**Phase 1: Introduction & Setup**

1. **What is Spring Boot & Why Use It?**
   * Features & benefits over traditional Spring
   * How Spring Boot simplifies development
   * Real-world use cases
2. **Setting Up a Spring Boot Project**
   * Using Spring Initializr (online & IDE-based)
   * Project structure walkthrough
   * Running a basic Spring Boot application

### ****Phase 1: Introduction & Setup****

#### **1. What is Spring Boot & Why Use It?**

* **Introduction to Spring Boot**
  + What is Spring Boot?
  + How it simplifies traditional Spring framework development
  + Key features: Auto-configuration, Starter dependencies, Embedded servers
  + Where Spring Boot is used (Real-world examples)
* **Spring Boot vs Traditional Spring**
  + Manual configuration in Spring vs. Auto-configuration in Spring Boot
  + Comparison of a simple application in Spring vs. Spring Boot

#### **2. Setting Up a Spring Boot Project**

* **Using Spring Initializr**
  + Generate a Spring Boot project using [start.spring.io](https://start.spring.io/)
  + Selecting dependencies (Spring Web, Spring Boot DevTools, etc.)
  + Downloading and importing into an IDE (IntelliJ/Eclipse/VS Code)
* **Project Structure Overview**
  + Understanding src/main/java and src/main/resources
  + application.properties vs application.yml
  + The @SpringBootApplication annotation
* **Running a Basic Spring Boot Application**
  + Using main() method in the Application class
  + Running with embedded Tomcat server
  + Checking logs and understanding startup messages

### ****Phase 1: Introduction & Setup (With Code & Exercises)****

## **1. What is Spring Boot & Why Use It?**

### ****What is Spring Boot?****

Spring Boot is a framework that simplifies Java application development by:

* Removing the need for complex XML configurations
* Providing built-in defaults (Auto-configuration)
* Embedding an application server (Tomcat, Jetty, etc.)
* Offering production-ready features (Monitoring, Security, Logging)

### ****Spring Boot vs Traditional Spring****

**Example: A simple REST API in Spring (without Spring Boot)**

* Requires configuring XML files or Java-based configurations.

<!-- Spring XML Configuration -->

<bean id="myService" class="com.example.MyService"/>

**Same REST API in Spring Boot**

* No XML configuration needed, just annotations!

@RestController

@RequestMapping("/hello")

public class HelloController {

@GetMapping

public String sayHello() {

return "Hello, Spring Boot!";

}

}

## **2. Setting Up a Spring Boot Project**

### ****Creating a Project Using Spring Initializr****

Go to [**start.spring.io**](https://start.spring.io/) and configure:

* **Project:** Maven
* **Language:** Java
* **Spring Boot Version:** Latest stable version
* **Dependencies:**
  + Spring Web (for building REST APIs)
  + Spring Boot DevTools (for live reload)
* Click **Generate** and download the ZIP file.
* Extract and open in **IntelliJ IDEA / Eclipse / VS Code**

### ****Creating a Project Using VS CODE****

You can start a Spring Boot project directly in **VS Code** using the **Spring Boot Extension Pack** and **Spring Initializr**. Here’s how:

**Step 1: Install Required Extensions**

1. Open **VS Code**.
2. Go to **Extensions** (Ctrl + Shift + X).
3. Install the following:
   * **Spring Boot Extension Pack** (includes Spring Initializr, Spring Boot support, and more).
   * **Java Extension Pack** (if not installed, to enable Java support).
   * **Sping Boot Dashboard**(optional)

**Step 2: Create a New Spring Boot Project**

1. Open **Command Palette** (Ctrl + Shift + P).
2. Search for **"Spring Initializr: Create a Maven Project"**.
3. Follow the prompts:
   * **Build System**: Choose **Maven** or **Gradle**.
   * **Language**: Choose **Java**.
   * **Spring Boot Version**: Select a stable version.
   * **Group Id**: Example: com.example
   * **Artifact Id**: Example: demo
   * **Packaging**: Choose jar (or war if needed).
   * **Java Version**: Choose 17 or 21 (LTS versions are recommended).
   * **Dependencies**: Add necessary dependencies like **Spring Web**, **Spring Boot DevTools**, etc.
4. Choose a folder to save the project.

**Step 3: Open the Project in VS Code**

Once the project is created:

1. Open the project in **VS Code** (File > Open Folder).
2. It will prompt you to install the recommended extensions if not installed—accept the suggestions.

**Step 4: Run the Spring Boot Application**

1. Open **src/main/java/com/example/demo/DemoApplication.java**.
2. Click on **Run** (Ctrl + F5) OR use the terminal:
   * Open the **Integrated Terminal** (Ctrl + ~).
   * Run:
   * ./mvnw spring-boot:run # For Maven
   * ./gradlew bootRun # For Gradle

**Step 5: Verify the Application**

Once the application starts, it will display a log with:

Tomcat started on port 8080

Started DemoApplication in X seconds

* Open **http://localhost:8080** in a browser (if you added a controller).

**Additional Tips**

* Use **Live Server** to auto-reload changes (Ctrl + Shift + P → "Live Reload: Start Server").
* Add **Spring Boot DevTools** for hot-reloading (pom.xml or build.gradle).
* Use **Spring Boot Dashboard** (Ctrl + Shift + P → "Spring Boot Dashboard") to manage multiple applications.

## **Project Structure Overview**

When you open the project, you will see:

my-springboot-app

│── src/main/java/com/example/demo

│ ├── DemoApplication.java <-- Main class

│ ├── controller/HelloController.java <-- Create this

│

│── src/main/resources

│ ├── application.properties <-- Configuration file

│

│── pom.xml <-- Dependencies and Build tool (Maven)

### ****Understanding**** @SpringBootApplication

In DemoApplication.java, you will see:

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

* @SpringBootApplication = Enables auto-configuration and component scanning.
* SpringApplication.run(...) starts the application with an embedded Tomcat server.

## **3. Running a Basic Spring Boot Application**

### ****Step 1: Create a Simple REST API****

Inside src/main/java/com/example/demo/controller/, create HelloController.java:

package com.example.demo.controller;

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

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

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

@RestController

@RequestMapping("/hello")

public class HelloController {

@GetMapping

public String sayHello() {

return "Hello, Spring Boot!";

}

}

### ****Step 2: Run the Application****

* Open DemoApplication.java and run the main() method.
* The application will start on **port 8080** (default).
* Open a browser or Postman and visit:

http://localhost:8080/hello

* Expected Output:

Hello, Spring Boot!

## **Exercises for Students**

### ****Exercise 1: Modify the Response****

* Modify the sayHello() method to return "Welcome to Spring Boot!".
* Restart the application and check the output in the browser/Postman.

### ****Exercise 2: Create a New API Endpoint****

* Create a new REST controller GreetController.java.
* Add an endpoint /greet that returns "Good Morning!".
* Test the endpoint in the browser/Postman.

### ****Additional Exercises & Explanations for Phase 1****

## **Exercise 3: Using Path Variables in API**

Modify HelloController.java to accept a **name** as a path variable.

### ****Task:****

* Update the /hello API to accept a name (e.g., /hello/John)
* Return "Hello, John!" instead of "Hello, Spring Boot!"

### ****Solution:****

@GetMapping("/{name}")

public String sayHello(@PathVariable String name) {

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

}

### ****Test the API:****

URL:

http://localhost:8080/hello/Alex

Expected Output:

Hello, Alex!

## **Exercise 4: Using Request Parameters**

Modify HelloController.java to accept a **name** as a query parameter.

### ****Task:****

* Create an endpoint /greet
* Accept a request parameter name (e.g., /greet?name=Emma)
* Return "Hello, Emma!"

### ****Solution:****

@GetMapping("/greet")

public String greet(@RequestParam String name) {

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

}

### ****Test the API:****

URL:

http://localhost:8080/greet?name=Emma

Expected Output:

Hello, Emma!

## **Exercise 5: Returning JSON Instead of Plain Text**

So far, the API is returning simple text. Now, let’s return **JSON data** instead.

### ****Task:****

* Modify /hello to return JSON with name and message.

### ****Solution:****

@RestController

@RequestMapping("/hello")

public class HelloController {

@GetMapping("/{name}")

public Map<String, String> sayHello(@PathVariable String name) {

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

response.put("name", name);

response.put("message", "Hello, " + name + "!");

return response;

}

}

### ****Test the API:****

URL:

http://localhost:8080/hello/John

Expected Output (JSON format):

{

"name": "John",

"message": "Hello, John!"

}

## **Exercise 6: Create a Student API**

**Task:**

* Create a new controller StudentController.java
* Create an endpoint /student that returns a JSON object with:
  + id (Integer)
  + name (String)
  + age (Integer)

### ****Solution:****

@RestController

@RequestMapping("/student")

public class StudentController {

@GetMapping

public Map<String, Object> getStudent() {

Map<String, Object> student = new HashMap<>();

student.put("id", 101);

student.put("name", "Dineshkumar");

student.put("age", 34);

return student;

}

}

### ****Test the API:****

URL:

http://localhost:8080/student

Expected Output (JSON format):

{

"id": 101,

"name": "Alice",

"age": 22

}

## **Additional Topics for Explanation (If Needed)**

### ****1. Understanding**** @RestController ****vs.**** @Controller

| **Annotation** | **Purpose** |
| --- | --- |
| @RestController | Used for REST APIs, returns JSON |
| @Controller | Used for MVC-based applications, returns HTML views |

Since we are building REST APIs, we always use @RestController.

### ****2. What Happens Internally When We Start a Spring Boot App?****

* The main() method runs SpringApplication.run(), which:
  + Creates a Spring ApplicationContext
  + Scans for components (@Component, @Service, @Repository)
  + Auto-configures dependencies (like Embedded Tomcat for web apps)

### ****Additional Topics for Phase 1****

These topics will give your students a **deeper understanding** of how Spring Boot works while keeping things practical.

## **1. What Happens Internally When We Start a Spring Boot App?**

When we run SpringApplication.run(DemoApplication.class, args);, Spring Boot does the following:

1. **Starts an ApplicationContext (IoC Container)**
   * Scans for components (@Component, @Service, @Repository)
   * Creates beans and injects dependencies
2. **Auto-configures Default Settings**
   * Configures embedded Tomcat for web apps
   * Sets default database connections (if Spring Data is used)
   * Configures default properties for logging, security, etc.
3. **Runs the Embedded Server (Tomcat/Jetty)**
   * Unlike traditional Spring apps, we **don’t need** to deploy manually in an external Tomcat server.

### ****How to Check What’s Auto-Configured?****

Run this command to see all auto-configured beans:

mvn spring-boot:run --debug

It helps in debugging configuration issues.

## **2. Understanding** @RestController **vs.** @Controller

Spring Boot allows two ways to create web controllers:

| **Annotation** | **Purpose** |
| --- | --- |
| @RestController | Used for REST APIs, returns JSON |
| @Controller | Used for MVC-based applications, returns HTML views |

### ****Example:**** @RestController ****(Returns JSON)****

@RestController

@RequestMapping("/api")

public class ApiController {

@GetMapping("/message")

public Map<String, String> getMessage() {

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

response.put("message", "Hello, this is a REST API!");

return response;

}

}

**Test:**  
http://localhost:8080/api/message  
**Output (JSON):**

{

"message": "Hello, this is a REST API!"

}

### ****Example:**** @Controller ****(Returns HTML View)****

@Controller

public class WebController {

@GetMapping("/home")

public String homePage() {

return "redirect:/index.html"; // Redirects to static HTML file

}

}

In this case, **Spring Boot looks for an index.html file** in the src/main/resources/static/ folder.

## **3. What is** application.properties **and Why Use It?**

Spring Boot provides a configuration file **application.properties** (or application.yml) to store settings like:

* Server port
* Database configuration
* Logging settings

### ****Example: Change the Server Port****

Add this to src/main/resources/application.properties:

server.port=9090

Restart the app, and it will now run on:

http://localhost:9090

### ****Example: Custom Message in Properties****

Define a property:

app.welcome.message=Welcome to My Spring Boot App!

Use it in Java:

@Value("${app.welcome.message}")

private String message;

**Exercise:** Ask students to add their own properties and retrieve them in a controller.

## **4. What Are Spring Boot Starters?**

Spring Boot simplifies dependencies using **starter packs**. These starters automatically add the necessary libraries to your project.

### ****Common Spring Boot Starters****

| **Starter** | **Purpose** |
| --- | --- |
| spring-boot-starter-web | Build REST APIs |
| spring-boot-starter-data-jpa | Connect to relational databases |
| spring-boot-starter-security | Add authentication & security |
| spring-boot-starter-test | Testing framework |

**Example:**  
In pom.xml, instead of manually adding dependencies for Spring MVC, Jackson, and Embedded Tomcat, we **just add one**:

<dependency>

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

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

</dependency>

This automatically includes all necessary dependencies.

## **5. What is Spring Boot DevTools?**

Spring Boot DevTools helps developers by enabling:  
✅ **Automatic Restart** – Restarts the app on code changes.  
✅ **LiveReload** – Refreshes the browser automatically.  
✅ **Enhanced Logging** – Provides better error messages.

### ****How to Enable DevTools?****

Add this dependency in pom.xml:

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

Now, when you **change Java code** and save, Spring Boot **automatically restarts** the app.

**Exercise:** Ask students to change a controller response and see if the change reflects without manually restarting.

## **6. Logging in Spring Boot**

Spring Boot uses **SLF4J + Logback** for logging.

### ****Using Logs Instead of**** System.out.println()

Replace System.out.println() with **SLF4J Logging**:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

@RestController

@RequestMapping("/log")

public class LoggingController {

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

@GetMapping

public String logExample() {

logger.info("INFO: This is an info log message!");

logger.warn("WARN: This is a warning log message!");

logger.error("ERROR: This is an error log message!");

return "Check the logs in the console!";

}

}

**Exercise:** Ask students to add different log levels in their controllers and observe the output.

## **7. Understanding Embedded Tomcat in Spring Boot**

Unlike traditional Java web apps, **Spring Boot does NOT require an external Tomcat server**. It comes with an **embedded Tomcat** by default.

### ****How to Change the Default Port?****

Modify application.properties:

server.port=8085

### ****How to Disable Embedded Tomcat?****

If using Spring Boot just as a **backend service (without a web app)**, disable the embedded Tomcat server:

spring.main.web-application-type=none

## **Final Recap for Phase 1**

By now, students should:  
✅ Understand what Spring Boot is and how it simplifies development.  
✅ Be able to create and run a Spring Boot project.  
✅ Understand controllers, REST APIs, and returning JSON responses.  
✅ Know how to configure application.properties for custom settings.  
✅ Use Spring Boot DevTools for faster development.  
✅ Implement logging in their applications.  
✅ Know about embedded Tomcat and how Spring Boot handles web applications.

### ****🛠 Practical Examples 🚀****

### ****User\_Crud operation: refer github link****

**Phase 2: Core Spring Concepts (Taught Within Spring Boot)**

1. **Dependency Injection & Beans**
   * Explain @Component, @Service, @Repository
   * Using @Autowired for Dependency Injection
   * Bean lifecycle & @PostConstruct, @PreDestroy (briefly)
2. **Spring Boot Configuration & Properties**
   * application.properties vs application.yml
   * Custom properties with @Value
   * Environment-specific configurations

### ****Phase 2: Core Spring Concepts****

## **📌 Topics in Phase 2**

1️⃣ **Spring IoC (Inversion of Control) & Dependency Injection**

* @Component, @Service, @Repository, @Controller
* @Autowired, Constructor vs Setter Injection

2️⃣ **Spring Bean Lifecycle & Scopes**

* @Bean, @Scope (Singleton, Prototype)
* InitializingBean & DisposableBean

3️⃣ **Spring AOP (Aspect-Oriented Programming)**

* @Aspect, @Before, @After, @Around
* Logging, Performance Monitoring

4️⃣ **Spring Transactions & Exception Handling**

* @Transactional for database operations
* @ExceptionHandler, @ControllerAdvice for global exception handling

5️⃣ **Spring Profiles & Configuration**

* @Profile for environment-specific beans
* Externalizing configuration (application.properties, YAML)

### ****🔹 Step 1: Understanding IoC & Dependency Injection****

Spring uses **IoC (Inversion of Control)** to manage object creation and dependency injection.

#### ✅ **Example: Without Spring (Manual Dependency Injection)**

class EmailService {

public void sendEmail(String message) {

System.out.println("Email sent: " + message);

}

}

class UserService {

private EmailService emailService = new EmailService(); // Manual object creation

public void registerUser(String name) {

System.out.println(name + " registered successfully!");

emailService.sendEmail("Welcome, " + name);

}

}

public class Main {

public static void main(String[] args) {

UserService userService = new UserService();

userService.registerUser("Alice");

}

}

💡 **Problem:** Tight coupling! We manually create objects (new EmailService()), making it hard to test and maintain.

#### ✅ **Example: Using Spring IoC & Dependency Injection**

import org.springframework.stereotype.Service;

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

@Service

class EmailService {

public void sendEmail(String message) {

System.out.println("Email sent: " + message);

}

}

@Service

class UserService {

private final EmailService emailService;

@Autowired

public UserService(EmailService emailService) { // Constructor Injection

this.emailService = emailService;

}

public void registerUser(String name) {

System.out.println(name + " registered successfully!");

emailService.sendEmail("Welcome, " + name);

}

}

💡 **Spring handles object creation** and injects EmailService into UserService.

### ****📌 Spring IoC (Inversion of Control) & Dependency Injection - Theoretical Explanation****

#### ✅ **What is IoC (Inversion of Control)?**

* IoC is a **design principle** where the control of object creation and dependency management is transferred from the **developer** to the **Spring framework**.
* Instead of manually creating objects using new, Spring **manages object lifecycle** and provides instances when needed.

💡 **Example (Without IoC):**

* Traditional Java approach where objects are created manually:

class EmailService {

void sendEmail(String message) {

System.out.println("Email sent: " + message);

}

}

class UserService {

private EmailService emailService = new EmailService(); // Manual object creation

void registerUser(String name) {

emailService.sendEmail("Welcome, " + name);

}

}

* + **Problem:** UserService is **tightly coupled** with EmailService because it directly creates an instance (new EmailService()).
  + **Difficult to test:** If we want to replace EmailService with SMSService, we have to change the UserService class.

#### ✅ **How Spring IoC Solves This?**

* Instead of manually creating objects, Spring **injects** them using **Dependency Injection (DI)**.
* IoC **manages** dependencies automatically, making code more flexible, testable, and maintainable.

## **🔹 Dependency Injection (DI)**

**DI is a technique where objects (dependencies) are provided to a class instead of creating them manually.**  
Spring supports three types of DI:

1️⃣ **Constructor Injection** (Recommended)  
2️⃣ **Setter Injection**  
3️⃣ **Field Injection** (Not recommended for testing)

### ****🔹 Spring Bean: The Foundation of IoC****

* A **Bean** is an object managed by the Spring container.
* Defined using **annotations** like @Component, @Service, @Repository, etc.
* The Spring container **creates, manages, and injects beans** where required.

### ****✅ Summary****

|  |  |
| --- | --- |
| Concept | Description |
| **IoC (Inversion of Control)** | Spring takes control of object creation and management. |
| **Bean** | An object managed by the Spring IoC container. |
| **Dependency Injection (DI)** | Injecting dependencies into a class instead of creating them manually. |
| **Annotations for Beans** | @Component, @Service, @Repository, @Controller |
| **Types of DI** | Constructor Injection, Setter Injection, Field Injection |

### ****🛠 Practical Examples 🚀****

### ****User\_Registration operation: refer github link****

### ****🔹 Step 2: Spring Bean Lifecycle & Scopes****

### ****📌 Spring Bean Lifecycle – Theoretical Explanation****

In Spring, a **bean's lifecycle** refers to the **various stages it goes through** from creation to destruction. Understanding this helps you **manage resources efficiently** and execute logic at specific points (e.g., initializing connections, releasing resources).

## **1️⃣ Stages of Spring Bean Lifecycle**

|  |  |
| --- | --- |
| Stage | Description |
| **1. Instantiation** | Spring **creates** the bean using the constructor. |
| **2. Populating Properties** | Spring **injects dependencies** via constructor/setter injection. |
| **3. Bean Post-Processing (Before Initialization)** | Custom logic before initialization (using BeanPostProcessor). |
| **4. Initialization** | Spring calls @PostConstruct, afterPropertiesSet(), or custom init methods. |
| **5. Bean Post-Processing (After Initialization)** | Custom logic after initialization (using BeanPostProcessor). |
| **6. Ready to Use** | The bean is **fully initialized** and ready for use. |
| **7. Destruction** | Spring calls @PreDestroy, destroy(), or custom destroy methods before removing the bean. |

## **2️⃣ Lifecycle Methods in Spring**

### ****✅ 1. Using**** InitializingBean ****&**** DisposableBean ****(Interface-Based)****

Spring provides two interfaces:

* InitializingBean → Implements afterPropertiesSet()
* DisposableBean → Implements destroy()

Example:

@Component

public class MyBean implements InitializingBean, DisposableBean {

public MyBean() {

System.out.println("1️⃣ Constructor: Bean is created.");

}

@Override

public void afterPropertiesSet() {

System.out.println("4️⃣ afterPropertiesSet: Bean is initialized.");

}

@Override

public void destroy() {

System.out.println("7️⃣ destroy: Bean is being destroyed.");

}

}

### ****✅ 2. Using**** @PostConstruct ****&**** @PreDestroy ****(Annotation-Based)****

More modern approach:

@Component

public class MyBean {

public MyBean() {

System.out.println("1️⃣ Constructor: Bean is created.");

}

@PostConstruct

public void init() {

System.out.println("4️⃣ @PostConstruct: Bean is initialized.");

}

@PreDestroy

public void destroy() {

System.out.println("7️⃣ @PreDestroy: Bean is being destroyed.");

}

}

### ****✅ 3. Using**** @Bean(initMethod, destroyMethod) ****(XML-Free Approach)****

If defining beans in a configuration class:

@Configuration

public class AppConfig {

@Bean(initMethod = "customInit", destroyMethod = "customDestroy")

public MyBean myBean() {

return new MyBean();

}

}

public class MyBean {

public void customInit() {

System.out.println("4️⃣ customInit: Bean is initialized.");

}

public void customDestroy() {

System.out.println("7️⃣ customDestroy: Bean is being destroyed.");

}

}

## **3️⃣ Summary**

|  |  |  |
| --- | --- | --- |
| Approach | Initialization Method | Destruction Method |
| **Interface-Based** | afterPropertiesSet() | destroy() |
| ****Annotation-Based**** | **@PostConstruct** | **@PreDestroy** |
| **Config-Based** | initMethod | destroyMethod |

## **Summary: Why is Lifecycle Useful in Real Time?**

|  |  |
| --- | --- |
| Use Case | Lifecycle Method Used |
| **Initialize a database connection** when the app starts | @PostConstruct |
| **Load configurations** (e.g., reading properties) | @PostConstruct |
| **Load data into cache** for fast access | @PostConstruct |
| **Release resources (DB connections, caches, threads)** before shutting down | @PreDestroy |

### ****Final Summary****

### Apart from **DB connections, threads, and caching**, the **Spring Bean Lifecycle** is used in **real-time applications** for various important tasks.

| **Use Case** | **@PostConstruct** | **@PreDestroy** |
| --- | --- | --- |
| ✅ **API Key & Configurations** | Load API keys on startup | Clear sensitive data on shutdown |
| ✅ **Message Queue Connection** | Connect to Kafka/RabbitMQ | Disconnect cleanly |
| ✅ **Service Registration (Eureka, Consul)** | Register microservice | Deregister before shutdown |
| ✅ **File Resource Management** | Open log/report files | Close file handles properly |
| ✅ **AI/ML Model Loading** | Load AI models into memory | Free memory on shutdown |

### ****🔍 Next Steps – Practical Example****

**Check github: Employee Management System.**

**Spring Bean Scopes**! 🚀

## **🔹 What is Bean Scope in Spring?**

When you create a bean in Spring, **Spring manages its lifecycle and scope**.  
**Scope** defines **how many instances** of a bean are created and **how long they exist**.

## **🔹 Types of Bean Scopes in Spring**

|  |  |  |
| --- | --- | --- |
| **Scope** | **Description** | **Usage** |
| **singleton** (default) | One instance per Spring container | Common for stateless beans (services, repositories) |
| **prototype** | A new instance every time it's requested | Useful for stateful beans (e.g., user sessions) |
| **request** (Web) | One instance per HTTP request | Used in web applications (Spring MVC) |
| **session** (Web) | One instance per user session | Stores user data across multiple requests |
| **application** (Web) | One instance per web application | Used for shared application-wide data |
| **websocket** (Web) | One instance per WebSocket session | Used for WebSocket communication |

## **1️⃣ Singleton Scope (Default)**

🔹 **Only one instance is created** in the entire Spring container.  
🔹 Even if multiple requests come, the same instance is returned.

💡 **Example: A Singleton Service**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Service;

@Service

@Scope("singleton") // Default scope, can be omitted

public class SingletonService {

public SingletonService() {

System.out.println("🔹 SingletonService Bean Created");

}

}

✅ **Usage:** Common for **stateless beans** like services, repositories.

## **2️⃣ Prototype Scope**

🔹 **A new instance is created every time** the bean is requested.  
🔹 **Spring does not manage the full lifecycle** of prototype beans (no @PreDestroy).

💡 **Example: Prototype Bean**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Service;

@Service

@Scope("prototype")

public class PrototypeService {

public PrototypeService() {

System.out.println("🔹 PrototypeService Bean Created");

}

}

✅ **Usage:** Useful when you need a **new object every time** (e.g., user requests, temporary data).

## **3️⃣ Request Scope (For Web Apps)**

🔹 **A new instance is created for each HTTP request**.  
🔹 Used when we want to **store request-specific data**.

💡 **Example: Request Scoped Bean**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import org.springframework.web.context.annotation.RequestScope;

@Component

@RequestScope

public class RequestScopedBean {

public RequestScopedBean() {

System.out.println("🔹 RequestScopedBean Created for a New HTTP Request");

}

}

✅ **Usage:** Used for **handling request-specific data** (e.g., request logs, request counters).

## **4️⃣ Session Scope (For Web Apps)**

🔹 **A new instance is created for each user session**.  
🔹 Useful when **storing user-specific data** across multiple requests.

💡 **Example: Session Scoped Bean**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import org.springframework.web.context.annotation.SessionScope;

@Component

@SessionScope

public class SessionScopedBean {

public SessionScopedBean() {

System.out.println("🔹 SessionScopedBean Created for a New User Session");

}

}

✅ **Usage:** Storing user **authentication info**, **shopping cart** data.

## **5️⃣ Application Scope (For Web Apps)**

🔹 **One instance per entire web application**.  
🔹 Used when **sharing global application-wide data**.

💡 **Example: Application Scoped Bean**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import org.springframework.web.context.annotation.ApplicationScope;

@Component

@ApplicationScope

public class ApplicationScopedBean {

public ApplicationScopedBean() {

System.out.println("🔹 ApplicationScopedBean Created for the Whole App");

}

}

✅ **Usage:** Used for **storing global configuration settings**.

## **6️⃣ WebSocket Scope (For Web Apps)**

🔹 **One instance per WebSocket session**.  
🔹 Used in **real-time messaging apps (e.g., chat apps, stock tickers)**.

💡 **Example: WebSocket Scoped Bean**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import org.springframework.web.socket.config.annotation.EnableWebSocket;

@Component

@Scope("websocket")

public class WebSocketScopedBean {

public WebSocketScopedBean() {

System.out.println("🔹 WebSocketScopedBean Created for a WebSocket Session");

}

}

✅ **Usage:** Managing **real-time user sessions in chat applications**.

## **🎯 Which Scope Should You Use?**

|  |  |  |
| --- | --- | --- |
| **Use Case** | **Scope** | **Why?** |
| ✅ **Global shared services** | singleton | One instance across the app |
| ✅ **New object every request** | prototype | Each call gets a new instance |
| ✅ **Store request-specific data** | request | One instance per HTTP request |
| ✅ **Store user-specific session data** | session | One instance per user session |
| ✅ **Application-wide shared data** | application | One instance for the app |
| ✅ **WebSocket real-time messaging** | websocket | One instance per WebSocket session |

## **🚀 Next Steps**

A **practical example** to compare singleton vs prototype

### ****Refer GitHub: Assigning Tasks to Employees & Tracking Completion****

#### **📌 Scenario:**

A company has an **EmployeeTaskManagerService (Singleton)** that assigns **unique tasks (Prototype Bean)** to employees.  
Each employee can **complete tasks**, and we keep track of **assigned and completed tasks**.

## **Key Takeaways**

|  |  |
| --- | --- |
| **Feature** | **Implementation** |
| **Singleton Bean** | EmployeeTaskManagerService (only 1 instance) |
| **Prototype Bean** | Task (new instance for every assignment) |
| **Tracking Tasks** | assignedTasks for pending, completedTasks for completed |
| **Endpoint Control** | API handles employee registration, task assignment, and completion |

### ****🔹 Step 3: Spring AOP (Aspect-Oriented Programming)****

AOP allows adding behavior like **logging** or **performance tracking** **without modifying the actual code**.

#### ✅ **Example: Logging Aspect**

import org.aspectj.lang.annotation.\*;

import org.springframework.stereotype.Component;

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.service.\*.\*(..))")

public void logBefore() {

System.out.println("Executing method...");

}

@After("execution(\* com.example.service.\*.\*(..))")

public void logAfter() {

System.out.println("Method execution completed!");

}

}

💡 This logs every method execution in com.example.service **without modifying the service class**.

### ****🔹 Step 4: Transaction Management****

@Transactional ensures database operations are atomic.

#### ✅ **Example: Handling Transactions**

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class BankService {

@Transactional

public void transferMoney(String fromAccount, String toAccount, double amount) {

// Deduct money from sender

// Add money to receiver

// If error occurs, rollback transaction

}

}

💡 **If any error occurs, the transaction rolls back automatically.**

### ****🔹 Step 5: Exception Handling in Spring****

Global exception handling using @ControllerAdvice.

#### ✅ **Example: Custom Exception Handling**

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

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(RuntimeException.class)

public String handleRuntimeException(RuntimeException ex) {

return "Error: " + ex.getMessage();

}

}

💡 This ensures all runtime exceptions return a friendly error message.

**Phase 3: Building a REST API with Spring Boot**

1. **Spring MVC in Spring Boot**
   * @RestController, @RequestMapping, @GetMapping, @PostMapping
   * Handling request parameters and path variables
   * Using @RequestBody & @ResponseBody
2. **Exception Handling & Validation**
   * Custom exception handling with @ControllerAdvice
   * Validation using @Valid and @NotNull, @Size

**Phase 4: Database Integration & JPA**

1. **Spring Boot with Databases**
   * Configuring H2/MySQL/PostgreSQL/MongoDB
   * Using Spring Data JPA (@Entity, @Id, @GeneratedValue)
   * Repository layer with JpaRepository
2. **Service Layer & Business Logic**
   * Best practices for @Service layer
   * Transaction management with @Transactional

# **Step 1: Spring Boot & MongoDB Integration**

We'll break this step into **three main parts**:

1️⃣ **Setting up MongoDB with Spring Boot**  
2️⃣ **Configuring application.properties**  
3️⃣ **Using MongoRepository for basic operations**

## **1️⃣ Setting Up MongoDB with Spring Boot**

### **Prerequisites**

🔹 **MongoDB installed** (locally or cloud-based like MongoDB Atlas)  
🔹 **Spring Boot project setup** (Spring Initializr)

### **Project Setup (Spring Initializr)**

Go to [Spring Initializr](https://start.spring.io/) and select:  
✅ **Spring Boot Version:** Latest stable version  
✅ **Dependencies:**

* Spring Web
* Spring DevTools
* Spring Data MongoDB
* Spring Boot Starter Data MongoDB
* Lombok (optional for reducing boilerplate code)

Download the project and open it in **IntelliJ IDEA / VS Code / Eclipse**.

## **2️⃣ Configuring** application.properties

Inside src/main/resources/application.properties, configure MongoDB:

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=mydatabase

If using **MongoDB Atlas**, replace with:

spring.data.mongodb.uri=mongodb+srv://<username>:<password>@cluster0.mongodb.net/mydatabase?retryWrites=true&w=majority

## **3️⃣ Using** MongoRepository **for Basic Operations**

# **Creating a Model Class**

Create a package com.example.model and add a class:

package com.example.model;

import lombok.Data;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

@Data

@Document(collection = "users") // Collection name in MongoDB

public class User {

@Id

private String id;

private String name;

private String email;

}

# **Creating a Repository Interface**

Inside com.example.repository:

package com.example.repository;

import com.example.model.User;

import org.springframework.data.mongodb.repository.MongoRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface UserRepository extends MongoRepository<User, String> {

}

· MongoRepository<User, String> provides built-in CRUD operations.

· findByEmail(String email): A custom query method to fetch users by email.

# **Service Layer (**UserService.java**)**

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User saveUser(User user) {

return userRepository.save(user);

}

public Optional<User> getUserByEmail(String email) {

return userRepository.findByEmail(email);

}

}

* @Service: Marks this as a service layer.
* userRepository.findAll(): Fetches all users from MongoDB.
* userRepository.save(user): Saves a user document in MongoDB.
* findByEmail(email): Fetches a user by email.

# **6. REST Controller (**UserController.java**)**

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping

public List<User> getAllUsers() {

return userService.getAllUsers();

}

@PostMapping

public User saveUser(@RequestBody User user) {

return userService.saveUser(user);

}

@GetMapping("/{email}")

public ResponseEntity<User> getUserByEmail(@PathVariable String email) {

return userService.getUserByEmail(email)

.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.notFound().build());

}

}

* @RestController: Marks this class as a REST API controller.
* @RequestMapping("/users"): Base URL for all user-related APIs.
* @GetMapping: Fetches all users.
* @PostMapping: Saves a new user in MongoDB.
* @GetMapping("/{email}"): Fetches a user by email.

# **Testing the API**

#### **1. Save a User (POST)**

POST http://localhost:8080/users

Content-Type: application/json

{

"name": "John Doe",

"email": "john@example.com"

}

#### **2. Get All Users (GET)**

GET http://localhost:8080/users

#### **3. Get User by Email (GET)**

GET http://localhost:8080/users/john@example.com

### **Summary**

✅ Connected Spring Boot to MongoDB Atlas  
✅ Created a User model mapped to MongoDB  
✅ Implemented a repository with MongoRepository  
✅ Built REST APIs to manage user data

CURD OPERATIONS

The MongoRepository<User, String> interface in **Spring Data MongoDB** provides several built-in methods for performing CRUD operations and queries on the MongoDB collection.

## **🛠️ Built-in Methods in** MongoRepository

The MongoRepository extends CrudRepository and PagingAndSortingRepository, so it inherits a variety of useful methods.

### **1️⃣ Basic CRUD Operations**

|  |  |
| --- | --- |
| Method | Description |
| save(S entity) | Saves or updates a document in the database. |
| findById(ID id) | Retrieves a document by its ID. |
| existsById(ID id) | Checks if a document exists by ID. |
| findAll() | Retrieves all documents from the collection. |
| findAllById(Iterable<ID> ids) | Retrieves multiple documents by their IDs. |
| count() | Returns the total count of documents in the collection. |
| deleteById(ID id) | Deletes a document by its ID. |
| delete(T entity) | Deletes a specific document. |
| deleteAll(Iterable<? extends T> entities) | Deletes multiple documents. |
| deleteAll() | Deletes all documents from the collection. |

### **2️⃣ Sorting & Pagination (From** PagingAndSortingRepository**)**

|  |  |
| --- | --- |
| Method | Description |
| findAll(Sort sort) | Retrieves all documents and sorts them based on a given field. |
| findAll(Pageable pageable) | Retrieves documents with pagination. |

👉 **Example Usage for Sorting**

List<User> users = userRepository.findAll(Sort.by("personal.name").ascending());

👉 **Example Usage for Pagination**

Pageable pageable = PageRequest.of(0, 5); // Page 0, 5 records per page

Page<User> usersPage = userRepository.findAll(pageable);

### **3️⃣ Custom Finder Methods (Derived Query Methods)**

Spring Data MongoDB allows you to define **custom finder methods** just by following a naming convention.

|  |  |
| --- | --- |
| Method | Description |
| findByPersonalName(String name) | Finds users by name. |
| findByPersonalEmail(String email) | Finds users by email. |
| findByLanguagesName(String language) | Finds users who know a specific language. |
| findByCredentialsUsername(String username) | Finds users by username. |
| findByCurrentCity(String city) | Finds users by current city. |

👉 **Example: Using a Custom Finder Method**

List<User> users = userRepository.findByPersonalName("Dineshkumar");

### **4️⃣ Custom Queries Using** @Query **Annotation**

For more complex queries, use **MongoDB's Query Language**.

#### **📌 Example: Find Users by Name (Case Insensitive)**

@Query("{'personal.name': {$regex: ?0, $options: 'i'}}")

List<User> searchByName(String name);

#### **📌 Example: Find Users with Specific Language**

@Query("{'languages.name': ?0}")

List<User> findByLanguage(String language);

## **📝 Summary**

The MongoRepository provides:

1. **Basic CRUD methods** (save(), findAll(), findById(), deleteById())
2. **Sorting & Pagination** (findAll(Sort), findAll(Pageable))
3. **Custom Finder Methods** (findByPersonalName(), findByCredentialsUsername())
4. **Custom Queries** using @Query

# **Use of Optional and ResponseEntity**

### **1️⃣ Optional in Java**

Optional<T> is a container object introduced in Java 8 that **may or may not contain a non-null value**. It helps in handling **null values safely** and **avoiding NullPointerException**.

#### **Why use Optional?**

* Prevents **NullPointerException**.
* Makes code **more readable and concise**.
* Provides methods like orElse(), orElseGet(), orElseThrow() to handle the absence of values.

#### **Common Optional Methods**

Optional<User> optionalUser = userRepository.findById("123");

|  |  |
| --- | --- |
| Method | Description |
| isPresent() | Returns true if a value is present, otherwise false. |
| get() | Returns the value if present, otherwise throws NoSuchElementException. |
| orElse(T other) | Returns the value if present, otherwise returns the default value. |
| orElseGet(Supplier<T>) | Returns the value if present, otherwise calls a function to provide a default value. |
| orElseThrow(Supplier<Exception>) | Returns the value if present, otherwise throws an exception. |

#### **Example**

public User getUserById(String id) {

return userRepository.findById(id)

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

}

### **2️⃣ ResponseEntity in Spring Boot**

ResponseEntity<T> is a class in Spring Boot that **represents the entire HTTP response**, including:

* **Status Code (200, 400, 500, etc.)**
* **Headers**
* **Body (data, error messages, etc.)**

#### **Why use ResponseEntity?**

* Provides **better control over HTTP responses**.
* Allows setting **custom status codes** and **headers**.
* Improves API **readability and flexibility**.

#### **Common Usage**

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

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

Optional<User> user = userRepository.findById(id);

return user.map(ResponseEntity::ok) // If found, return 200 OK with user

.orElseGet(() -> ResponseEntity.status(HttpStatus.NOT\_FOUND)

.body(null)); // If not found, return 404

}

#### **Creating Responses with ResponseEntity**

|  |  |
| --- | --- |
| Method | Description |
| ok(T body) | Returns 200 OK with a body. |
| status(HttpStatus).body(T body) | Returns response with a custom status and body. |
| noContent() | Returns 204 No Content (when there is nothing to return). |
| badRequest().body(T body) | Returns 400 Bad Request. |
| notFound().build() | Returns 404 Not Found. |

#### **Example**

@PostMapping("/register")

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

if (userRepository.existsByEmail(user.getEmail())) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST)

.body("Email already exists!");

}

userRepository.save(user);

return ResponseEntity.status(HttpStatus.CREATED)

.body("User registered successfully!");

}

### **🔹 Combining Optional & ResponseEntity**

@GetMapping("/{id}")

public ResponseEntity<User> getUserById(@PathVariable String id) {

return userRepository.findById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

This **avoids null checks** and **ensures proper HTTP response handling**.

### **Final Thoughts**

✅ Optional<T> → Used for **null-safe** handling of data.  
✅ ResponseEntity<T> → Used for **customized HTTP responses**.  
✅ **Combining both** makes your API robust and **handles missing data gracefully**.

# **Spring Boot MongoDB Custom Queries (**@Query**)**

## **📌 Introduction**

In Spring Boot with MongoDB, we often use the MongoRepository interface to perform basic CRUD operations. However, for complex queries, **custom queries** using the @Query annotation come in handy.

## **📖 Table of Contents**

1️⃣ What is @Query in Spring Boot?  
2️⃣ Why use @Query instead of method naming?  
3️⃣ Basic Syntax of @Query  
4️⃣ **Examples**

* **Basic Filtering**
* **Using Logical Operators**
* **Using Comparison Operators**
* **Using $regex for Pattern Matching**
* **Using $in for Multiple Values**
* **Using $gte and $lte for Range Queries**
* **Aggregation Queries** 5️⃣ Summary

## **1️⃣ What is** @Query **in Spring Boot?**

* @Query allows defining **custom MongoDB queries** inside repository methods.
* Instead of relying on **method name conventions**, you can **write actual MongoDB queries** inside @Query.

## **2️⃣ Why use** @Query **instead of method naming?**

### ✅ ****Advantages of**** @Query

* **More control over queries** (e.g., complex conditions).
* **Easier to optimize** compared to method name-based queries.
* **Supports MongoDB operators** like $in, $gte, $lte, $regex, etc.
* **Works well with Aggregation Pipelines**.

## **3️⃣ Basic Syntax of** @Query

@Query("{ 'fieldName': ?0 }")

List<Entity> findByField(String value);

* { 'fieldName': ?0 } → MongoDB JSON query.
* ?0 → First parameter passed to the method.

## **4️⃣ Examples of Custom Queries**

### ****📌 1️⃣ Basic Filtering****

#### **Query: Find users by city and state**

@Query("{ 'current.city': ?0, 'current.state': ?1 }")

List<User> findByCityAndState(String city, String state);

#### **Example Call:**

userRepository.findByCityAndState("Chennai", "Tamil Nadu");

#### **Equivalent MongoDB Query:**

{

"current.city": "Chennai",

"current.state": "Tamil Nadu"

}

### ****📌 2️⃣ Using Logical Operators (****$or****,**** $and****)****

#### **Query: Find users in either "Chennai" or "Bangalore"**

@Query("{ '$or': [ { 'current.city': ?0 }, { 'current.city': ?1 } ] }")

List<User> findByCityOrCity(String city1, String city2);

#### **Example Call:**

userRepository.findByCityOrCity("Chennai", "Bangalore");

#### **Equivalent MongoDB Query:**

{

"$or": [

{ "current.city": "Chennai" },

{ "current.city": "Bangalore" }

]

}

### ****📌 3️⃣ Using Comparison Operators (****$gte****,**** $lte****)****

#### **Query: Find users whose age is greater than or equal to 25**

@Query("{ 'personal.age': { '$gte': ?0 } }")

List<User> findByAgeGreaterThanEqual(int age);

#### **Example Call:**

userRepository.findByAgeGreaterThanEqual(25);

#### **Equivalent MongoDB Query:**

{

"personal.age": { "$gte": 25 }

}

### ****📌 4️⃣ Using**** $regex ****for Pattern Matching****

#### **Query: Find users whose name starts with "D"**

@Query("{ 'personal.name': { '$regex': ?0, '$options': 'i' } }")

List<User> findByNamePattern(String pattern);

#### **Example Call:**

userRepository.findByNamePattern("^D");

#### **Equivalent MongoDB Query:**

{

"personal.name": { "$regex": "^D", "$options": "i" }

}

🔹 **Explanation:**

* ^D → Matches names **starting with "D"**.
* $options: 'i' → **Case-insensitive search**.

### ****📌 5️⃣ Using**** $in ****for Multiple Values****

#### **Query: Find users in a list of cities**

@Query("{ 'current.city': { '$in': ?0 } }")

List<User> findByCities(List<String> cities);

#### **Example Call:**

userRepository.findByCities(Arrays.asList("Chennai", "Mumbai", "Delhi"));

#### **Equivalent MongoDB Query:**

{

"current.city": { "$in": ["Chennai", "Mumbai", "Delhi"] }

}

🔹 **Explanation:**

* $in checks if current.city **exists in the given list**.

### ****📌 6️⃣ Using**** $gte ****and**** $lte ****for Date Ranges****

#### **Query: Find users born between two dates**

@Query("{ 'personal.dob': { '$gte': ?0, '$lte': ?1 } }")

List<User> findByDobBetween(Date startDate, Date endDate);

#### **Example Call:**

userRepository.findByDobBetween(

new SimpleDateFormat("yyyy-MM-dd").parse("1990-01-01"),

new SimpleDateFormat("yyyy-MM-dd").parse("2000-12-31")

);

#### **Equivalent MongoDB Query:**

{

"personal.dob": {

"$gte": "1990-01-01",

"$lte": "2000-12-31"

}

}

🔹 **Explanation:**

* Finds users whose dob is between 1990-01-01 and 2000-12-31.

### ****📌 7️⃣ Aggregation Queries****

#### **Query: Count users in each city**

@Aggregation("{ '$group': { '\_id': '$current.city', 'count': { '$sum': 1 } } }")

List<Map<String, Object>> countUsersByCity();

#### **Equivalent MongoDB Query:**

[

{

"$group": {

"\_id": "$current.city",

"count": { "$sum": 1 }

}

}

]

🔹 **Explanation:**

* Groups users **by city** and counts the number of users in each.

## **5️⃣ Summary**

|  |  |  |
| --- | --- | --- |
| **Query Type** | **MongoDB Query Syntax** | **Example Repository Method** |
| Find by exact match | { 'field': ?0 } | findByCity(String city) |
| $or operator | { '$or': [ { 'a': ?0 }, { 'b': ?1 } ] } | findByAOrB(String a, String b) |
| $in operator | { 'field': { '$in': ?0 } } | findByFieldIn(List<String> values) |
| $gte / $lte | { 'field': { '$gte': ?0, '$lte': ?1 } } | findByRange(int min, int max) |
| $regex | { 'field': { '$regex': ?0, '$options': 'i' } } | findByPattern(String regex) |
| Aggregation | { '$group': { '\_id': '$field', 'count': { '$sum': 1 } } } | countByField() |

## **🚀 Combining Multiple Operators in a Single Query & Projection (**$project**) in Spring Boot MongoDB**

In MongoDB, we can combine multiple operators inside the @Query annotation to build **powerful and efficient queries**. We can also use **projection ($project)** to return only the necessary fields.

## **📖 Table of Contents**

1️⃣ Combining Multiple Operators in @Query  
2️⃣ Using Projection ($project) to Return Specific Fields  
3️⃣ Example Queries  
4️⃣ Summary

# **1️⃣ Combining Multiple Operators in** @Query

We can combine operators like:

* $and (Logical AND)
* $or (Logical OR)
* $gte, $lte (Greater/Less than)
* $regex (Pattern Matching)
* $in (List of Values)

### ****📌 Example 1: Find users by city AND age greater than 25****

@Query("{ '$and': [ { 'current.city': ?0 }, { 'personal.age': { '$gte': ?1 } } ] }")

List<User> findByCityAndAgeGreaterThan(String city, int age);

#### **Example Call:**

userRepository.findByCityAndAgeGreaterThan("Chennai", 25);

#### **Equivalent MongoDB Query:**

{

"$and": [

{ "current.city": "Chennai" },

{ "personal.age": { "$gte": 25 } }

]

}

🔹 **Combines filtering on city (current.city) and age (personal.age).**

### ****📌 Example 2: Find users who either live in Chennai OR Bangalore and are older than 30****

@Query("{ '$and': [ { 'personal.age': { '$gte': ?0 } }, { '$or': [ { 'current.city': ?1 }, { 'current.city': ?2 } ] } ] }")

List<User> findByAgeAndCities(int age, String city1, String city2);

#### **Example Call:**

userRepository.findByAgeAndCities(30, "Chennai", "Bangalore");

#### **Equivalent MongoDB Query:**

{

"$and": [

{ "personal.age": { "$gte": 30 } },

{

"$or": [

{ "current.city": "Chennai" },

{ "current.city": "Bangalore" }

]

}

]

}

🔹 **Filters users based on age AND checks if they live in one of two cities.**

### ****📌 Example 3: Find users whose name starts with "D" AND live in Chennai****

@Query("{ '$and': [ { 'personal.name': { '$regex': ?0, '$options': 'i' } }, { 'current.city': ?1 } ] }")

List<User> findByNamePatternAndCity(String pattern, String city);

#### **Example Call:**

userRepository.findByNamePatternAndCity("^D", "Chennai");

#### **Equivalent MongoDB Query:**

{

"$and": [

{ "personal.name": { "$regex": "^D", "$options": "i" } },

{ "current.city": "Chennai" }

]

}

🔹 **Finds users whose name starts with "D" and live in "Chennai".**

# **2️⃣ Using Projection (**$project**) to Return Specific Fields**

By default, queries return **all fields**, but we can use **projections ($project)** to return **only required fields**.

### ****📌 Example 4: Return only**** name ****and**** email ****of users****

@Query(value = "{ 'current.city': ?0 }", fields = "{ 'personal.name': 1, 'personal.email': 1, '\_id': 0 }")

List<Map<String, Object>> findNamesAndEmailsByCity(String city);

#### **Example Call:**

userRepository.findNamesAndEmailsByCity("Chennai");

#### **Equivalent MongoDB Query:**

{

"current.city": "Chennai"

}

#### **Returned Data Example:**

[

{ "personal.name": "Dinesh", "personal.email": "sendmail2dk@gmail.com" },

{ "personal.name": "Karthik", "personal.email": "karthik123@gmail.com" }

]

🔹 **Hides \_id and other fields while returning only name and email.**

### ****📌 Example 5: Return**** name****,**** email****, and**** age ****for users older than 30****

@Query(value = "{ 'personal.age': { '$gte': ?0 } }", fields = "{ 'personal.name': 1, 'personal.email': 1, 'personal.age': 1, '\_id': 0 }")

List<Map<String, Object>> findBasicDetailsByAge(int age);

#### **Example Call:**

userRepository.findBasicDetailsByAge(30);

#### **Returned Data Example:**

[

{ "personal.name": "Dinesh", "personal.email": "sendmail2dk@gmail.com", "personal.age": 32 },

{ "personal.name": "Karthik", "personal.email": "karthik123@gmail.com", "personal.age": 35 }

]

🔹 **Returns only selected fields (name, email, age).**

### ****📌 Example 6: Return only**** city ****and**** state ****of users****

@Query(value = "{}", fields = "{ 'current.city': 1, 'current.state': 1, '\_id': 0 }")

List<Map<String, Object>> findAllCitiesAndStates();

#### **Returned Data Example:**

[

{ "current.city": "Chennai", "current.state": "Tamil Nadu" },

{ "current.city": "Mumbai", "current.state": "Maharashtra" }

]

🔹 **Returns only city and state for all users.**

## **3️⃣ Summary**

| **Query Type** | **MongoDB Query Syntax** | **Example Repository Method** |
| --- | --- | --- |
| **AND condition** | { '$and': [ { 'a': ?0 }, { 'b': ?1 } ] } | findByAAndB(String a, String b) |
| **OR condition** | { '$or': [ { 'a': ?0 }, { 'b': ?1 } ] } | findByAOrB(String a, String b) |
| **AND + OR combined** | { '$and': [ { 'x': ?0 }, { '$or': [ { 'y': ?1 }, { 'z': ?2 } ] } ] } | findByXAndYorZ(String x, String y, String z) |
| **Regex search** | { 'name': { '$regex': ?0, '$options': 'i' } } | findByNamePattern(String pattern) |
| **Projection ($project)** | { 'field1': 1, 'field2': 1, '\_id': 0 } | findByFieldProjection() |

# Spring Boot project with MongoDB

Kindly refer the SpringBootMongoDBJPA document.

**Phase 5: Advanced Spring Boot Features**

1. **Spring Boot Security (Optional)**
   * Securing APIs with Spring Security
   * JWT Authentication basics
2. **Spring Boot with React (Full Stack Connection - Optional)**

* Exposing APIs for frontend consumption
* CORS handling

**Phase 6: Project Development & Deployment**

1. **Building the Final Project**

* Designing database schema
* Implementing CRUD operations
* Adding business logic

1. **Testing & Debugging**

* Writing unit tests with JUnit & Mockito
* Using Postman for API testing

1. **Deployment & DevOps Basics**

* Packaging Spring Boot as a JAR
* Deploying to AWS/Heroku/Render