SPRING BOOT

SPRING BOOT

Definition:
Spring Boot is a **framework built on top of Spring** that helps developers create **stand-alone, production-ready Java applications quickly** with **minimum configuration**.

Why We Use It (Purpose)

- To **reduce complex configuration** of Spring
- To develop applications **faster**
- To avoid writing **boilerplate code**
- To create **microservices and REST APIs easily**

Key Points / Core Concepts

- **Auto Configuration** – Automatically configures beans based on dependencies
- **Starter Dependencies** – Predefined dependency sets (like spring-boot-starter-web)
- **Embedded Server** – Comes with Tomcat/Jetty, no external server needed
- **Production Ready** – Supports monitoring, health checks, metrics

---

Real-Life Example (Easy to Understand)
Imagine buying a **ready-made laptop**:

- OS already installed
- Drivers already configured
- Just turn it on and start working

Spring Boot is like that laptop.
Spring (without Boot) is like **assembling PC parts manually**.

Technical / Practical Example

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

This single class:

- Enables auto-configuration
- Starts the embedded server
- Runs the application

Advantages

- Faster development
- Less configuration
- Easy to learn for freshers
- Built-in server
- Good support for microservices

---

7. Interview Questions & Answers
Q1. What is Spring Boot?
A. Spring Boot is a framework that simplifies Spring application development by providing auto-configuration, starter dependencies, and embedded servers.
Q2. Difference between Spring and Spring Boot?
A. Spring needs a lot of configuration, while Spring Boot minimizes configuration and speeds up development.

Q3. Does Spring Boot require external server?
A. No, it has an embedded server like Tomcat.

Summary (Quick Revision)
Spring Boot makes Spring development **easy, fast, and configuration-free**, especially useful for **REST APIs and microservices.**

Spring Initializr (start.spring.io)

What is Spring Initializr?
Spring Initializr is a **web-based project generator** that:

- Creates a **ready-to-run Spring Boot application**
- Manages **Maven / Gradle** configuration
- Sets up **dependencies automatically**
- Follows **Spring best practices**

Website: https://start.spring.io

Why Use Spring Initializr?
Instead of manually:

- Creating folders
- Writing pom.xml
- Managing versions

Spring Initializr does everything **correctly and fast.**

- Saves time
- Avoids configuration mistakes
- Industry-standard structure

Main Options Explained
1. Project

- **Maven Project** → Most commonly used
- **Gradle Project** → Faster builds (optional)

Beginners: **Maven Project**

---

2. Language

- **Java** (most common)
- Kotlin / Groovy (advanced)

Choose **Java**

---

3. Spring Boot Version

- Default is usually **stable**
- Avoid SNAPSHOT unless needed

Keep **default**

---

4. Project Metadata

Field	Meaning
Group	Package root (e.g. com.company)
Artifact	Project name
Name	App name
Package Name	Base package

Field	Meaning
Packaging	Jar (default)
Java	Version (17 or 21 recommended)

Example:
Group: com.revpay
Artifact: revpay-app
Package: com.revpay

Dependencies (Most Important)
You choose what your app needs:
Common Dependencies

- **Spring Web** → REST APIs
- **Spring Data JPA** → Database access
- **Spring Security** → Authentication
- **Validation** → Input validation
- **Lombok** → Reduce boilerplate
- **H2 / MySQL / Oracle Driver** → Database

Example for backend project:
Spring Web
Spring Data JPA
Spring Security
Lombok
Oracle Driver

Generate & Import

1. Click **Generate**
2. ZIP file downloads
3. Extract it
4. Open in **IntelliJ / Eclipse / STS**
5. Run:

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

Folder Structure (Auto-created)

```
src/main/java
 └── com.revpay
      ├── RevpayApplication.java
      ├── controller
      ├── service
      ├── dao
      ├── model

src/main/resources
 ├── application.properties
 └── static / templates
```

How It Helps in Real Projects (Like RevPay)

- Fast project setup
- Easy dependency management
- Clean architecture

- Team-friendly
- Production ready

Interview Answer (Short & Strong)
Spring Initializr is an official Spring tool used to generate a pre-configured Spring Boot project with required dependencies and best-practice structure, reducing setup time and configuration errors.

Spring Initializr (start.spring.io)

Definition:
Spring Initializr is a **web-based tool** used to **quickly create a Spring Boot project** with required **dependencies, configuration, and project structure**.

Why We Use It (Purpose)
- To avoid **manual project setup**
- To generate a **ready-to-run Spring Boot project**
- To select dependencies easily (Web, JPA, Security, etc.)
- To save time during **training and real projects**

Key Points / Core Concepts
- Provides **pre-configured project structure**
- Supports **Maven and Gradle**
- Allows selecting **Spring Boot version**
- Generates a **ZIP project** or directly imports into IDE
- Maintained by the **Spring team**

Real-Life Example (Easy to Understand)
Think of Spring Initializr like an **online application form**:
- You select what you need (course, branch, documents)
- Submit the form
- You get everything prepared for you

Spring Initializr prepares your project in the same way.

Technical / Practical Example (How to Use)
Steps to create a project:
1. Go to **start.spring.io**
2. Select:
    - Project: **Maven**
    - Language: **Java**
    - Spring Boot Version
3. Enter:
    - Group: com.example
    - Artifact: demo
4. Add Dependencies:
    - Spring Web
    - Spring Data JPA
5. Click **Generate**

A complete Spring Boot project is created.

---

6. Advantages
- No manual configuration needed

- Standard project structure
- Beginner-friendly
- Reduces setup errors
- Industry-standard tool

---

7. Interview Questions & Answers

Q1. What is Spring Initializr?

A. It is a tool used to generate Spring Boot projects with required dependencies and default configuration.

Q2. Is Spring Initializr mandatory to create Spring Boot project?

A. No, but it is the easiest and most recommended way.

Q3. Who maintains Spring Initializr?

A. The Spring team.

Summary (Quick Revision)

Spring Initializr helps developers **quickly start Spring Boot projects** without manual setup.

Spring Boot Project Structure

Definition:

Spring Boot project structure is a **standard folder layout** that organizes application code, configuration files, and resources in a **clean and maintainable way**.

Why We Use It (Purpose)

- To **separate responsibilities** (controller, service, repository)
- To make code **easy to understand and maintain**
- To follow **industry standards** used in companies like Cognizant
- To support **scalability and teamwork**

Standard Spring Boot Project Structure

```
project-name
│
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.example.project
│   │   │       ├── controller
│   │   │       ├── service
│   │   │       ├── repository
│   │   │       ├── model (entity)
│   │   │       └── ProjectApplication.java
│   │   │
│   │   └── resources
│   │       ├── application.properties
│   │       ├── static
│   │       └── templates
│   │
│   └── test
│       └── java
│
├── pom.xml
```

└── mvnw / gradlew

Explanation of Important Folders

controller

- Handles **HTTP requests**
- Exposes REST APIs
- Uses @Controller/@RestController

service

- Contains **business logic**
- Communicates between controller and repository
- Uses @Service

repository

- Handles **database operations**
- Uses Spring Data JPA
- Uses @Repository

model / entity

- Represents **database tables**
- Uses @Entity

resources

- Stores **configuration and static files**

Real-Life Example (Easy to Understand)

Think of a **company office**:

- Reception → Controller
- Manager → Service
- Accountant → Repository
- Files → Database

Each department has a clear responsibility.

Important Files

ProjectApplication.java

- Main class
- Starts the Spring Boot application
- Must be in **base package**

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

application.properties

- Stores database, server, and app configurations

Interview Questions & Answers

Q1. Why should main class be in base package?

A. To allow Spring Boot to scan all sub-packages automatically.

Q2. Where do we write database configuration?

A. In application.properties or application.yml.

Q3. What happens if we place controller outside base package?

A. Spring will not detect it.

Summary (Quick Revision)

Spring Boot project structure helps maintain **clean, organized, and scalable applications** by separating concerns clearly.

Auto-Configuration (Spring Boot)

Definition:
Auto-configuration is a **Spring Boot feature** that **automatically configures beans and application settings** based on the **dependencies present in the project.**

Why We Use It (Purpose)

- To avoid **manual configuration**
- To reduce **boilerplate code**
- To speed up application development
- To let developers focus on **business logic**

Key Points / Core Concepts

- Works based on **classpath dependencies**
- Uses **default configurations**
- Can be **overridden** by developers
- Enabled by @SpringBootApplication
- Internally uses @EnableAutoConfiguration

Real-Life Example (Easy to Understand)
When you insert a **SIM card** into your phone:

- Network settings are configured automatically
- You don't configure tower or frequency manually

Spring Boot auto-configuration works the same way.

Technical / Practical Example
If you add this dependency:

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

Spring Boot automatically:

- Configures **Tomcat**
- Sets up **DispatcherServlet**
- Enables **JSON conversion**

No manual setup needed.

How Auto-Configuration Works (Internally – Fresher Level)

1. Spring Boot checks **dependencies**
2. Reads spring.factories / auto-config classes
3. Applies configuration using conditions like:
    - @ConditionalOnClass
    - @ConditionalOnMissingBean
4. Creates beans automatically

Advantages

- Less configuration
- Faster setup
- Reduces errors
- Beginner-friendly
- Production-ready defaults

Interview Questions & Answers
Q1. What is auto-configuration?
A. It automatically configures Spring beans based on project dependencies.

Q2. Can we disable auto-configuration?
A. Yes, using exclude in @SpringBootApplication.
Q3. Can we override auto-configuration?
A. Yes, by defining our own beans.

Summary (Quick Revision)
Auto-configuration allows Spring Boot to **configure the application automatically**, reducing manual work and speeding up development.

Starter Dependencies (Spring Boot)

Definition:
Starter dependencies are **predefined dependency bundles** in Spring Boot that include **all required libraries** for a specific functionality.

Why We Use It (Purpose)

- To avoid adding **multiple dependencies manually**
- To reduce **version compatibility issues**
- To simplify **project setup**
- To speed up development

Key Points / Core Concepts

- Provided by Spring Boot
- Follow naming pattern: spring-boot-starter-*
- Automatically compatible versions
- Works with **auto-configuration**
- Reduces dependency conflicts

Real-Life Example (Easy to Understand)
Think of ordering a **combo meal**:

- Burger
- Fries
- Drink

Starter dependency is like a combo – everything you need comes together.

Common Starter Dependencies (Important for Freshers)

Starter Dependency	Purpose
spring-boot-starter-web	REST APIs, MVC, Tomcat
spring-boot-starter-data-jpa	Database & JPA
spring-boot-starter-security	Authentication & Authorization
spring-boot-starter-test	Unit & Integration Testing
spring-boot-starter-thymeleaf	UI with Thymeleaf

Technical / Practical Example

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

This automatically includes:

- Hibernate

- JPA APIs
- Transaction management

Advantages

- Less configuration
- Easy dependency management
- Cleaner pom.xml
- Faster learning for freshers
- Industry standard approach

---

8. Interview Questions & Answers

Q1. What are starter dependencies?

A. Predefined dependency sets that simplify Spring Boot project configuration.

Q2. Can we create custom starter?

A. Yes, Spring Boot supports custom starters.

Q3. How starters work with auto-configuration?

A. Starters bring dependencies, and auto-configuration configures them automatically.

Summary (Quick Revision)

Starter dependencies simplify Spring Boot development by providing **ready-made dependency sets** for common functionalities.

@SpringBootApplication

Definition:

@SpringBootApplication is a **core Spring Boot annotation** used to **mark the main class** of a Spring Boot application and **enable auto-configuration, component scanning, and configuration support.**

Why We Use It (Purpose)

- To **start** a Spring Boot application
- To **enable auto-configuration**
- To tell Spring **where to scan components**
- To reduce multiple annotations into **one single annotation**

Key Points / Core Concepts

- Placed on **main class**
- Combines **three important annotations**
- Enables **embedded server startup**
- Mandatory for Spring Boot applications

Internal Annotations (Very Important)

@SpringBootApplication is a **combination of**:

1. **@SpringBootConfiguration**
    - Marks class as configuration class
2. **@EnableAutoConfiguration**
    - Enables Spring Boot auto-configuration
3. **@ComponentScan**
    - Scans components in base package and sub-packages

Real-Life Example (Easy to Understand)

Think of a **TV remote**:

- One button turns on TV
- Controls volume
- Changes channels

@SpringBootApplication is that **one button** that enables everything.

Technical / Practical Example

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

This single annotation:

- Starts application
- Scans components
- Configures beans automatically

Important Rules

- Main class must be in **base package**
- All controllers, services, repositories should be in **sub-packages**
- Otherwise Spring won't detect them

Interview Questions & Answers

Q1. What is @SpringBootApplication?

A. It is a shortcut annotation that enables auto-configuration, component scanning, and configuration support.

Q2. Can we use @EnableAutoConfiguration alone?

A. Yes, but @SpringBootApplication is preferred.

Q3. What happens if we remove @ComponentScan?

A. Spring will not detect components.

Summary (Quick Revision)

@SpringBootApplication is the **heart of Spring Boot**, enabling auto-configuration and component scanning with a single annotation.

---

application.properties (Spring Boot)

Definition:

application.properties is a **configuration file** in Spring Boot used to **define application-level settings** such as **database, server, logging, and security configurations.**

Why We Use It (Purpose)

- To avoid **hardcoding values** in Java code
- To **externalize configuration**
- To easily **change behavior without code changes**
- To manage **environment-specific configurations**

Key Points / Core Concepts

- Located in src/main/resources
- Uses **key = value** format
- Automatically loaded by Spring Boot

- Supports **profiles** (dev, test, prod)
- Can be replaced by application.yml

Real-Life Example (Easy to Understand)
Think of **mobile settings**:
- Wi-Fi
- Brightness
- Language

You change settings without changing the phone itself.
Same way, application.properties controls app behavior.

Common Configurations (Very Important)
Server Configuration
server.port=8081
Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
spring.datasource.password=root
JPA / Hibernate Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

Profiles Example
spring.profiles.active=dev
Files:
- application-dev.properties
- application-prod.properties

Advantages
- Centralized configuration
- Easy to maintain
- Environment-specific support
- Cleaner code
- Industry standard practice

---

8. Interview Questions & Answers
Q1. What is application.properties used for?
A. It is used to configure Spring Boot application settings.
Q2. Can we have multiple application.properties files?
A. Yes, using Spring profiles.
Q3. Difference between application.properties and application.yml?
A. Properties uses key-value format, YAML uses hierarchical format.

Summary (Quick Revision)
application.properties helps manage **configuration externally**, making Spring Boot applications **flexible and maintainable**.

Profile Management (Spring Boot Profiles)

Definition:
Profile management in Spring Boot allows us to **define different configurations for different environments** (like **dev, test, prod**) using **Spring Profiles**.

Why We Use It (Purpose)

- To use **different databases** for different environments
- To avoid changing configuration manually
- To maintain **clean and safe deployments**
- To support **real-time project environments**

Key Points / Core Concepts

- Profiles are **environment-specific**
- Common profiles: dev, test, prod
- Spring loads config based on **active profile**
- Supports both **properties and YAML**
- Beans can be profile-specific

Real-Life Example (Easy to Understand)

Think of **office ID cards**:

- Trainee → Limited access
- Employee → Normal access
- Admin → Full access

Same application, different behavior based on profile.

Configuration Example

application.properties

```
spring.profiles.active=dev
```

application-dev.properties

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/devdb
```

application-prod.properties

```
server.port=8080
spring.datasource.url=jdbc:mysql://prod-server:3306/proddb
```

Profile-Specific Beans

```
@Profile("dev")
@Bean
public DataSource devDataSource() {
    return new HikariDataSource();
}
```

Ways to Activate Profiles

- application.properties
- JVM argument: -Dspring.profiles.active=prod
- Environment variable
- Command-line argument

Advantages

- Clean configuration management
- Easy switching between environments
- Prevents production mistakes
- Industry-standard practice

---

9. Interview Questions & Answers

Q1. What are Spring Profiles?

A. Spring Profiles allow environment-based configuration.

Q2. Can multiple profiles be active?

A. Yes, multiple profiles can be active at the same time.

Q3. What happens if no profile is active?
A. Default profile is used.

Summary (Quick Revision)
Profile management helps run the **same application with different configurations** for different environments safely and efficiently.

What is DispatcherServlet? (Spring MVC – Simple & Clear)
DispatcherServlet is the FRONT CONTROLLER of Spring MVC
It:

- Receives **all HTTP requests**
- Decides **which controller** should handle them
- Manages the **entire request–response flow**

Think of it as the **traffic police** of a Spring web application.

Why is it called "Dispatcher"?
Because it:

- **Dispatches (routes)** requests to the correct controller
- Coordinates between:
  - Controller
  - Service
  - View (JSP/Thymeleaf/JSON)

Request Flow (Very Important)

```
Client (Browser / Postman)
        ↓
DispatcherServlet
        ↓
HandlerMapping
        ↓
Controller (@Controller / @RestController)
        ↓
Service Layer
        ↓
Controller returns Model / Response
        ↓
ViewResolver (if UI app)
        ↓
DispatcherServlet
        ↓
Client Response
```

What does DispatcherServlet actually do?

Responsibility	Description
Front Controller	Central entry point
Handler Mapping	Finds correct controller
Handler Adapter	Invokes controller method
View Resolution	Finds correct view
Exception Handling	Handles errors centrally
Interceptors	Applies pre/post logic

DispatcherServlet in Spring Boot
You DON'T create it manually
Spring Boot:
- Auto-configures it
- Maps it to / (all requests)

```
@SpringBootApplication
public class Application { }

Behind the scenes:
@Bean
public DispatcherServlet dispatcherServlet() { ... }
```

Traditional Spring (Non-Boot)
You had to configure it manually:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
```

Why DispatcherServlet is IMPORTANT (Interview)
- Enables **loose coupling**
- Centralizes request handling
- Supports REST + MVC
- Handles validation, security, exceptions

DispatcherServlet vs Controller

DispatcherServlet	Controller
Entry point	Handles business logic
One per app	Many controllers
Framework-level	Application-level

One-line Interview Answer
DispatcherServlet is the central front controller in Spring MVC that receives all HTTP requests and dispatches them to appropriate controllers.

Real-life Analogy
- DispatcherServlet → **Receptionist**
- Controllers → **Departments**
- ViewResolver → **Output desk**

MVC Architecture (Model–View–Controller)

Definition:
MVC Architecture is a **design pattern** that divides an application into **three layers**: **Model**, **View**, and **Controller**, to **separate concerns** and make the application **clean and maintainable**.

Why We Use It (Purpose)
- To separate **business logic, UI, and request handling**

- To improve **code maintainability**
- To support **teamwork and scalability**
- To follow **industry-standard architecture**

Key Components / Core Concepts

Model

- Represents **data and business logic**
- Maps to **database entities**
- Contains **business rules**

View

- Responsible for **user interface**
- Displays data to users
- Examples: HTML, JSP, Thymeleaf

Controller

- Handles **user requests**
- Acts as a **bridge between Model and View**
- Uses @Controller or @RestController

Flow of MVC Architecture

1. User sends request
2. Controller receives request
3. Controller calls Service / Model
4. Model processes data
5. Controller returns View or Response

Real-Life Example (Easy to Understand)

Think of a **restaurant**:

- Customer → User
- Waiter → Controller
- Kitchen → Model
- Food served → View

Each has a clear role.

MVC in Spring Boot (Practical View)

```
Controller → Service → Repository → Database
              ↑
            Model
```

- Controller handles request
- Service applies business logic
- Repository accesses database

Advantages

- Clear separation of concerns
- Easy maintenance
- Reusable code
- Better testing
- Scalable architecture

---

8. Interview Questions & Answers

Q1. What is MVC Architecture?

A. It is a design pattern that separates application into Model, View, and Controller.

Q2. Is Spring Boot REST application MVC?

A. Yes, REST uses MVC but returns JSON instead of views.

Q3. Difference between @Controller and @RestController?
A. @Controller returns views, @RestController returns data (JSON).

Summary (Quick Revision)
MVC architecture separates **data, UI, and request handling**, making applications **clean, scalable, and easy to manage.**

Spring Boot and Spring MVC Integration

Definition:
Spring Boot and Spring MVC integration means **using Spring Boot to automatically configure and run a Spring MVC application** with **minimal setup and no manual configuration.**

Why We Use It (Purpose)
- To build **web applications and REST APIs quickly**
- To avoid manual Spring MVC configuration
- To use **MVC architecture easily**
- To develop **production-ready applications**

Key Points / Core Concepts
- Spring Boot **internally uses Spring MVC**
- spring-boot-starter-web enables MVC
- DispatcherServlet is auto-configured
- Supports **REST APIs and web views**
- Uses **embedded server (Tomcat)**

How Integration Happens (Behind the Scenes)
When we add:
spring-boot-starter-web
Spring Boot automatically:
- Enables **Spring MVC**
- Configures **DispatcherServlet**
- Sets up **ViewResolvers**
- Enables **JSON conversion (Jackson)**
- Starts embedded **Tomcat**

Request Flow (Very Important for Interview)
1. Client sends HTTP request
2. **DispatcherServlet** receives request
3. DispatcherServlet finds **Controller**
4. Controller processes request (via Service)
5. Response returned as:
    - View (HTML) OR
    - JSON (REST)

Practical Example
```
@RestController
@RequestMapping("/hello")
public class HelloController {
    @GetMapping
    public String hello() {
```

```
        return "Hello Spring Boot MVC";
    }
}
```
No XML, no servlet config – Spring Boot handles it.

Advantages

- Zero XML configuration
- Faster development
- Easy REST API creation
- Clean MVC architecture
- Ideal for microservices

---

9. Interview Questions & Answers

Q1. Does Spring Boot replace Spring MVC?

A. No, Spring Boot uses Spring MVC internally.

Q2. Which dependency is needed for MVC in Spring Boot?

A. spring-boot-starter-web.

Q3. Who configures DispatcherServlet in Spring Boot?

A. Spring Boot auto-configuration.

Summary (Quick Revision)

Spring Boot simplifies Spring MVC by **auto-configuring all components**, making web development **fast and easy**.

Controllers (Spring Boot / Spring MVC)

Definition:

A Controller is a **Spring component** that **handles incoming HTTP requests**, processes them using business logic, and **returns a response** (view or data).

Why We Use Controllers (Purpose)

- To **receive client requests**
- To **route requests to business logic**
- To act as a **bridge between client and service layer**
- To implement **MVC architecture**

Key Points / Core Concepts

- Controllers are part of **presentation layer**
- Defined using **annotations**
- Should not contain heavy business logic
- Communicates with **Service layer**
- Returns **View or JSON response**

Types of Controllers

@Controller

- Used for **web applications**
- Returns **views (HTML/JSP/Thymeleaf)**

@RestController

- Used for **REST APIs**
- Returns **JSON/XML**
- Combination of @Controller + @ResponseBody

Real-Life Example (Easy to Understand)

Think of a **customer support desk**:

- Customer asks a question (request)
- Support agent listens and forwards it
- Customer gets an answer (response)

Controller plays the support agent role.

Common Controller Annotations

Annotation	Purpose
@RestController	Marks class as REST controller
@Controller	Marks MVC controller
@RequestMapping	Maps URL
@GetMapping	Handle GET request
@PostMapping	Handle POST request
@PutMapping	Handle PUT request
@DeleteMapping	Handle DELETE request
@RequestBody	Read request data
@PathVariable	Read URL value
@RequestParam	Read query parameter

Practical Example

```
@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping("/{id}")
    public String getUser(@PathVariable int id) {
        return "User ID: " + id;
    }
}
```

Controller receives request and returns response.

Best Practices (Important for Freshers)

- Keep controller **thin**
- Business logic in **service layer**
- Use proper HTTP methods
- Validate input
- Handle exceptions globally

Interview Questions & Answers

Q1. What is a Controller in Spring?

A. It handles HTTP requests and returns responses.

Q2. Difference between @Controller and @RestController?

A. @Controller returns views, @RestController returns JSON.

Q3. Can we call repository directly from controller?

A. No, best practice is Controller → Service → Repository.

Summary (Quick Revision)

Controllers handle **client requests**, delegate logic to services, and return **responses**, forming the entry point of Spring applications.

@Controller and @RestController (Spring MVC / Spring Boot)

Definition:
@Controller
@Controller is used to create **web (MVC) controllers** that **return views** such as HTML, JSP, or Thymeleaf pages.
@RestController
@RestController is used to create **RESTful web services** that **return data (JSON/XML)** directly to the client.

Why We Use Them (Purpose)

- **@Controller**
  - For **UI-based applications**
  - When we need **pages/views**
- **@RestController**
  - For **REST APIs**
  - When frontend and backend are **separate**

Key Points / Core Concepts

- Both handle **HTTP requests**
- Both are part of **Controller layer**
- @RestController = @Controller + @ResponseBody
- Used with **Spring MVC**

Main Differences (Very Important – Interview)

Feature	@Controller	@RestController
Purpose	Web MVC	REST API
Return Type	View name	JSON / XML
@ResponseBody	Required	Not required
Usage	UI-based apps	Microservices / APIs

Real-Life Example (Easy to Understand)

- **@Controller** → Movie Theater
  - Shows the movie on screen (View)
- **@RestController** → OTT API (Netflix)
  - Sends data to mobile/TV app (JSON)

Practical Examples
@Controller Example

```
@Controller
public class PageController {
    @GetMapping("/home")
    public String homePage() {
        return "home"; // returns home.html
    }
}
```

@RestController Example

```
@RestController
public class UserController {
    @GetMapping("/user")
    public User getUser() {
        return new User(1, "John");
    }
```

}
Automatically converted to JSON.

When to Use Which?

- Use **@Controller** when:
  - You return **HTML pages**
  - Using Thymeleaf / JSP
- Use **@RestController** when:
  - Building **REST APIs**
  - Working with frontend frameworks (React, Angular)

---

8. Interview Questions & Answers

Q1. What is the difference between @Controller and @RestController?
A. @Controller returns views, @RestController returns data.
Q2. Why @ResponseBody is not needed in @RestController?
A. Because it is already included internally.
Q3. Can we return JSON from @Controller?
A. Yes, by using @ResponseBody.

Summary (Quick Revision)

- @Controller → View-based MVC
- @RestController → REST APIs
- @RestController simplifies API development

@RequestMapping (Spring MVC / Spring Boot)

Definition:
@RequestMapping is a **Spring MVC annotation** used to **map HTTP requests (URL + method)** to **controller classes or methods.**

Why We Use It (Purpose)

- To **define which URL** should call which controller
- To **handle different HTTP methods**
- To create **structured and readable APIs**
- To control **request routing**

Key Points / Core Concepts

- Can be used at **class level and method level**
- Supports **multiple HTTP methods**
- Acts as a **parent mapping** when used at class level
- Base annotation for all shortcut mappings

Real-Life Example (Easy to Understand)
Think of **house addresses**:

- Street name → Class-level mapping
- House number → Method-level mapping

Together they form a complete address.

Syntax Examples
Class-Level Mapping

```
@RestController
@RequestMapping("/users")
```

```
public class UserController {
}
```

Method-Level Mapping

```
@GetMapping("/{id}")
public String getUser(@PathVariable int id) {
    return "User ID: " + id;
}
```

Final URL: /users/1

@RequestMapping Attributes

Attribute	Purpose
value / path	URL path
method	HTTP method
consumes	Request content type
produces	Response content type
params	Query parameters
headers	Header conditions

HTTP Method Example

```
@RequestMapping(value = "/save", method = RequestMethod.POST)
public String saveUser() {
    return "Saved";
}
```

Modern alternative: @PostMapping

Shortcut Annotations

- @GetMapping
- @PostMapping
- @PutMapping
- @PatchMapping
- @DeleteMapping

Internally all are @RequestMapping

Interview Questions & Answers

Q1. Can @RequestMapping be used on class and method?
A. Yes, on both.
Q2. Difference between @RequestMapping and @GetMapping?
A. @RequestMapping is generic, @GetMapping is specific to GET.
Q3. What happens if two methods have same mapping?
A. Application fails to start due to ambiguity.

Summary (Quick Revision)

@RequestMapping maps **URLs and HTTP methods** to controller logic and is the **foundation of request handling** in Spring MVC.

@GetMapping and @PostMapping (Spring MVC / Spring Boot)

Definition:
@GetMapping is used to **handle HTTP GET requests**, mainly to **retrieve data** from the server.

@PostMapping is used to **handle HTTP POST requests**, mainly to **send data to the server** for **creation or processing**.

Why We Use Them (Purpose)

- **@GetMapping**
  - To **fetch/read data**
  - Used in search, view, list operations
- **@PostMapping**
  - To **create/save data**
  - Used in form submission, data insertion

---

3. Key Points / Core Concepts

- Both are **shortcut annotations** of @RequestMapping
- Improve **code readability**
- Map specific HTTP methods
- Commonly used in **REST APIs**

---

4. Real-Life Example (Easy to Understand)

- **GET** → Reading a book
- **POST** → Writing notes in a book

Reading doesn't change data, writing does.

---

5. Syntax & Examples

@GetMapping Example

```
@GetMapping("/users")
public List<User> getAllUsers() {
    return userService.getUsers();
}
```

Retrieves data.

@PostMapping Example

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return userService.saveUser(user);
}
```

Sends data to server.

---

6. Key Differences (Important for Interview)

Feature	@GetMapping	@PostMapping
HTTP Method	GET	POST
Purpose	Fetch data	Send data
Request Body	Not recommended	Common
Data Visible	Yes (URL)	No (Body)
Idempotent	Yes	No

---

7. When to Use Which?

- Use **@GetMapping** when:
  - Data is not changing
  - Reading information
- Use **@PostMapping** when:
  - Creating new resource
  - Sending sensitive data

---

8. Interview Questions & Answers

Q1. Can we send request body in GET?
A. Not recommended and usually ignored.
Q2. Can @PostMapping return data?
A. Yes, it can return created object or response.
Q3. Are these mandatory in Spring Boot?
A. No, but they are best practice.

---

9. Summary (Quick Revision)

- @GetMapping → Read data
- @PostMapping → Create/send data
- Both simplify REST API development

@PutMapping and @DeleteMapping (Spring MVC / Spring Boot)

1. Definition (Simple & Interview-Friendly)
@PutMapping
@PutMapping is used to **handle HTTP PUT requests**, mainly to **update an existing resource** completely.
@DeleteMapping
@DeleteMapping is used to **handle HTTP DELETE requests**, mainly to **remove an existing resource**.

---

2. Why We Use Them (Purpose)

- **@PutMapping**
  - To **update existing data**
  - Used in **edit/update operations**
- **@DeleteMapping**
  - To **delete data**
  - Used in **remove operations**

---

3. Key Points / Core Concepts

- Both are **shortcut annotations** of @RequestMapping
- Used in **CRUD operations**
- Follow **REST standards**
- Mostly used with **@PathVariable**

---

4. Real-Life Example (Easy to Understand)

- **PUT** → Replacing old address in Aadhaar card
- **DELETE** → Removing a contact from phone

One updates, one removes.

---

5. Syntax & Practical Examples
@PutMapping Example

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

Updates user details.

@DeleteMapping Example

```
@DeleteMapping("/users/{id}")
```

```
public String deleteUser(@PathVariable int id) {
    userService.deleteUser(id);
    return "User deleted successfully";
}
```

Deletes user.

---

6. Key Differences (Important for Interview)

Feature	@PutMapping	@DeleteMapping
HTTP Method	PUT	DELETE
Purpose	Update resource	Delete resource
Request Body	Yes	No
Idempotent	Yes	Yes
CRUD Operation	Update	Delete

---

7. When to Use Which?

- Use **@PutMapping** when:
  - Updating full resource
  - Client sends complete object
- Use **@DeleteMapping** when:
  - Removing resource by ID

---

8. Interview Questions & Answers

Q1. Difference between PUT and POST?

A. PUT updates existing data, POST creates new data.

Q2. Can DELETE have request body?

A. Not recommended and rarely used.

Q3. Are PUT and DELETE idempotent?

A. Yes, calling multiple times gives same result.

---

9. Summary (Quick Revision)

- @PutMapping → Update
- @DeleteMapping → Delete
- Both follow RESTful best practices

@PathVariable and @RequestParam (Spring MVC / Spring Boot)

1. Definition (Simple & Interview-Friendly)

@PathVariable

@PathVariable is used to **extract values directly from the URL path.**

@RequestParam

@RequestParam is used to **extract values from query parameters** in the URL.

---

2. Why We Use Them (Purpose)

- **@PathVariable**
  - To identify a **specific resource**
  - Common in **RESTful URLs**
- **@RequestParam**
  - To pass **optional or filtering data**
  - Used in **search, pagination, sorting**

---

3. Key Points / Core Concepts

- Both read data from **URL**
- Used in **controller methods**
- @PathVariable is **mandatory by default**
- @RequestParam can be **optional**

---

4. Real-Life Example (Easy to Understand)

- **PathVariable** → House number
  /house/101
- **RequestParam** → Extra instructions
  /house?floor=2

One identifies, the other adds details.

---

5. Syntax & Practical Examples

@PathVariable Example

```
@GetMapping("/users/{id}")
public String getUser(@PathVariable int id) {
    return "User ID: " + id;
}
URL: /users/10
```

@RequestParam Example

```
@GetMapping("/users")
public String getUsers(@RequestParam String role) {
    return "Role: " + role;
}
URL: /users?role=admin
```

---

6. Optional Parameters Example

```
@GetMapping("/search")
public String search(@RequestParam(required = false) String keyword) {
    return "Keyword: " + keyword;
}
```

---

7. Key Differences (Very Important – Interview)

Feature	@PathVariable	@RequestParam
URL Type	Path	Query
Mandatory	Yes (default)	No
Usage	Resource identification	Filtering / options
REST Style	Strong	Weak

---

8. When to Use Which?

- Use **@PathVariable** when:
  - Identifying resource (/users/1)
- Use **@RequestParam** when:
  - Filtering/searching (?page=1)

---

9. Interview Questions & Answers

Q1. Can we use both together?

A. Yes, in same method.

Q2. Can @PathVariable be optional?

A. No (by default).

Q3. Difference between PathVariable and RequestParam?

A. PathVariable is part of URL path, RequestParam is query parameter.

---

10. Summary (Quick Revision)

- @PathVariable → URL path value
- @RequestParam → Query parameter
- Both help pass data to controllers

@RequestBody and @ResponseBody (Spring MVC / Spring Boot)

1. Definition (Simple & Interview-Friendly)

@RequestBody

@RequestBody is used to **read data from the HTTP request body** and **convert it into a Java object.**

@ResponseBody

@ResponseBody is used to **convert a Java object into HTTP response body** (usually JSON) and send it back to the client.

---

2. Why We Use Them (Purpose)

- **@RequestBody**
  - To receive **JSON/XML data** from client
  - Used mainly in **POST and PUT** requests
- **@ResponseBody**
  - To send **data (JSON/XML)** instead of views
  - Used in **REST APIs**

---

3. Key Points / Core Concepts

- Works with **HTTP message converters**
- Uses **Jackson** for JSON conversion
- @RestController already includes @ResponseBody
- Used at **method level**

---

4. Real-Life Example (Easy to Understand)

- **@RequestBody** → Receiving filled application form
- **@ResponseBody** → Sending confirmation slip

One reads input, one sends output.

---

5. Syntax & Practical Examples

@RequestBody Example

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return user;
}
```

JSON → Java Object

@ResponseBody Example

```
@GetMapping("/message")
@ResponseBody
public String message() {
    return "Hello World";
}
```

Java → JSON/String

---

6. Using @Controller vs @RestController

With @Controller

```
@Controller
public class DemoController {

    @GetMapping("/data")
    @ResponseBody
    public String data() {
        return "Data Response";
    }
}
```

With @RestController

```
@RestController
public class DemoController {

    @GetMapping("/data")
    public String data() {
        return "Data Response";
    }
}
```

@ResponseBody not needed.

---

7. Common Mistakes (Freshers)

- Forgetting @RequestBody for POST/PUT
- Using @ResponseBody with @RestController
- Sending wrong content-type header

---

8. Interview Questions & Answers

Q1. What is @RequestBody?

A. Used to read request body and convert it to Java object.

Q2. What is @ResponseBody?

A. Used to send Java object as HTTP response body.

Q3. Why not needed in @RestController?

A. Because it is already included internally.

---

9. Summary (Quick Revision)

- @RequestBody → Client → Server
- @ResponseBody → Server → Client
- Essential for REST APIs

Introduction to Thymeleaf

1. Definition (Simple & Interview-Friendly)

Thymeleaf is a **server-side Java template engine** used with Spring Boot to **build dynamic HTML pages** by embedding backend data into HTML.

---

2. Why We Use Thymeleaf (Purpose)

- To create **dynamic web pages**
- To integrate **Spring MVC with HTML**
- To replace JSP in modern Spring applications
- To keep **HTML readable even without server**

---

3. Key Points / Core Concepts

- Server-side rendering
- Works smoothly with **Spring Boot + MVC**
- Uses **natural templates** (valid HTML)
- Processes templates at **server**
- Uses special th:* attributes

---

4. Real-Life Example (Easy to Understand)

Think of a **printed form**:

- Blank form → HTML
- Filled form → Thymeleaf + data

Thymeleaf fills data into HTML before sending to browser.

---

5. Where Thymeleaf Fits in MVC

Controller → Model → Thymeleaf(View) → Browser

- Controller sends data
- Thymeleaf renders HTML
- Browser displays page

---

6. Basic Setup

Dependency

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

---

7. Basic Example

Controller

```
@Controller
public class HomeController {

    @GetMapping("/home")
    public String home(Model model) {
        model.addAttribute("name", "Abdul Gaffoor");
        return "home";
    }
}
```

home.html

```
<p th:text="${name}">Name</p>
```

Output: **Abdul Gaffoor**

---

8. Advantages

- Clean HTML
- Easy to learn for freshers
- Tight Spring integration
- Good for server-rendered apps

---

9. Interview Questions & Answers

Q1. What is Thymeleaf?

A. A server-side template engine for Spring applications.

Q2. Where are Thymeleaf templates stored?

A. src/main/resources/templates

Q3. Is Thymeleaf frontend or backend?

A. Backend (server-side).

---

10. Summary (Quick Revision)

Thymeleaf helps create **dynamic HTML pages** in Spring MVC by combining **backend data with HTML templates.**

Thymeleaf Configuration (Spring Boot)

1. Definition (Simple & Interview-Friendly)
Thymeleaf configuration defines **how Spring Boot finds, processes, and renders Thymeleaf HTML templates** in a web application.

---

2. Why We Configure Thymeleaf (Purpose)

- To tell Spring **where templates are located**
- To control **template suffix, prefix, and mode**
- To enable/disable **cache (dev vs prod)**
- To integrate smoothly with **Spring MVC**

---

3. Key Points / Core Concepts

- Auto-configured by **Spring Boot**
- Uses **application.properties**
- Templates stored in **/templates**
- Supports HTML, XHTML
- Caching enabled by default

---

4. Default Thymeleaf Configuration (Important)
Spring Boot automatically sets:

Property	Default Value
Prefix	classpath:/templates/
Suffix	.html
Mode	HTML
Encoding	UTF-8
Cache	true

No manual config needed for beginners.

---

5. application.properties Configuration

```
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.cache=false
```

cache=false is useful during development.

---

6. Real-Life Example (Easy to Understand)
Think of a **file explorer**:

- Folder → templates
- File type → .html
- Cache → memory copy for fast access

Thymeleaf config controls all this.

---

7. How Controller Connects to Thymeleaf

```
@Controller
public class PageController {
```

```
    @GetMapping("/login")
    public String loginPage() {
        return "login"; // login.html
    }
}
```

Thymeleaf resolves:
classpath:/templates/login.html

---

8. Common Mistakes (Freshers)

- Placing HTML outside /templates
- Returning full file name (login.html)
- Forgetting to disable cache in dev
- Using @RestController instead of @Controller

---

9. Interview Questions & Answers

Q1. Is Thymeleaf configuration mandatory?
A. No, Spring Boot auto-configures it.
Q2. Why disable Thymeleaf cache?
A. To see HTML changes without restarting server.
Q3. Which file controls Thymeleaf configuration?
A. application.properties or application.yml.

---

10. Summary (Quick Revision)

Thymeleaf configuration controls **template location, format, and behavior**, and Spring Boot handles most of it automatically.

Thymeleaf Syntax and Expressions

1. Definition (Simple & Interview-Friendly)

Thymeleaf syntax and expressions are used to **bind backend data to HTML, control page logic**, and **create dynamic web pages** in Spring MVC applications.

---

2. Why We Use Thymeleaf Syntax (Purpose)

- To display **dynamic data** in HTML
- To perform **conditions and loops**
- To bind **form data**
- To keep HTML **clean and readable**

---

3. Thymeleaf Attribute Syntax

Thymeleaf uses attributes starting with **th:**

```
<p th:text="${name}">Default Text</p>
```

Replaces content at runtime.

---

4. Types of Thymeleaf Expressions (Very Important)

Variable Expressions – ${...}

- Access **model attributes**

```
<p th:text="${user.name}"></p>
```

Selection Expressions – *{...}

- Used with th:object
- Shorter syntax

```
<form th:object="${user}">
    <input th:field="*{name}" />
</form>
```

Message Expressions – #{...}

- Used for **i18n (internationalization)**

```
<p th:text="#{label.name}"></p>
```

URL Expressions – @{...}

- Used to generate URLs

```
<a th:href="@{/home}">Home</a>
```

Fragment Expressions – ~{...}

- Used for reusable layouts

```
<div th:replace="fragments/header :: header"></div>
```

---

5. Common Thymeleaf Attributes

Attribute	Purpose
th:text	Set text
th:utext	Set unescaped text
th:each	Loop
th:if	Condition
th:unless	Opposite of if
th:href	Link
th:src	Image
th:value	Input value
th:field	Form binding

---

6. Loop Example (th:each)

```
<li th:each="user : ${users}"
    th:text="${user.name}">
</li>
```

Loops through users list.

---

7. Conditional Example

```
<p th:if="${user.active}">Active User</p>
<p th:unless="${user.active}">Inactive User</p>
```

---

8. Real-Life Example (Easy to Understand)

Think of Thymeleaf as **filling blanks in a form**:

- Blank → ${}
- Loop → repeating lines
- Condition → show/hide sections

---

9. Common Mistakes (Freshers)

- Forgetting ${} in expressions
- Using @RestController for Thymeleaf
- Not adding data to Model
- Syntax errors in th:*

---

10. Interview Questions & Answers

Q1. What is ${} in Thymeleaf?

A. Used to access model attributes.

Q2. Difference between ${} and *{}?
A. ${} accesses model, *{} works with form object.
Q3. What is th:each used for?
A. Looping over collections.

---

11. Summary (Quick Revision)
Thymeleaf expressions help create **dynamic HTML pages** using backend data, loops, and conditions in a clean way.

Thymeleaf Attributes – th:text, th:each, th:if

1. Definition (Simple & Interview-Friendly)
Thymeleaf attributes are **special HTML attributes** (th:*) used to **display data**, **iterate collections**, and **apply conditions** in server-side rendered pages.

---

2. Why We Use These Attributes (Purpose)

- To show **dynamic data** in HTML
- To loop through **lists or collections**
- To conditionally **show or hide content**
- To keep HTML **clean and readable**

---

3. th:text – Display Data
Purpose

- Replaces the **text content** of an HTML element

Example

```
<p th:text="${username}">Default Name</p>
```

Displays value of username
Key Point

- Escapes HTML for security (XSS safe)

---

4. th:each – Loop / Iterate
Purpose

- Used to **iterate over collections**

Example

```
<li th:each="user : ${users}"
    th:text="${user.name}">
</li>
```

Loops through users list
Syntax

```
variable : collection
```

---

5. th:if – Conditional Display
Purpose

- Displays element **only if condition is true**

Example

```
<p th:if="${user.active}">Active User</p>
```

Shown only when active = true

---

6. Combined Example (Very Important)

```
<ul>
  <li th:each="user : ${users}"
      th:if="${user.active}"
```

```
      th:text="${user.name}">
  </li>
</ul>
```
Shows **only active users**

---

7. Real-Life Example (Easy to Understand)
- th:text → Writing name on ID card
- th:each → Printing multiple ID cards
- th:if → Printing only valid ID cards

---

8. Common Mistakes (Freshers)
- Forgetting ${} in expressions
- Using th:if incorrectly with null values
- Not passing data from controller
- Syntax errors in loops

---

9. Interview Questions & Answers

Q1. What does th:text do?

A. Displays dynamic text.

Q2. What is the syntax of th:each?

A. item : collection

Q3. Difference between th:if and th:unless?

A. th:if shows when true, th:unless shows when false.

---

10. Summary (Quick Revision)
- th:text → Display data
- th:each → Loop data
- th:if → Conditional display

Thymeleaf Layouts and Fragments

1. Definition (Simple & Interview-Friendly)

Thymeleaf layouts and fragments are used to **reuse common HTML parts** (like header, footer, navbar) across multiple pages, helping create a **consistent layout** and **avoid code duplication**.

---

2. Why We Use Layouts & Fragments (Purpose)
- To **reuse common UI components**
- To maintain **consistent design**
- To reduce **duplicate HTML code**
- To make UI changes **easy and centralized**

---

3. Key Points / Core Concepts
- A **fragment** is a reusable part of HTML
- Layouts act as **master pages**
- Uses th:fragment, th:replace, th:include
- Common in **real-time projects**

---

4. Real-Life Example (Easy to Understand)

Think of a **letter format**:
- Header → Company name
- Body → Content

- Footer → Address

Same format, different content.

---

5. Creating a Fragment

header.html

```
<div th:fragment="header">
    <h1>My Application</h1>
</div>
```

---

6. Using Fragment in a Page

home.html

```
<div th:replace="fragments/header :: header"></div>
```

Replaces div with header fragment.

---

7. Footer Fragment Example

```
<div th:fragment="footer">
    <p>© 2026 MyApp</p>
</div>
```

Usage:

```
<div th:replace="fragments/footer :: footer"></div>
```

---

8. th:replace vs th:include (Interview Focus)

Attribute	Behavior
th:replace	Replaces entire tag
th:include	Includes content inside tag

---

9. Layout Structure Example

```
templates/
 ├── fragments/
 │    ├── header.html
 │    ├── footer.html
 ├── home.html
 ├── login.html
```

---

10. Common Mistakes (Freshers)

- Wrong fragment path
- Forgetting :: fragmentName
- Using @RestController
- Duplicating layout code

---

11. Interview Questions & Answers

Q1. What is a fragment in Thymeleaf?

A. A reusable HTML part.

Q2. Why use layouts?

A. To maintain consistent UI.

Q3. Difference between th:replace and th:include?

A. Replace removes tag, include keeps it.

---

12. Summary (Quick Revision)

Thymeleaf layouts and fragments help **reuse UI components**, making applications **clean, maintainable, and professional**.

Handling Form Submissions in Thymeleaf (Spring Boot + MVC)

1. Definition (Simple & Interview-Friendly)
Handling form submissions means **collecting user input from an HTML form, binding it to a Java object**, and **processing it in a Spring Controller.**

---

2. Why We Handle Forms (Purpose)

- To accept **user input** (login, registration, search)
- To send data from **UI → Backend**
- To perform **CRUD operations**
- To validate and store data

---

3. Flow of Form Submission (Very Important)

```
User fills form
        ↓
Thymeleaf Form
        ↓
Controller (@PostMapping)
        ↓
Service → DAO → DB
        ↓
Response View
```

---

4. Step 1: Model Class (Entity / DTO)

```
public class User {
    private String name;
    private String email;
    // getters & setters
}
```

---

5. Step 2: Controller – Show Form

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

---

6. Step 3: Thymeleaf Form

```
<form th:action="@{/register}" th:object="${user}" method="post">

    Name: <input type="text" th:field="*{name}" /><br>
    Email: <input type="email" th:field="*{email}" /><br>

    <button type="submit">Submit</button>
</form>
```

---

7. Step 4: Controller – Handle Submission

```
@PostMapping("/register")
public String submitForm(@ModelAttribute User user) {
    System.out.println(user.getName());
    System.out.println(user.getEmail());
    return "success";
}
```

---

8. Key Annotations Used (Interview Focus)

Annotation	Purpose
th:action	Form submit URL
th:object	Bind object
th:field	Bind field
@PostMapping	Handle form
@ModelAttribute	Receive data

---

9. Real-Life Example (Easy to Understand)
Like filling a **college admission form**:

- Form → Student details
- Submit → Office processing
- Response → Confirmation slip

---

10. Common Mistakes (Freshers)

- Forgetting th:object
- Mismatch between field name & model variable
- Using @RequestBody instead of @ModelAttribute
- Missing getters/setters

---

11. Interview Questions & Answers
Q1. How does Thymeleaf bind form data?
A. Using th:object and th:field.
Q2. Why use @ModelAttribute?
A. To bind form fields to object.
Q3. Difference between @RequestParam and @ModelAttribute?
A. @RequestParam → single field
@ModelAttribute → full object

---

12. Summary (Quick Revision)
Thymeleaf form handling allows **easy data binding** between UI and backend using Spring MVC.

Data Access with Spring Boot

1. Definition (Simple & Interview-Friendly)
Data access in Spring Boot means **connecting to a database, performing CRUD operations,** and **managing data persistence** using Spring-provided technologies like **JPA, Hibernate, and Repositories.**

---

2. Why Data Access Is Needed (Purpose)

- To **store application data** permanently
- To **retrieve, update, delete** data
- To separate **business logic from database logic**
- To make applications **scalable and maintainable**

---

3. Common Data Access Options in Spring Boot

Approach	Used For
JDBC	Low-level DB operations
Spring JDBC	Simplified JDBC

Approach	Used For
JPA (Hibernate)	ORM-based approach (most used)
Spring Data JPA	Auto CRUD + minimal code

Spring Data JPA is most preferred in real projects

---

4. Spring Data JPA (Core Concept)

What is JPA?

JPA (Java Persistence API) is a **specification** for ORM (Object Relational Mapping).

What is Hibernate?

Hibernate is the **implementation of JPA.**

---

5. Typical Data Access Flow

```
Controller
   ↓
Service
   ↓
Repository (DAO)
   ↓
Database
```

---

6. Entity Class (Maps Table to Object)

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

    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String email;
}
```

One object = one table row

---

7. Repository Interface (Very Important)

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

What JpaRepository Provides

- save()
- findById()
- findAll()
- deleteById()

No implementation needed!

---

8. Service Layer (Business Logic)

```
@Service
public class UserService {

    @Autowired
    private UserRepository repository;

    public User saveUser(User user) {
        return repository.save(user);
    }
```

```
}
```

---

9. Database Configuration (application.properties)

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=1234

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

---

10. Real-Life Example (Easy to Understand)

Think of a **bank system**:

- Entity → Account form
- Repository → Bank clerk
- DB → Bank vault

---

11. Common Mistakes (Freshers)

- Missing @Entity annotation
- Wrong table or column mapping
- Forgetting primary key
- Mixing controller and repository logic

---

12. Interview Questions & Answers

Q1. What is Spring Data JPA?
A. A framework that simplifies database operations using JPA.
Q2. Why use JpaRepository?
A. Provides built-in CRUD methods.
Q3. Difference between JPA and Hibernate?
A. JPA is specification, Hibernate is implementation.

---

13. Advantages of Spring Boot Data Access

- Less boilerplate code
- Easy DB configuration
- Auto transaction management
- Production-ready

---

14. Summary (Quick Revision)

Spring Boot makes data access **simple, clean, and powerful** using **Spring Data JPA**, helping developers focus on business logic instead of SQL.

REST Principles (Representational State Transfer)

1. Definition (Simple & Interview-Friendly)

REST is an **architectural style** used to design **scalable, stateless web services** that allow clients and servers to communicate using **standard HTTP methods**.

---

2. Why REST Is Used (Purpose)

- To build **lightweight and scalable APIs**
- To enable **client–server separation**
- To support **multiple clients** (web, mobile, third-party)
- To improve **performance and maintainability**

---

3. Core REST Principles (VERY IMPORTANT for Interview)

Client–Server Architecture

- Client (UI) and Server (API) are **separate**
- Both can evolve independently

Example:

Browser → REST API → Database

Statelessness

- Each request contains **all required information**
- Server does **not store client session**

Example:

Every request sends Authorization token

Uniform Interface

- Consistent way to access resources
- Uses **standard HTTP methods**

Method	Operation
GET	Read
POST	Create
PUT	Update
DELETE	Delete

Resource-Based

- Everything is treated as a **resource**
- Identified using **URI**

Example:

/users
/users/101

Representation of Resources

- Data exchanged in formats like:
  - JSON (most common)
  - XML

Example:

```
{
  "id": 101,
  "name": "Shiva"
}
```

Cacheability

- Responses can be **cached**
- Improves performance

Example:

Cache-Control: max-age=3600

Layered System

- Client doesn’t know if it talks directly to server
- Supports **load balancers, gateways**

---

4. RESTful API Design Example

Action	Endpoint	Method
Get all users	/users	GET
Get user by id	/users/1	GET

Action	Endpoint	Method
Create user	/users	POST
Update user	/users/1	PUT
Delete user	/users/1	DELETE

---

5. REST vs SOAP (Interview Favorite)

REST	SOAP
Lightweight	Heavy
JSON	XML
Stateless	Stateful
Easy to learn	Complex

---

6. Real-Life Example (Easy to Understand)

Think of **ATM machine**:

- Card → Authentication
- Button → Request
- Cash → Response

Each request is **independent.**

---

7. Common Mistakes (Freshers)

- Using verbs in URLs (/getUsers)
- Misusing HTTP methods
- Storing session on server
- Ignoring HTTP status codes

---

8. Important HTTP Status Codes

Code	Meaning
200	OK
201	Created
400	Bad Request
401	Unauthorized
404	Not Found
500	Server Error

---

9. Interview Questions & Answers

Q1. What does REST stand for?

A. Representational State Transfer

Q2. Is REST a protocol?

A. No, it’s an architectural style.

Q3. Why is REST stateless?

A. Improves scalability and performance.

---

10. Summary (Quick Revision)

REST principles help build **scalable, stateless, and maintainable APIs** using **standard HTTP methods and URIs.**

HTTP Methods (For REST APIs – Fresher & Interview Ready)

1. Definition (Simple & Interview-Friendly)
HTTP methods define **what action the client wants to perform** on a resource in a RESTful web service.

---

2. Why HTTP Methods Are Important (Purpose)
- To perform **CRUD operations**
- To follow **REST standards**
- To make APIs **clear and predictable**
- To improve **maintainability and scalability**

---

3. Most Common HTTP Methods (VERY IMPORTANT)
GET – Read Data
Purpose
- Fetch data from server
- Should **not change** server state

Example
```
GET /users
GET /users/1
```
Spring Boot Example
```
@GetMapping("/users")
public List<User> getUsers() {
    return userService.getAllUsers();
}
```
Key Points
- No request body
- Can be cached
- Safe method

POST – Create Data
Purpose
- Create a new resource

Example
```
POST /users
```
Spring Boot Example
```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return userService.save(user);
}
```
Key Points
- Has request body
- Not idempotent
- Returns 201 Created

PUT – Update (Full Update)
Purpose
- Update an **existing resource completely**

Example
```
PUT /users/1
```
Spring Boot Example
```
@PutMapping("/users/{id}")
public User updateUser(@PathVariable Long id,
                       @RequestBody User user) {
    return userService.update(id, user);
```

```
}
```

Key Points

- Replaces full resource
- Idempotent

DELETE – Remove Data

Purpose

- Delete a resource

Example

```
DELETE /users/1
```

Spring Boot Example

```
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable Long id) {
    userService.delete(id);
}
```

Key Points

- No request body
- Idempotent

---

4. Other HTTP Methods (Interview Awareness)

Method	Use
PATCH	Partial update
HEAD	Headers only
OPTIONS	Supported methods
TRACE	Debugging

---

5. HTTP Methods vs CRUD Mapping

CRUD	HTTP
Create	POST
Read	GET
Update	PUT / PATCH
Delete	DELETE

---

6. Real-Life Example (Easy to Understand)

Think of **library system**:

- GET → View books
- POST → Add new book
- PUT → Update book details
- DELETE → Remove book

---

7. Common Mistakes (Freshers)

- Using GET to save data
- Sending request body in GET
- Using POST for delete
- Wrong status codes

---

8. Important HTTP Status Codes (Quick View)

Code	Meaning
200	OK
201	Created
204	No Content

Code	Meaning
400	Bad Request
404	Not Found
500	Server Error

---

9. Interview Questions & Answers

Q1. Which HTTP method is idempotent?

A. GET, PUT, DELETE

Q2. Difference between PUT and PATCH?

A. PUT → Full update
PATCH → Partial update

Q3. Can GET have request body?

A. No (not recommended)

---

10. Summary (Quick Revision)

HTTP methods define **actions** on resources and are the **foundation of REST APIs.**

HTTP Methods: GET, POST, PUT, DELETE, PATCH

1. Definition (Simple & Interview-Friendly)

HTTP methods define **what operation is performed on a resource** in a RESTful application, such as **reading, creating, updating, or deleting data.**

---

2. Purpose of Using HTTP Methods

- To follow **REST principles**
- To map **CRUD operations** clearly
- To make APIs **understandable and predictable**
- To enable **proper client–server communication**

---

3. CRUD Mapping (Must Remember)

CRUD Operation	HTTP Method
Create	POST
Read	GET
Update (Full)	PUT
Update (Partial)	PATCH
Delete	DELETE

---

4. GET – Retrieve Data

Purpose

- Fetch data from server

Example URL

GET /users
GET /users/10

Spring Boot Example

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

Key Points

- No request body
- Safe & idempotent
- Can be cached

---

5. POST – Create New Resource

Purpose

- Create a new record

Example URL

POST /users

Spring Boot Example

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return userService.save(user);
}
```

Key Points

- Has request body
- Not idempotent
- Returns 201 Created

---

6. PUT – Full Update

Purpose

- Update **entire resource**

Example URL

PUT /users/10

Spring Boot Example

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

Key Points

- Replaces full object
- Idempotent

---

7. DELETE – Remove Resource

Purpose

- Delete a record

Example URL

DELETE /users/10

Spring Boot Example

```
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable Long id) {
    userService.delete(id);
}
```

Key Points

- No request body
- Idempotent

---

8. PATCH – Partial Update

Purpose

- Update **specific fields only**

Example URL

PATCH /users/10

Spring Boot Example

```
@PatchMapping("/users/{id}")
```

```
public User updateEmail(@PathVariable Long id,
                        @RequestParam String email) {
    return userService.updateEmail(id, email);
}
```

Key Points

- Partial update
- Efficient
- Not always idempotent

---

9. PUT vs PATCH (Interview Favorite)

PUT	PATCH
Full update	Partial update
Replaces resource	Modifies fields
Idempotent	May not be idempotent

---

10. Real-Life Example (Easy to Understand)

Profile Management App

- GET → View profile
- POST → Create profile
- PUT → Update full profile
- PATCH → Change mobile number
- DELETE → Remove account

---

11. Common Mistakes (Freshers)

- Using POST for update
- Using GET to modify data
- Ignoring PATCH
- Wrong HTTP status codes

---

12. Interview Questions & Answers

Q1. Which methods are idempotent?

A. GET, PUT, DELETE

Q2. Can PATCH replace PUT?

A. No, PATCH is partial update.

Q3. Which method is safest?

A. GET

---

13. Summary (Quick Revision)

HTTP methods define **what action is performed on a resource**, and are the **foundation of RESTful APIs**.

HTTP Status Codes (REST APIs – Fresher & Interview Ready)

1. Definition (Simple & Interview-Friendly)

HTTP status codes are **standard response codes** sent by the server to indicate **the result of a client's request**.

---

2. Why HTTP Status Codes Are Important (Purpose)

- To tell client **success or failure** of request
- To enable **proper error handling**
- To follow **REST standards**

- To improve **debugging and communication**

---

3. Status Code Categories (Must Remember)

Range	Meaning
1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client Error
5xx	Server Error

---

4. Commonly Used Status Codes (VERY IMPORTANT)

2xx – Success Codes

Code	Meaning	When Used
200	OK	Successful GET/PUT
201	Created	Resource created (POST)
204	No Content	Delete success

4xx – Client Errors

Code	Meaning	When Used
400	Bad Request	Invalid input
401	Unauthorized	Authentication required
403	Forbidden	Access denied
404	Not Found	Resource not found
409	Conflict	Duplicate resource

5xx – Server Errors

Code	Meaning	When Used
500	Internal Server Error	Unexpected failure
502	Bad Gateway	Invalid upstream response
503	Service Unavailable	Server down

---

5. Spring Boot Example

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {

    User user = userService.findById(id);

    if (user == null) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }

    return ResponseEntity.ok(user);
}
```

---

6. Real-Life Example (Easy to Understand)

Online Order

- 200 → Order details fetched
- 201 → Order placed
- 400 → Wrong order data

- 404 → Order not found
- 500 → Server issue

---

7. Common Mistakes (Freshers)
- Always returning 200
- Ignoring error codes
- Not using ResponseEntity
- Exposing internal errors

---

8. Interview Questions & Answers
Q1. Difference between 200 and 201?
A. 200 → Success
201 → Resource created
Q2. When to use 204?
A. Successful delete with no response body
Q3. What is 404?
A. Resource not found

---

9. Status Codes vs HTTP Methods (Quick View)

Method	Success Code
GET	200
POST	201
PUT	200
DELETE	204

---

10. Summary (Quick Revision)
HTTP status codes clearly indicate **request outcomes**, helping clients understand **success, failure, or errors** in REST APIs.

HTTP Status Codes: 200, 201, 400, 404, 500

1. What Are HTTP Status Codes? (Definition)
HTTP status codes are **standard response codes** sent by the server to indicate **whether a client request was successful or failed**.

---

2. Why These Status Codes Are Important? (Purpose)
- To clearly tell **result of request**
- To help **client-side handling**
- To follow **REST API standards**
- To improve **debugging**

---

3. 200 – OK
Meaning
- Request processed successfully

When Used
- Successful **GET**
- Successful **PUT**

Example
GET /users/1 → 200 OK
Spring Boot Example
```
return ResponseEntity.ok(user);
```

---

4. 201 – Created

Meaning

- New resource created successfully

When Used

- Successful **POST**

Example

POST /users → 201 Created

Spring Boot Example

```
return ResponseEntity.status(HttpStatus.CREATED).body(user);
```

---

5. 400 – Bad Request

Meaning

- Client sent **invalid or incorrect data**

When Used

- Validation errors
- Wrong request format

Example

POST /users (missing field) → 400 Bad Request

Spring Boot Example

```
return ResponseEntity.badRequest().build();
```

---

6. 404 – Not Found

Meaning

- Requested resource **does not exist**

When Used

- Invalid ID
- Wrong URL

Example

GET /users/99 → 404 Not Found

Spring Boot Example

```
return ResponseEntity.notFound().build();
```

---

7. 500 – Internal Server Error

Meaning

- Server failed to process request

When Used

- NullPointerException
- Database down

Example

GET /users → 500 Internal Server Error

Spring Boot Example

```
return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
```

---

8. Real-Life Example (Easy to Understand)

Bank Application

- 200 → Balance fetched
- 201 → Account created
- 400 → Invalid account details
- 404 → Account not found
- 500 → Server issue

---

9. Common Fresher Mistakes

- Always returning 200
- Not handling errors

- Exposing stack trace
- Wrong status code usage

---

10. Interview Questions & Answers

Q1. Difference between 200 and 201?

A. 200 → Request success
201 → Resource created

Q2. When do you use 400?

A. Client sends invalid data

Q3. What does 500 mean?

A. Server-side error

---

11. Quick Summary (One-Liner)

- **200** → Success
- **201** → Created
- **400** → Client error
- **404** → Not found
- **500** → Server error

ResponseEntity (Spring Boot – REST APIs)

1. Definition (Simple & Clear)

ResponseEntity is a Spring class used to **represent the full HTTP response**, including:

- **Response body**
- **HTTP status code**
- **HTTP headers**

---

2. Why We Use ResponseEntity (Purpose)

- To **control HTTP status codes**
- To send **custom headers**
- To return **proper REST responses**
- To follow **REST API best practices**

Without ResponseEntity, Spring returns only data, **not status control**.

---

3. What ResponseEntity Contains

Part	Description
Body	Data sent to client
Status	HTTP status code
Headers	Metadata (optional)

---

4. Basic Syntax

```
ResponseEntity<T>
```

T → Type of response body

---

5. Simple Example (200 OK)

```
@GetMapping("/user/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    User user = userService.getById(id);
    return ResponseEntity.ok(user);
}
```

Returns **data + 200 OK**

6. Returning 201 Created

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User user) {
    User saved = userService.save(user);
    return ResponseEntity.status(HttpStatus.CREATED).body(saved);
}
```

---

7. Returning 404 Not Found

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {

    User user = userService.getById(id);

    if (user == null) {
        return ResponseEntity.notFound().build();
    }

    return ResponseEntity.ok(user);
}
```

---

8. Returning 400 Bad Request

```
return ResponseEntity.badRequest().build();
```

---

9. Returning 500 Internal Server Error

```
return ResponseEntity
        .status(HttpStatus.INTERNAL_SERVER_ERROR)
        .build();
```

---

10. Adding HTTP Headers

```
HttpHeaders headers = new HttpHeaders();
headers.add("source", "spring-boot");

return new ResponseEntity<>(user, headers, HttpStatus.OK);
```

---

11. ResponseEntity vs @ResponseBody (Interview Question)

ResponseEntity	@ResponseBody
Controls status & headers	Returns only body
Flexible	Limited
REST best practice	Basic usage

---

12. Real-Life Example (Easy to Understand)

Courier Package

- Body → Item
- Status → Delivered / Failed
- Headers → Tracking info

---

13. Common Fresher Mistakes

- Always returning 200
- Not using ResponseEntity in REST APIs
- Ignoring proper status codes
- Returning null instead of status

---

14. Interview Questions & Answers

Q1. Why use ResponseEntity?
A. To control HTTP response completely.
Q2. Can ResponseEntity return only status?
A. Yes, using .build().
Q3. Is ResponseEntity mandatory?
A. No, but recommended for REST APIs.

---

15. One-Line Summary (Quick Revision)
ResponseEntity allows sending **data + status + headers** in REST responses.

@ExceptionHandler (Spring Boot – REST & MVC)

1. Definition (Simple & Clear)
@ExceptionHandler is a Spring annotation used to **handle exceptions in a controlled way** inside a controller or a global exception handler.

---

2. Why We Use @ExceptionHandler (Purpose)
- To avoid **application crash**
- To return **proper HTTP status codes**
- To send **meaningful error messages**
- To separate **error handling logic** from business logic

---

3. Where We Can Use @ExceptionHandler
Option 1: Inside a Controller (Local Handling)
- Handles exceptions **only for that controller**

Option 2: Inside @ControllerAdvice (Global Handling)
- Handles exceptions **for entire application** (Best Practice)

---

4. Basic Syntax
```
@ExceptionHandler(ExceptionType.class)
public ResponseEntity<?> handleException(Exception ex) {
    return new ResponseEntity<>(message, status);
}
```

---

5. Example: Local Exception Handling
```
@RestController
public class UserController {

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
        if (id == 0) {
            throw new IllegalArgumentException("Invalid ID");
        }
        return new User();
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleIllegalArg(IllegalArgumentException ex) {
        return ResponseEntity
                .badRequest()
                .body(ex.getMessage());
    }
```

```
}
```
Handles exception **only in this controller**

---

6. Global Exception Handling (Recommended)
Using @ControllerAdvice

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return ResponseEntity
                .status(HttpStatus.NOT_FOUND)
                .body(ex.getMessage());
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGeneric(Exception ex) {
        return ResponseEntity
                .status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body("Something went wrong");
    }
}
```

---

7. Common Exceptions Mapped to Status Codes

Exception	Status
IllegalArgumentException	400
ResourceNotFoundException	404
NullPointerException	500
Exception	500

---

8. Real-Life Example (Easy to Understand)
ATM Machine

- Invalid PIN → Handled gracefully
- Server issue → Error message
- Machine doesn’t crash

---

9. @ExceptionHandler vs try-catch (Interview Focus)

@ExceptionHandler	try-catch
Centralized handling	Repeated code
Clean controllers	Messy code
REST-friendly	Not scalable

---

10. Common Fresher Mistakes

- Using try-catch everywhere
- Not returning proper status code
- Exposing stack trace
- Not using global handler

---

11. Interview Questions & Answers
Q1. What is @ExceptionHandler used for?
A. To handle exceptions and return custom responses.

Q2. Difference between @ExceptionHandler and @ControllerAdvice?
A. @ExceptionHandler handles exception,
@ControllerAdvice makes it global.
Q3. Can we handle multiple exceptions?
A. Yes, by adding multiple methods.

---

12. One-Line Summary (Quick Revision)
@ExceptionHandler helps handle errors **gracefully and consistently** in Spring applications.

@ControllerAdvice (Spring Boot – Global Exception Handling)

1. Definition (Simple & Clear)
@ControllerAdvice is a Spring annotation used to **handle exceptions globally** across all controllers in an application.

---

2. Why We Use @ControllerAdvice (Purpose)

- To **centralize exception handling**
- To avoid writing **duplicate error code** in controllers
- To return **consistent error responses**
- To keep controllers **clean and readable**

---

3. How @ControllerAdvice Works

```
Controller throws exception
        ↓
@ControllerAdvice catches it
        ↓
@ExceptionHandler method executes
        ↓
Proper HTTP response returned
```

---

4. Basic Structure

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception ex) {
        return ResponseEntity
                .status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body("Something went wrong");
    }
}
```

---

5. Handling Custom Exception (Very Important)
Custom Exception

```
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String msg) {
        super(msg);
    }
}
```

Global Handler

```
@ControllerAdvice
```

```
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return ResponseEntity
                .status(HttpStatus.NOT_FOUND)
                .body(ex.getMessage());
    }
}
```

---

6. Handling Multiple Exceptions

```
@ExceptionHandler({
    IllegalArgumentException.class,
    NullPointerException.class
})
public ResponseEntity<String> handleMultiple(Exception ex) {
    return ResponseEntity
            .badRequest()
            .body(ex.getMessage());
}
```

---

7. @ControllerAdvice vs @RestControllerAdvice (Interview Focus)

@ControllerAdvice	**@RestControllerAdvice**
Used for MVC & REST	Used only for REST
Need @ResponseBody	Auto JSON response

- **Use @RestControllerAdvice for REST APIs**

---

8. Real-Life Example (Easy to Understand)

Hospital Help Desk

- Any issue → Go to help desk
- One place → Handles all problems

---

9. Common Fresher Mistakes

- Not using global handler
- Returning only String messages
- Not mapping correct status codes
- Exposing internal error details

---

10. Interview Questions & Answers

Q1. Why use @ControllerAdvice?

A. For global exception handling.

Q2. Difference between @ControllerAdvice and @ExceptionHandler?

A. @ControllerAdvice is global,
@ExceptionHandler handles specific exception.

Q3. Can @ControllerAdvice handle validation errors?

A. Yes.

---

11. One-Line Summary (Quick Revision)

@ControllerAdvice provides **centralized and consistent exception handling** across the entire Spring application.

Validation in Spring Boot (@Valid, @NotNull)

1. Definition (Simple & Clear)
Validation in Spring Boot is used to **check user input data** and ensure it follows **defined rules** before processing or saving it.

---

2. Why Validation Is Needed (Purpose)

- To prevent **invalid data** entering the system
- To avoid **runtime errors**
- To ensure **data consistency**
- To give **clear error messages** to users

---

3. What is @Valid?
Definition
@Valid is used to **trigger validation** on an object based on validation annotations present in that object.
Where Used

- Controller method parameters

Example

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody User user) {
    return ResponseEntity.status(HttpStatus.CREATED).body(user);
}
```

- @Valid tells Spring: *"Validate this object before using it."*

---

4. What is @NotNull?
Definition
@NotNull ensures that a field **must not be null**.
Example

```
public class User {

    @NotNull(message = "Name cannot be null")
    private String name;
}
```

- If name is null → validation fails.

---

5. Common Validation Annotations (Interview Must-Know)

Annotation	Purpose
@NotNull	Value must not be null
@NotEmpty	Not null + not empty
@NotBlank	Not null + not blank
@Size	Length constraint
@Min / @Max	Number limits
@Email	Valid email format
@Pattern	Regex validation

---

6. Complete Validation Flow

```
Client Request
      ↓
@Valid checks input
      ↓
Validation fails?
      ↓
```

```
@ControllerAdvice handles error
      ↓
Error response sent
```

---

7. Handling Validation Errors (Important)
ControllerAdvice Example

```
@RestControllerAdvice
public class ValidationHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<String> handleValidation(
            MethodArgumentNotValidException ex) {

        String error = ex.getBindingResult()
                         .getFieldError()
                         .getDefaultMessage();

        return ResponseEntity.badRequest().body(error);
    }
}
```

---

8. Real-Life Example (Easy to Understand)
Online Registration Form

- Name empty → Error shown
- Email invalid → Error shown
- Valid input → Saved successfully

---

9. Common Fresher Mistakes

- Forgetting @Valid in controller
- Using @NotNull for empty strings
- Not handling validation exceptions
- Missing getters/setters

---

10. Interview Questions & Answers
Q1. What does @Valid do?
A. Triggers validation rules.
Q2. Difference between @NotNull and @NotBlank?
A. @NotNull → not null
@NotBlank → not null + not empty + not whitespace
Q3. Where do you write validation rules?
A. In model/entity class.

---

11. One-Line Summary (Quick Revision)
@Valid triggers validation and @NotNull ensures required fields are not null, helping keep data **clean and safe**.

Authentication vs Authorization *(Security Basics)*

1. Definition (Simple & Clear)
Authentication

- **Who are you?**
  Process of **verifying the identity** of a user.

Authorization

- **What are you allowed to do?**
  Process of **checking permissions** after authentication.

---

2. Why Both Are Needed (Purpose)

- To protect **application resources**
- To ensure **right user gets right access**
- To prevent **unauthorized actions**
- Core concept of **application security**

---

3. Order of Execution (Very Important)

Authentication → Authorization

- Authorization cannot happen without authentication.

---

4. Authentication Explained

What Happens

- User provides credentials
- System verifies them

Examples

- Username & Password
- OTP
- Token (JWT)
- Biometrics

Real-Life Example

Showing ID card at office gate

---

5. Authorization Explained

What Happens

- System checks user role/permission

Examples

- Admin can delete users
- User can view profile
- Manager can approve requests

Real-Life Example

Access to specific office rooms

---

6. Authentication vs Authorization (Interview Table)

Authentication	Authorization
Who are you?	What can you do?
Login process	Permission check
Done first	Done after auth
Username, password	Roles, privileges

---

7. Spring Boot Example (Basic Idea)

User logs in → Authenticated
Check role → Authorized
Example:

```
@PreAuthorize("hasRole('ADMIN')")
@DeleteMapping("/users/{id}")
public void deleteUser() { }
```

---

8. Common Fresher Mistakes

- Thinking both are same
- Skipping authorization

- Hardcoding roles
- Exposing secured APIs

---

9. Interview Questions & Answers

Q1. Can authorization happen without authentication?

A. No.

Q2. Which one comes first?

A. Authentication.

Q3. Example of authorization failure?

A. Logged-in user tries admin-only action.

---

10. One-Line Summary (Quick Revision)

- **Authentication** → Verifies identity
- **Authorization** → Verifies access rights

UserDetailsService (Spring Security)

1. Definition (Simple & Clear)

UserDetailsService is a **core Spring Security interface** used to **load user-specific data** (username, password, roles) during **authentication.**

- In short: **Spring Security uses UserDetailsService to verify users at login time.**

---

2. Why We Use UserDetailsService (Purpose)

- To **fetch user details from database**
- To integrate **custom user tables** with Spring Security
- To perform **authentication**
- To support **role-based authorization**

---

3. Where UserDetailsService Fits (Flow)

```
Login Request
      ↓
AuthenticationManager
      ↓
UserDetailsService.loadUserByUsername()
      ↓
UserDetails returned
      ↓
Password check
      ↓
User authenticated
```

---

4. UserDetailsService Interface

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException;
}
```

---

5. What is UserDetails? (Related Concept)

UserDetails represents **authenticated user information**:

- Username
- Password
- Roles / Authorities

- Account status

---

6. Custom UserDetailsService Implementation

```
@Service
public class CustomUserDetailsService
        implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username)
            .orElseThrow(() ->
                new UsernameNotFoundException("User not found"));

        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                user.getAuthorities()
        );
    }
}
```

---

7. User Entity Example

```
@Entity
public class User {

    private String username;
    private String password;
    private String role;
}
```

---

8. How Spring Security Uses It (Important)

- Spring Security **automatically calls** loadUserByUsername()
- Compares:
  - Entered password
  - Stored (encrypted) password
- If valid → authentication success

---

9. Real-Life Example (Easy to Understand)

Bank Login

- You give account number
- Bank fetches your details
- Verifies password
- Grants access

---

10. Common Fresher Mistakes

- Not implementing UserDetailsService
- Returning null instead of UserDetails
- Storing passwords without encryption
- Confusing User entity with UserDetails

---

11. Interview Questions & Answers

Q1. What is UserDetailsService used for?

A. To load user data during authentication.

Q2. When is loadUserByUsername() called?

A. During login/authentication.

Q3. Can we have multiple UserDetailsService?

A. Yes, but one must be primary.

---

12. UserDetailsService vs UserDetails

UserDetailsService	UserDetails
Loads user	Holds user data
Interface	Interface
Called during login	Used for auth check

---

13. One-Line Summary (Quick Revision)

UserDetailsService connects **Spring Security authentication** with **database user data.**

@SpringBootTest (Spring Boot Testing)

1. Definition (Simple & Clear)

@SpringBootTest is a Spring Boot annotation used to **load the complete application context** for **integration testing.**

- In simple words: it starts the **entire Spring Boot application** just like real runtime, but **for tests.**

---

2. Why We Use @SpringBootTest (Purpose)

- To test **full application flow**
- To test **integration between layers** (Controller → Service → Repository)
- To verify **configuration & beans**
- To simulate **real production behavior**

---

3. What @SpringBootTest Does Internally

- Loads **ApplicationContext**
- Applies **auto-configuration**
- Scans **all beans**
- Can start **embedded server**

---

4. Basic Usage Example

```
@SpringBootTest
class UserServiceTest {

    @Autowired
    private UserService userService;

    @Test
    void testCreateUser() {
        User user = new User("Shiva");
        User saved = userService.save(user);
        assertNotNull(saved);
    }
}
```

5. When to Use @SpringBootTest

- Integration tests
- End-to-end tests
- Not recommended for small unit tests

---

6. @SpringBootTest vs @WebMvcTest (Interview Focus)

@SpringBootTest	@WebMvcTest
Loads full context	Loads only web layer
Slow	Fast
Integration testing	Controller testing
Real DB (optional)	Mocked beans

---

7. Web Environment Options

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
```

Option	Meaning
MOCK	Default (no server)
RANDOM_PORT	Random embedded server
DEFINED_PORT	Uses server.port

---

8. Testing REST APIs Example

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class UserControllerTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void testGetUsers() {
        ResponseEntity<String> response =
            restTemplate.getForEntity("/users", String.class);

        assertEquals(HttpStatus.OK, response.getStatusCode());
    }
}
```

---

9. Real-Life Example (Easy to Understand)

Building Inspection

- Check whole building, not just one room
- Ensures everything works together

---

10. Common Fresher Mistakes

- Using @SpringBootTest for every test
- Slow test execution
- Not isolating unit tests
- Forgetting @Test annotation

---

11. Interview Questions & Answers

Q1. What is @SpringBootTest used for?

A. Integration testing.

Q2. Does it load full application context?

A. Yes.

Q3. Is @SpringBootTest slow? Why?
A. Yes, because it loads all beans.

---

12. One-Line Summary (Quick Revision)
@SpringBootTest is used for **full-context integration testing** in Spring Boot.

Microservices Basics

1. Definition (Simple & Clear)
A **microservice** is a **small, independent, and loosely coupled service** that performs **one specific business function** and communicates with other services over a network, usually HTTP/REST or messaging queues.

- In simple terms: **Break a big application into small services.**

---

2. Why Microservices Are Needed (Purpose)

- **Scalability** → Scale only the required service
- **Flexibility** → Use different tech stacks per service
- **Faster deployment** → Independent release of services
- **Resilience** → Failure in one service does not crash the whole app
- **Team organization** → Teams own services independently

---

3. Monolithic vs Microservices (Interview Favorite)

Feature	Monolithic	Microservices
Application Size	Single large app	Small independent services
Deployment	Single	Individual services
Scalability	Entire app	Service-level
Technology Stack	Single	Can vary per service
Development Speed	Slower	Faster, independent teams

---

4. Key Principles of Microservices

1. **Single Responsibility** – Each service has one business capability
2. **Independent Deployment** – Can deploy/update separately
3. **Decentralized Data Management** – Each service has its own database
4. **Communication** – Lightweight protocols (HTTP/REST, gRPC, Kafka)
5. **Resilience** – Service failure doesn’t crash system

---

5. Components in Microservices Architecture

Component	Purpose
Service	Business logic for one domain
API Gateway	Single entry point for clients
Database	Each service can have its own DB
Service Registry	Keeps track of all services (Eureka, Consul)
Messaging	Async communication (Kafka, RabbitMQ)
Load Balancer	Distributes requests evenly

---

6. Communication Between Services
Synchronous

- REST API calls between services
- Easy but slower if one service fails

Asynchronous

- Messaging systems like Kafka, RabbitMQ
- Decouples services, more resilient

---

7. Real-Life Example (Easy to Understand)
E-commerce Application

- User Service → Handles accounts
- Product Service → Manages products
- Order Service → Processes orders
- Payment Service → Handles payments

Each service is **independent** but communicates as needed.

---

8. Advantages of Microservices

- Faster development & deployment
- Independent scaling
- Easy to maintain & test
- Better fault isolation

---

9. Challenges

- Complexity in communication
- Data consistency issues
- Deployment management
- Service monitoring & logging

---

10. Interview Questions & Answers
Q1. What is a microservice?
A. Small, independent service handling one business function.
Q2. Difference between Monolith and Microservice?
A. Monolith = single large app; Microservice = multiple small apps.
Q3. How do microservices communicate?
A. Synchronously via REST/gRPC, or asynchronously via messaging queues.
Q4. Why choose microservices?
A. Scalability, independent deployment, resilience, faster development.

---

11. One-Line Summary (Quick Revision)
Microservices break a big application into **small, independent, and scalable services**, improving flexibility and maintainability.

Microservices: Building Blocks & Data Management

1. Microservices Building Blocks
A microservice application is made up of **several core components**. Each building block serves a specific purpose:

Building Block	Purpose	Example / Notes
Service	Encapsulates a **single business functionality**	User Service, Order Service
API / REST Endpoints	Allows communication with clients	/users, /orders
Database	Stores service-specific data	Each service has **its own DB**
Service Registry	Keeps track of available services	Eureka, Consul

Building Block	Purpose	Example / Notes
API Gateway	Single entry point for clients	Zuul, Spring Cloud Gateway
Load Balancer	Distributes requests to multiple instances	Ribbon, Nginx
Messaging / Event Bus	Enables async communication	Kafka, RabbitMQ
Configuration Server	Centralized config management	Spring Cloud Config

Key Point: Each microservice is **independent, loosely coupled, and deployable separately**.

---

2. Data Management in Microservices

Unlike monolithic apps, **each microservice manages its own data**:

1. **Database per Service**
   - Every service has a **separate database**
   - Avoids tight coupling between services

   Example:
   - User Service → MySQL
   - Order Service → PostgreSQL
2. **Types of Data**
   - **Transactional Data:** Local to the service (CRUD operations)
   - **Shared Data / Event Data:** Published for other services to consume
3. **Communication for Shared Data**
   - **Synchronous:** REST API calls
   - **Asynchronous:** Messaging systems (Kafka, RabbitMQ)

Patterns for Data Management

Pattern	Description
Database per Service	Each service has its own DB (recommended)
Shared Database	Multiple services use same DB (not recommended)
Event Sourcing	Changes stored as events; services consume events
CQRS (Command Query Responsibility Segregation)	Separate read & write models for scalability

---

3. Real-Life Example (Easy to Understand)

E-commerce Microservices

Service	DB	Notes
User Service	MySQL	Stores user accounts
Product Service	MongoDB	Stores product catalog
Order Service	PostgreSQL	Stores order history
Payment Service	MySQL	Processes payments

- User Service → publishes events (new user)
- Order Service → subscribes to events to update analytics
- Each service manages its **own data independently**

---

4. Advantages

- Independent scaling of services and DB
- Data ownership clearly defined
- Fault isolation (failure in one DB doesn’t crash all services)

---

5. Challenges

- Data consistency across services
- Distributed transactions are complex
- Maintaining eventual consistency

---

6. Interview Questions & Answers

Q1. How is data managed in microservices?

A. Each service has its own database (Database per Service pattern).

Q2. Can microservices share a database?

A. Technically yes, but it’s not recommended due to tight coupling.

Q3. What is event sourcing?

A. Storing changes as events for other services to consume asynchronously.

Q4. Why is database per service recommended?

A. Ensures **loose coupling**, **independent scaling**, and **resilience**.

---

7. One-Line Summary (Quick Revision)

Microservices consist of **independent services with their own databases**, communicating via APIs or messaging, ensuring **scalability, fault isolation, and loose coupling**.
