoval** ka concept samjhayega.
—

99 1. Relationship Basics: Parent and Child

99.1 Parent and Child Concept:

- **Parent Entity**: The entity that owns the lifecycle of the related entities (e.g., 'User-SignupEntity' is the parent of 'TodoEntity').
- **Child Entity**: The entity that is dependent on the parent entity (e.g., 'TodoEntity' is the child in the relationship).

99.2 Defining Parent Entity: ‘UserSignupEntity‘

Example

UserSignupEntity.java

```java
@Entity
public class UserSignupEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String email;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL,
    ↪ orphanRemoval = true)
    private List<TodoEntity> todos = new ArrayList<>();
}
```

99.3 Explanation of ‘private List¡TodoEntity¿ todos = new ArrayList¡¿();‘

- **1. List¡TodoEntity¿:**
- Ye ek list hai jo ‘UserSignupEntity‘ ke saath associated sabhi ‘TodoEntity‘ objects ko store karegi.
- Matlab, ek user ke paas multiple todos ho sakte hain, aur ye list un todos ko hold karegi.
- **2. ‘new ArrayList¡¿()‘:**
- Ye list ko initialize karta hai aur ek empty ‘ArrayList‘ banata hai.
- **Kyun zaroori hai?**
- Agar list initialize nahi ki jaye (yani ‘new ArrayList¡¿()‘ na likha jaye), to ye list ‘null‘ rahegi.
- ‘Null‘ list par koi bhi operation (jaise add ya remove) karne se error aayega (‘NullPointerException‘).
- Isliye, ‘new ArrayList¡¿()‘ likhna zaroori hai taaki list ready ho objects ko store karne ke liye.
- **3. Purpose (Maqsad):**
- Ye list ‘UserSignupEntity‘ aur ‘TodoEntity‘ ke beech ka relationship represent karti hai.
- Ek user (‘UserSignupEntity‘) ke paas multiple todos (‘TodoEntity‘) ho sakte hain, aur ye list un todos ko store karti hai.

—

100 2. Child Entity: ‘TodoEntity‘

Example

TodoEntity.java

```java
@Entity
public class TodoEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private UserSignupEntity user;
}
```

100.1 Explanation:

- **1. ‘@ManyToOne‘:**
- Define karta hai ki ek ‘TodoEntity‘ ek ‘UserSignupEntity‘ ke saath associated hai.
- **2. ‘@JoinColumn(name = ”user**$_i$$d$”)‘ : $Databaseme‘user_id‘naamkaforeignkeycolumncreatekaregaj$

—

101 3. Explanation of Each Part

101.1 1. ‘@OneToMany‘:

- Yeh relationship define karta hai ki ek **UserSignupEntity** (Parent) ke paas **many TodoEntity** (Child) ho sakte hain.

101.2 2. ‘mappedBy = ”user”‘:

- ‘mappedBy‘ batata hai ki relationship ka ”ownership” kis side se hai.
- ‘mappedBy = ”user”‘ ka matlab hai ki ‘TodoEntity‘ me ek field ‘user‘ hai jo relationship ko define karta hai.
- Iska matlab:
- ‘UserSignupEntity‘ relationship ko ”own” nahi karta.
- ‘TodoEntity‘ ke ‘@ManyToOne‘ annotation ke andar ‘user‘ field relationship ke liye responsible hai.

101.3 3. ‘cascade = CascadeType.ALL‘:

- Cascade ka matlab hai ki:
- Jab ‘UserSignupEntity‘ par koi operation (save, update, delete) perform hota hai, to uske saath linked **todos par bhi wahi operation chalega**.
- Example:
- Agar ek ‘User‘ delete hota hai, to uske saare ‘Todos‘ automatically delete ho jayenge.

101.4 4. ‘orphanRemoval = true‘:

- Orphan ka matlab hai: **A child entity that is no longer associated with its parent.**
- ‘orphanRemoval = true‘ ka matlab hai ki agar:
- ‘TodoEntity‘ ko ‘UserSignupEntity‘ ki ‘todos‘ list se hata diya jaye, to woh database me bhi automatically delete ho jayega.

- Example:

Example

Removing a Todo

```java
    user.getTodos().remove(todo); // Removes the Todo from the
    ↪   User's list

```

- Iske baad, woh 'Todo' entity database se bhi delete ho jayega.

—

102 4. Why Cascading in '@ManyToOne' is Not Recommended?

102.1 Problem with Cascade in '@ManyToOne':

Example

Not Recommended

```java
@ManyToOne(cascade = CascadeType.ALL) //     Not Recommended
@JoinColumn(name = "user_id", nullable = false)
private UserSignupEntity user;
```

- Iska matlab hai:
- Jab ek **TodoEntity** par operation perform hota hai, to uske associated **UserSignupEntity** par bhi woh operation chalega.
- **Issue Example:**
- Ek 'Todo' delete karte ho to associated 'User' bhi delete ho jayega.
- This is **wrong behavior**, kyunki ek task delete hone se poora user delete nahi hona chahiye.

—

103 5. Real-Life Example: Parent-Child Flow

103.1 1. Parent ('UserSignupEntity'):

Example

UserSignupEntity.java

```java
@Entity
public class UserSignupEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String email;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL,
    ↪ orphanRemoval = true)
    private List<TodoEntity> todos = new ArrayList<>();
}
```

103.2 2. Child (‘TodoEntity‘):

Example

TodoEntity.java

```java
@Entity
public class TodoEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private UserSignupEntity user;
}
```

103.3 Flow Example:

- **1. Delete a Parent (UserSignupEntity):**
- Jab ek user delete hota hai, to uske saare todos (‘cascade = CascadeType.ALL‘) delete ho jate hain.
- **2. Remove a Todo from User’s List:**

Example

Removing a Todo

```java
    user.getTodos().remove(todo);

```

- ‘orphanRemoval = true‘ hone ki wajah se woh ‘Todo‘ database se bhi delete ho jata hai.
- **3. Delete a Todo Directly:**
- ‘TodoEntity‘ delete karna user ke upar koi impact nahi dalta.

—

104 6. Beginner-Friendly Explanation of Terms

- **1. ‘mappedBy‘:**
- Batata hai ki **relationship ka control parent ya child me kahan hai.**
- Example:
- ‘mappedBy = ”user”‘ ka matlab hai ki relationship ko child (‘TodoEntity‘ ke ‘user‘ field) define kar raha hai.
- **2. ‘cascade‘:**
- Parent ke operations ko automatically child entities tak propagate karta hai.
- Example:
- ‘CascadeType.ALL‘: Save, update, delete, sab parent ke saath child par bhi chalega.
- **3. ‘orphanRemoval‘:**
- Agar ek child entity parent ki list se remove ho jaye, to woh database se bhi automatically delete ho jata hai.

—

105 7. Final Notes for Beginners

- **1. Cascading is One-Way:**
- Cascade behavior hamesha **parent se child** hota hai, reverse nahi.
- **2. Parent Lifecycle Controls the Child:**
- Parent (‘UserSignupEntity‘) ke operations (delete, update) child (‘TodoEntity‘) ko impact karte hain.
- **3. Avoid Cascade in @ManyToOne:**
- ‘@ManyToOne‘ me cascading mat lagao, kyunki child entity ka lifecycle parent ke control me hona chahiye.
- **4. Key Annotations to Remember:**
- ‘@OneToMany‘ (Parent ke liye cascading behavior define karna).
- ‘@ManyToOne‘ (Child ke liye relationship define karna).
- ‘orphanRemoval = true‘ (Orphaned child entities ko automatically delete karna).

—
================================

Point To Note

Object and var: Real Use Cases in Spring Boot

—
1. Using ‘Object‘: When the Variable’s Type Is Dynamic

105.1 Problem:

Suppose a user can update any field in a ‘TodoEntity‘. Since you don’t know in advance which field (e.g., ‘title‘, ‘description‘, ‘completed‘) the user wants to update, the type of data could be different (string, boolean, etc.).

105.2 Solution: Use ‘Object‘

Example

TodoController.java

```java
@RestController
@RequestMapping("/todo")
public class TodoController {

    @Autowired
    private TodoService todoService;

    @PatchMapping("/{id}")
    public ResponseEntity<?> updateTodoField(@PathVariable
    ↪ Long id, @RequestBody Map<String, Object> updates) {
        // Fetch the Todo to update
        TodoEntity todo = todoService.findById(id);
        if (todo == null) {
            return ResponseEntity.status(404).body("Todo not
    ↪ found");
        }

        // Iterate over the fields to update
        for (String key : updates.keySet()) {
            Object value = updates.get(key); // Object handles
    ↪  any type of value

            // Check which field to update
            switch (key) {
                case "title":
                    todo.setTitle((String) value); // Cast to
    ↪ String
                    break;
                case "description":
                    todo.setDescription((String) value); //
    ↪ Cast to String
                    break;
                case "completed":
                    todo.setCompleted((Boolean) value); //
    ↪ Cast to Boolean
                    break;
                default:
                    return ResponseEntity.status(400).body("
    ↪ Invalid field: " + key);
            }
        }

        // Save the updated Todo
        todoService.save(todo);
        return ResponseEntity.ok(todo);
    }
}
```

Why Use ‘Object‘ Here?

- The value from the ‘updates‘ map could be of any type (e.g., ‘String‘ for ‘title‘, ‘Boolean‘ for ‘completed‘), so you use ‘Object‘ to handle it.
- Later, you cast it to the correct type when updating the field.

105.3 Request Example:

Example

Input JSON

```json
{
    "title": "Learn Spring Boot",
    "completed": true
}
```

Example

Response

```json
{
    "id": 1,
    "title": "Learn Spring Boot",
    "description": "Basic Spring Boot tasks",
    "completed": true
}
```

—

2. Using ‘var‘: When the Type Can Be Automatically Inferred

105.4 Problem:

In many cases, the datatype is clear from the context, but writing the full type makes the code longer and harder to read. For example, fetching a list of todos (‘List¡TodoEntity¿‘).

105.5 Solution: Use ‘var‘

Example

TodoService.java

```java
@Service
public class TodoService {

    @Autowired
    private TodoRepository todoRepository;

    public List<TodoEntity> findAllTodos() {
        // Using var to simplify code
        var todos = todoRepository.findAll(); // Compiler
    ↪ infers todos as List<TodoEntity>
        return todos;
    }
}
```

Why Use ‘var‘ Here?

- You don’t have to write ‘List¡TodoEntity¿‘ explicitly; the compiler knows the type from ‘findAll()‘ and infers it automatically.
- It simplifies the code without losing clarity.

105.6 Example in a Loop:

Example

TodoController.java

```java
@GetMapping("/todos")
public ResponseEntity<?> getAllTodos() {
    var todos = todoService.findAllTodos(); // Compiler infers
    ↪  todos as List<TodoEntity>

    for (var todo : todos) { // Compiler infers todo as
    ↪ TodoEntity
        System.out.println(todo.getTitle());
    }

    return ResponseEntity.ok(todos);
}
```

—
Side-by-Side Simple Comparison

105.7 ‘Object‘ Example: Dynamic Field Update

Example

Dynamic Field Update

```java
Object value = updates.get("completed"); // Value could be
    ↪ Boolean or String
todo.setCompleted((Boolean) value); // Cast to Boolean
```

105.8 ‘var‘ Example: Fetching Todos

Example

Fetching Todos

```java
var todos = todoRepository.findAll(); // Type inferred as List
    ↪ <TodoEntity>
```

—
Key Takeaways:

· Use ‘Object‘ when the type of data is dynamic and unknown at compile time.
· Use ‘var‘ when the type is clear from the context, and you want to simplify the code.
· Both ‘Object‘ and ‘var‘ are powerful tools in Spring Boot for handling dynamic and inferred types.

—
==================================