Lombok, Request Handling, Validation & Web Scopes"

📦 What is Lombok?

Project Lombok is a Java library that plugs into your IDE and build tools (Maven/Gradle).
It helps you **auto-generate** common code like:

- Getters & setters
- Constructors
- toString(), equals(), hashCode()
- Builders
- Loggers

👉 **You write less Java**, Lombok generates the rest **at compile time**.

✅ Why Use Lombok?

Without Lombok:

java

CopyEdit

```
public class Student {
    private String name;
    private int age;

    public Student() {}
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    // and so on...
}
```

With Lombok:

```
java
CopyEdit
import lombok.*;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
    private String name;
    private int age;
}
```

✅ Much cleaner, same result.

🧰 How to Add Lombok (Spring Boot / Maven)

In pom.xml:

```
xml
CopyEdit
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version> <!-- latest at time of writing -->
  <scope>provided</scope>
</dependency>
```

Also enable **Lombok plugin** in IntelliJ or Eclipse (important!).

🔑 Common Lombok Annotations

✅ @Getter and @Setter

Generates getter/setter for all fields.

java

CopyEdit

@Getter

@Setter

public class Student {

    private String name;

    private int age;

}

✅ @Data

Shortcut for:

- @Getter
- @Setter
- @ToString
- @EqualsAndHashCode
- @RequiredArgsConstructor

java

CopyEdit

@Data

public class Employee {

    private String id;

    private String name;

}

✅ **@NoArgsConstructor, @AllArgsConstructor, @RequiredArgsConstructor**

java

CopyEdit

```
@NoArgsConstructor
@AllArgsConstructor
public class Course {
    private String name;
    private int duration;
}
```

✅ **@Builder**

Allows you to build objects like:

java

CopyEdit

```
Student s = Student.builder()
                .name("Ashif")
                .age(25)
                .build();
```

java

CopyEdit

```
@Builder
public class Student {
    private String name;
    private int age;
}
```

✅ **@Slf4j – Logger**

Automatically adds a logger instance:

java

CopyEdit

```
@Slf4j
public class DemoService {
    public void serve() {
        log.info("Serving request...");
    }
}
```

🛠️ **Real Use Case in Spring Boot**

Without Lombok:

java

CopyEdit

```
public class Contact {
    private String name;
    private String email;

    // Getters, setters, constructor, toString() ...
}
```

With Lombok:

java

CopyEdit

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Contact {
```

```
    private String name;

    private String email;

}
```

✅ Less code, same functionality.

🏁 **Summary**

Annotation	Use
@Getter, @Setter	Generate getters/setters
@Data	Full POJO boilerplate
@NoArgsConstructor, @AllArgsConstructor	Generate constructors
@Builder	Create builder pattern
@Slf4j	Logger without writing Logger logger = LoggerFactory...

Awesome! Let's do a hands-on Spring Boot exercise that uses **Lombok** to simplify a form submission flow.

We'll simulate a **"Student Registration"** form with fields like name, email, and course.

🎯 **Objective:**

- Use **Lombok** to define a POJO
- Bind it to a **form**
- Use @**ModelAttribute** to receive form data
- Display a confirmation page using **Thymeleaf**

🧪 **Step-by-Step: Hands-On with Lombok**

✅ **1. Create the Lombok-Enabled Model Class**

📄 Student.java

```
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {
    private String name;
    private String email;
    private String course;
}
```

✅ No need to manually create getters, setters, constructors — Lombok does it all.

✅ **2. Create Controller**

📄 StudentController.java

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
public class StudentController {

    @GetMapping("/register")
    public String showForm(Model model) {
        model.addAttribute("student", new Student()); // empty object for form binding
        return "register";
    }
```

```
    @PostMapping("/submitRegistration")
    public String submitForm(@ModelAttribute("student") Student student, Model model) {
        model.addAttribute("registeredStudent", student);
        return "success";
    }
}
```

✅ **3. Create Form Page**

📄 register.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Student Registration</title>
</head>
<body>
<h2>Register Student</h2>

<form th:action="@{/submitRegistration}" th:object="${student}" method="post">
    <label>Name:</label>
    <input type="text" th:field="*{name}" /><br><br>

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

    <label>Course:</label>
    <input type="text" th:field="*{course}" /><br><br>
```

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


</body>
</html>
```

✅ 4. Create Success Page

📄 success.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Success</title>
</head>
<body>
<h2>Registration Successful!</h2>
<p>Name: <span th:text="${registeredStudent.name}"></span></p>
<p>Email: <span th:text="${registeredStudent.email}"></span></p>
<p>Course: <span th:text="${registeredStudent.course}"></span></p>
</body>
</html>
```

✅ Result:

1. Visit http://localhost:8080/register
2. Fill in the form
3. Submit → Redirects to confirmation page
4. All values are populated through the **Lombok-powered POJO**

💥 Key Benefits Observed

With Lombok	Without Lombok
@Data, @NoArgsConstructor, @AllArgsConstructor save 15+ lines	Manual getters/setters
Cleaner code	More noise
Faster development	Slower, repetitive

Would you like to now:

- Add **validation** to this form using Java Bean + Hibernate Validator? (e.g., @NotEmpty, @Email)
- Or move to @**RequestParam** use cases?

Let me know which track you'd prefer!

Great question! Let's **clearly compare** @RequiredArgsConstructor vs @AllArgsConstructor in Lombok — because they look similar, but behave differently.

🔍 **1. @RequiredArgsConstructor**

✅ **What it does:**

Generates a constructor **only for**:

- Fields marked final
- OR fields annotated with @NonNull

❗ **Ignores:**

- Non-final and non-annotated fields

💡 **Use case:**

Ideal for **constructor-based dependency injection in Spring Boot**, where dependencies are marked final.

💉 **Example:**

@RequiredArgsConstructor

public class UserService {

```
    private final UserRepository userRepository;
    private String optionalField;
}
```

✅ Generated constructor:

```
public UserService(UserRepository userRepository) {
    this.userRepository = userRepository;
}
```

No constructor parameter for optionalField because it's not final or @NonNull.

🔍 2. @AllArgsConstructor

✅ **What it does:**

Generates a constructor **for all fields**, regardless of whether they're final, non-final, or @NonNull.

💡 **Use case:**

When you want to quickly create a full-argument constructor for object creation.

💉 **Example:**

```
@AllArgsConstructor
public class UserService {
    private final UserRepository userRepository;
    private String optionalField;
}
```

✅ Generated constructor:

```
public UserService(UserRepository userRepository, String optionalField) {
    this.userRepository = userRepository;
    this.optionalField = optionalField;
}
```

📊 Side-by-Side Comparison

Feature	@RequiredArgsConstructor	@AllArgsConstructor
Includes final fields	✅ Yes	✅ Yes
Includes non-final fields	❌ No (unless annotated with @NonNull)	✅ Yes
Ignores @NonFinal fields	✅ Yes	❌ No (includes all)
Best use case	Spring DI constructor injection	Full-object creation manually
Spring-friendly?	✅ Most preferred	⚠️ Used rarely for services

---

🧠 Interview Summary:

"@RequiredArgsConstructor creates a constructor only for required fields (final or @NonNull) — perfect for dependency injection in Spring.
@AllArgsConstructor creates a constructor for **all fields**, useful for testing or object creation but not ideal for Spring-managed beans."

Would you like to try a small **practice example combining both**, or move to **@RequestParam deep dive** now?

Excellent — let's learn **@Slf4j in Lombok** clearly and practically.

🔧 What is @Slf4j?

@Slf4j is a Lombok annotation that **automatically injects a logger** into your Java class — so you don't need to manually declare one.

🛑 Without Lombok:

You normally write:

```
import org.slf4j.Logger;

import org.slf4j.LoggerFactory;
```

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

    public void sendEmail() {
        log.info("Email sent successfully!");
    }
}
```

✅ **With Lombok:**

You write:

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class EmailService {

    public void sendEmail() {
        log.info("Email sent successfully!");
    }
}
```

✅ Cleaner and shorter — log is automatically injected as:

```
private static final org.slf4j.Logger log = LoggerFactory.getLogger(CurrentClassName.class);
```

📘 **What is SLF4J?**

SLF4J stands for **Simple Logging Facade for Java**. It's an API, not an actual logger.

You typically pair it with:

- **Logback** (Spring Boot's default logging backend)
- **Log4j2** or others (optional)

💡 **Use Cases in Spring Boot**

- Logging API requests/responses
- Debugging service methods
- Handling exceptions

✅ **Example:**

```
@Slf4j
@Service
public class OrderService {

    public void placeOrder(String userId) {
        log.info("Placing order for user: {}", userId);
        try {
            // logic...
        } catch (Exception e) {
            log.error("Error placing order", e);
        }
    }
}
```

📌 **Log Levels:**

Method	Use for
log.trace(...)	Very detailed debug info
log.debug(...)	Developer-level debugging
log.info(...)	General app flow messages
log.warn(...)	Something unexpected, but app continues
log.error(...)	Errors, exceptions, failures

🛠 Enabling Logging in application.properties

logging.level.com.yourpackage=DEBUG

You can also write logs to file:

logging.file.name=logs/app.log

🧠 Summary

Feature	Purpose
@Slf4j	Injects SLF4J logger
Avoids manual declaration	LoggerFactory.getLogger(...)
Works with Spring Boot's Logback	Yes
Use for	Logging, debugging, error tracking

Would you like me to update the **Lombok document to include @Slf4j in detail**, or proceed to @RequestParam now?

Great! Let's move on to the next topic in your roadmap:

🎯 @RequestParam in Spring Boot — In-Depth

This is one of the most used annotations in Spring MVC for **handling form inputs or query parameters**.

🔍 What is @RequestParam?

@RequestParam is used to **extract query parameters** or **form data** from the incoming HTTP request and bind them directly to method arguments.

✅ Example: Query Parameter

@GetMapping("/greet")

public String greet(@RequestParam String name) {

```
    return "Hello, " + name;
}
```

URL:

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

✅ Output:

Hello, Ashif

🔧 **When is @RequestParam used?**

Used When	Example
A value is passed as a query string	/search?query=Java
A form is submitted using method="GET" or method="POST"	form with <input name="email" />
You want to grab a simple string/number from the URL or form	Name, Age, Status, etc.

🧠 **Full Syntax**

@RequestParam(value = "name", required = true, defaultValue = "Guest")

Attribute	Meaning
value	Name of the parameter
required	Whether the param is mandatory (true by default)
defaultValue	Value to use if none is provided

💉 **Example: With Default and Optional**

```
@GetMapping("/greet")
public String greet(@RequestParam(value = "name", required = false, defaultValue = "Guest")
String name) {
    return "Hello, " + name;
}
```

✅ Works even for /greet without a query parameter → returns "Hello, Guest"

💉 POST Form with RequestParam

HTML Form:

```
<form action="/submit" method="post">
    <input type="text" name="email" />
    <input type="text" name="name" />
    <button type="submit">Submit</button>
</form>
```

Controller:

```
@PostMapping("/submit")
public String handleForm(@RequestParam String name, @RequestParam String email) {
    return "Thanks " + name + ", we'll email you at " + email;
}
```

⚠️ Common Mistakes

- ❌ Using wrong parameter name: Must match the name in the form/input
- ❌ Missing required value: causes 400 Bad Request unless you use required = false
- ❌ Forgetting to set method="post" or method="get" in form

🔁 Comparing with Others

Annotation	Use
@RequestParam	For form/query values
@PathVariable	For URL path values (/user/{id})
@ModelAttribute	For binding full objects (used with forms)
@RequestBody	For reading JSON payload in REST APIs

🧠 Interview Line You Can Say:

"@RequestParam lets me map query string or form parameters directly into method arguments in Spring Boot.
It's ideal for quick inputs like filters, search terms, or form fields without needing full object mapping."

Would you like to:

- Try a **practice task** using @RequestParam and a form?
- Or continue to **@PathVariable** next?

Awesome — let's dive deep into:

🧩 **@PathVariable in Spring Boot — In-Depth Guide**

@PathVariable is used when you want to **extract data directly from the URL path**, instead of query strings or form data.

✅ **What is @PathVariable?**

It binds a portion of the **URL path itself** to a method parameter in your controller.

🧠 **Simple Example:**

```
@GetMapping("/greet/{name}")
public String greet(@PathVariable String name) {
    return "Hello, " + name;
}
```

If a user accesses:

http://localhost:8080/greet/Ashif

✅ Output:

Hello, Ashif

💡 **Syntax Overview**

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

public String getUser(@PathVariable("id") Long userId)

- @PathVariable("id") tells Spring to extract the {id} from the path
- It maps it to the method variable userId

🍼 Example: Multiple Path Variables

@GetMapping("/course/{courseId}/student/{studentId}")

public String enroll(@PathVariable String courseId, @PathVariable String studentId) {

return "Student " + studentId + " enrolled in course " + courseId;

}

URL:

http://localhost:8080/course/SPRING101/student/STU500

✅ Output:

Student STU500 enrolled in course SPRING101

🧱 Why Use Path Variables?

Use Case	Benefit
RESTful API design	Clean, readable URLs like /users/101
Simpler routing	Avoids query strings for ID-based routes
Resource targeting	CRUD APIs for entities (GET/PUT/DELETE /id)

🔁 Optional Parameters?

@PathVariable is **required by default**.

If needed, you must:

1. Change the endpoint itself (e.g., use different mappings)
2. Use @RequestParam for optional fields instead

🔍 **Difference: @RequestParam vs @PathVariable**

Feature	@RequestParam	@PathVariable
Reads from	Query string or form input	URL path
Used for	Filters, forms, optional	Resource IDs, required
Example URL	/user?name=Ashif	/user/Ashif
Flexible?	More flexible	Tied to route structure

🧠 **Interview Line You Can Say:**

"@PathVariable allows me to extract dynamic segments from a URL path and bind them directly into my controller methods.
It's best used in REST APIs for accessing specific resources by ID or type."

Would you like to now try a **form + @PathVariable example** or go straight to **Java Bean & Hibernate Validation** in form inputs?

Types of Validation:
1) Client-side Validation

2) Server-Side Validation

Perfect! Let's now explore:

✅ **Input Validation using Java Bean and Hibernate Validators — General Usage**

This is how we make sure **input values meet certain rules** (like non-empty, within range, valid email, etc.) **before processing them in Spring Boot**.

We use:

- **Java Bean Validation API** (JSR-380)
- **Hibernate Validator** (the most common implementation)

🧠 **Why Do We Need Validation?**

To **avoid accepting invalid data**, such as:

- Blank names
- Negative ages
- Incorrect emails
- Too-short passwords

Without validation, these might break your logic or corrupt your database.

✅ **1. Add Hibernate Validator Dependency (Spring Boot already includes it)**

If needed separately:

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

✅ **2. Annotate Your POJO**

```
import jakarta.validation.constraints.*;

public class User {

    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Enter a valid email")
    private String email;

    @Min(value = 18, message = "Must be at least 18 years old")
    private int age;

    @Size(min = 6, message = "Password must be at least 6 characters")
```

```
    private String password;

    // getters & setters
}
```

✅ 3. Use @Valid in Controller (for API or Web)

```
@PostMapping("/register")
public ResponseEntity<String> register(@RequestBody @Valid User user, BindingResult result) {
    if (result.hasErrors()) {
        return ResponseEntity.badRequest().body("Validation failed");
    }
    return ResponseEntity.ok("User registered");
}
```

Spring will:

- Validate user based on annotations
- Inject any validation errors into BindingResult

💉 Supported Validation Annotations

Annotation	What it does
@NotNull	Field must not be null
@NotBlank	Must not be empty or whitespace
@Email	Must be a valid email format
@Min(value)	Must be ≥ value
@Max(value)	Must be ≤ value
@Size(min, max)	Length must be within given bounds
@Pattern(regex)	Must match the provided regular expression

Annotation	What it does
@Positive	Must be > 0
@NegativeOrZero	Must be ≤ 0

🧠 **Tip: Nested Object Validation**

If your class has another class inside it (like an Address inside User), annotate it with @Valid.

```
public class User {
    @Valid
    private Address address;
}
```

🏁 **Summary**

- Use **Java annotations** to validate inputs in your Java class.
- Use @Valid to trigger validation in Spring Controllers.
- Access validation errors via BindingResult.
- This works for **REST APIs**, **Thymeleaf forms**, and **Spring services**.

Would you like to now:

- Try a quick example using a REST API with validation?
- Or move to the next topic: **Web Scoped Beans (@RequestScope, etc.)**?

Absolutely! Let's take a **deeper and clearer dive into @Valid** — it's a core part of form and request validation in Spring Boot using the **Java Bean Validation** standard.

✅ **What is @Valid in Spring Boot?**

@Valid is an annotation provided by **javax.validation** (or **jakarta.validation** in newer versions) that:

Triggers **automatic validation** of the annotated object, based on constraints defined on its fields.

It works for:

- Spring **REST controllers** (@RequestBody)
- Spring **MVC controllers** (@ModelAttribute)
- **Nested objects** inside a POJO
- Even **service-layer methods** (when using @Validated on class)

📘 Where You Can Use @Valid

Place	Example
REST controller (JSON input)	@RequestBody @Valid
Web controller (Form binding)	@ModelAttribute @Valid
Nested objects	@Valid on a field
Service layer (extra)	Use @Validated + @Valid

🧠 How @Valid Works

Let’s say you have this POJO:

```
public class Contact {
    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Enter a valid email")
    private String email;
}
```

Now, in your controller:

```
@PostMapping("/contact")
public String handleContact(@Valid @ModelAttribute Contact contact, BindingResult result) {
    if (result.hasErrors()) {
```

```
        return "contactForm"; // show form again with error messages
    }
    return "success";
}
```

🔥 **What happens:**

- Spring checks the Contact object for all constraints (e.g., @NotBlank, @Email)
- If any rule fails → BindingResult will contain the error(s)
- You can now return a custom error message, or stop further execution

---

💉 **REST API Example (with @RequestBody)**

```
@PostMapping("/api/users")
public ResponseEntity<?> registerUser(@Valid @RequestBody User user, BindingResult result)
{
    if (result.hasErrors()) {
        return ResponseEntity.badRequest().body("Invalid user data");
    }
    return ResponseEntity.ok("User created");
}
```

🧠 **Key Points About @Valid**

✅ **1. @Valid doesn't throw exceptions directly.**

It populates a BindingResult or throws MethodArgumentNotValidException (if no BindingResult provided).

✅ **2. It cascades into nested objects when used with @Valid.**

```
public class User {
    @Valid
    private Address address;
```

```
}
```

✅ **3. Used with @ModelAttribute, @RequestBody, or even @PathVariable (rare)**

✅ **4. For method-level validation, you use @Validated:**

```
@Validated
@Service
public class MyService {
    public void saveUser(@Valid User user) { ... }
}
```

⚠️ **Difference: @Valid vs @Validated**

Feature	@Valid	@Validated
From	JSR-303 (javax/jakarta.validation)	Spring-specific
Used for	Objects (fields or method params)	Classes/methods
Nested object support	✅ Yes	✅ Yes
Groups (advanced)	❌ No	✅ Yes (for group-based validation)

✅ **Final Summary**

- @Valid = core trigger for validation
- Works on form objects, JSON payloads, nested beans
- Paired with BindingResult or used in REST APIs
- Can validate entire POJOs automatically

Would you like a **live hands-on REST API with @Valid and custom error messages**, or should we now move to the next topic: **Web Scoped Beans** (@RequestScope, @SessionScope)?

✅ Great question — and it's important to understand **when** validation using @Valid happens.

🔍 **So, does @Valid validation happen *before* the controller?**

🟢 **Answer: Yes, but only partially — it depends on the context.**

Let's break it down clearly.

🧠 **In a Spring Boot REST Controller:**

When using:

@PostMapping("/user")

public ResponseEntity<?> create(@Valid @RequestBody User user) { ... }

What happens:

- Spring **validates the User object before** the controller method executes.
- If the validation **fails**, Spring throws a MethodArgumentNotValidException.
- If you have **no BindingResult**, the controller method **will NOT be called** at all.

So yes — for REST APIs:

❗ **Validation happens before the controller is actually invoked.**

You'll typically handle the exception using:

@ControllerAdvice

@ExceptionHandler(MethodArgumentNotValidException.class)

🧠 **In a Spring Boot Web MVC Controller:**

When using:

@PostMapping("/register")

public String handle(@Valid @ModelAttribute User user, BindingResult result) { ... }

What happens:

- Spring **performs validation just before** method execution
- But the method is still called

- You must check BindingResult for errors manually

❗ In this case, controller method is invoked **with errors already available**.

🔁 Summary Table

Scenario	Does Validation Happen Before Controller?	Is Controller Skipped on Failure?
@Valid @RequestBody (REST)	✅ Yes	✅ Yes (if no BindingResult)
@Valid @ModelAttribute (Form)	✅ Yes	❌ No (controller is called)
With BindingResult	✅ Yes	❌ No (controller handles errors)

🧠 Interview Insight

"@Valid triggers automatic validation before the controller method runs. In REST controllers, this throws an exception and skips the method if validation fails. In web controllers with forms, it still calls the method, and you handle validation via BindingResult."

Would you like me to now show:

- How to handle failed validation in **REST APIs** globally?
- Or move to the next topic: **Web Scoped Beans** (@RequestScope, etc.)?

Excellent! Let’s dive into **Web Scoped Beans in Spring Boot**, clearly and step-by-step.

🌐 What Are Web Scoped Beans?

In a Spring Boot **web application**, you might want some beans to:

- Be **created per request**
- Or **persist across a user’s session**
- Or **live globally across the application**

To control that behavior, Spring provides **web scopes**:

✅ **1. @RequestScope**

- 🔁 A **new bean instance is created for every HTTP request**
- 💡 Used for storing per-request data (e.g., a user's form data)

Example:

```
@Component
@RequestScope
public class RequestTracker {
    private final String requestId = UUID.randomUUID().toString();

    public String getRequestId() {
        return requestId;
    }
}
@RestController
public class MyController {
    @Autowired
    private RequestTracker tracker;

    @GetMapping("/track")
    public String getRequestId() {
        return tracker.getRequestId();
    }
}
```

🔁 Each time /track is called, you get a different requestId.

✅ **2. @SessionScope**

- 🧠 One bean per **HTTP session**
- Lives as long as the user's session is alive (like a login session)
- Often used to store **logged-in user info**

Example:

```
@Component
@SessionScope
public class SessionData {
    private String username;

    // getter/setter
}
@RestController
public class SessionController {

    @Autowired
    private SessionData sessionData;

    @GetMapping("/set/{name}")
    public String setUser(@PathVariable String name) {
        sessionData.setUsername(name);
        return "Username set!";
    }

    @GetMapping("/get")
    public String getUser() {
        return "Current session user: " + sessionData.getUsername();
    }
}
```

✅ Now:

- /set/Ashif stores the name in session
- /get reads it for the current session only

✅ 3. @ApplicationScope

- 🏛️ A **singleton shared across the entire application**
- All users and all sessions use the same instance

Example:

```
@Component
@ApplicationScope
public class AppTracker {
    private int count = 0;

    public void increment() { count++; }
    public int getCount() { return count; }
}
```

✅ All users share the same counter.

🧠 Summary of Web Scopes

Annotation	Scope	Lifecycle	Use Case
@RequestScope	Per request	One HTTP request	Logging, per-request data
@SessionScope	Per session	One browser session	Logged-in user data, cart
@ApplicationScope	Global	App lifetime	Shared config, global metrics

---

🔥 Important Notes

- To use @RequestScope and @SessionScope, Spring Boot **must be a web app**.
- These scopes work only in **HTTP-aware contexts** (not in CLI apps).

- You can also use them in **Beans created with @Bean**, like:

```
@Bean
@RequestScope
public UserInfo userInfo() {
    return new UserInfo();
}
```

🧠 **Interview Line You Can Say:**

"@RequestScope, @SessionScope, and @ApplicationScope in Spring Boot control the lifecycle of beans depending on the web context. They're essential for managing request-specific or user-specific data without manually handling state."

Would you like to now try a **real example using all 3 scopes**, or move to **Web API Security Implementation**?

Perfect! Let's build a **real example that demonstrates all 3 web scopes**:
✅ @RequestScope, ✅ @SessionScope, and ✅ @ApplicationScope — in one mini Spring Boot app.

🧪 **Use Case: Track Page Visits**

We'll simulate a scenario where:

- Each **request** gets a unique ID (RequestScope)
- A **user session** tracks how many pages the user has visited (SessionScope)
- The **entire application** counts total visits across all users (ApplicationScope)

✅ **Step 1: Create the Beans**

🔹 **RequestTracker.java — @RequestScope**

```
import org.springframework.stereotype.Component;
import org.springframework.web.context.annotation.RequestScope;
```

```
import java.util.UUID;


@Component
@RequestScope
public class RequestTracker {
    private final String requestId = UUID.randomUUID().toString();


    public String getRequestId() {
        return requestId;
    }
}
```

- **SessionTracker.java — @SessionScope**

```
import org.springframework.stereotype.Component;
import org.springframework.web.context.annotation.SessionScope;


@Component
@SessionScope
public class SessionTracker {
    private int sessionVisitCount = 0;


    public void increment() {
        sessionVisitCount++;
    }


    public int getSessionVisitCount() {
        return sessionVisitCount;
```

```
    }
}
```

🔹 **AppTracker.java — @ApplicationScope**

```
import org.springframework.stereotype.Component;
import org.springframework.web.context.annotation.ApplicationScope;

@Component
@ApplicationScope
public class AppTracker {
    private int totalVisits = 0;

    public synchronized void increment() {
        totalVisits++;
    }

    public int getTotalVisits() {
        return totalVisits;
    }
}
```

✅ **Step 2: Create the Controller**

🔹 **VisitController.java**

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
```

```
public class VisitController {

    @Autowired
    private RequestTracker requestTracker;

    @Autowired
    private SessionTracker sessionTracker;

    @Autowired
    private AppTracker appTracker;

    @GetMapping("/visit")
    public String visitPage() {
        sessionTracker.increment();
        appTracker.increment();

        return String.format(
            "Request ID: %s\nSession Visits: %d\nTotal App Visits: %d",
            requestTracker.getRequestId(),
            sessionTracker.getSessionVisitCount(),
            appTracker.getTotalVisits()
        );
    }
}
```

✅ **Step 3: Run the App and Test**

1. Start your Spring Boot app.
2. Visit http://localhost:8080/visit in your browser.

3. Refresh multiple times → session count increases.
4. Open a new browser/incognito → new session starts at 1.
5. Every request shows a **new Request ID**, but **session and app counters persist**.

🔁 Output Example

Request ID: 9b174d0d-534d-4f22-8001-7f20d2b23c7e

Session Visits: 3

Total App Visits: 12

🧠 Summary

Scope	Bean Name	Tracks
@RequestScope	RequestTracker	New ID per request
@SessionScope	SessionTracker	Pages visited in this session
@ApplicationScope	AppTracker	All visits across all users

Would you like me to generate a **PDF of this example**, or shall we continue to the next topic: **Web API Security Implementation**?