Activity guide

Spring Framework

Contents

[Activity 1.1 :Basic Setup & "Hello World" Application 4](#_Toc202878840)

[Activity 1.2: Basic Setup & "Hello World" Application using Spring Boot 9](#_Toc202878842)

[Activity 2.1: Spring DI Fundamentals 15](#_Toc202878848)

[Activity 3.1: Spring ORM JPA “Product Management” 25](#_Toc202878851)

[Activity 4.1: Spring AOP “Logging with Aspects” 46](#_Toc202878861)

[Activity 5.1: Spring Expression Language (SpEL) 60](#_Toc202878868)

[Activity 7.1: Spring Security “Basic Authentication & Authorization” 96](#_Toc202878882)

[Activity 8.1: Spring Remoting “HTTP-based Communication with RestTemplate” 113](#_Toc202878890)

[Activity 9.1: Spring with JMS “Messaging with ActiveMQ” 125](#_Toc202878900)

[Activity 10.1: Spring JavaMail “Sending Emails in Spring Boot” 135](#_Toc202878909)

# Activity 1.1 :Basic Setup & "Hello World" Application

This guide will walk you through the basics of the Spring Framework, covering its core concepts and a simple "Hello World" application setup.

Let's create a very simple "Hello World" application to see Spring in action. We'll use Maven for project management.

#### STEP 1: Project Setup (Maven)

1. **Create a New Maven Project:**
   * Open your IDE (e.g., IntelliJ IDEA, Eclipse, VS Code).
   * Create a new Maven project.
   * Set the GroupId to com.example.spring and ArtifactId to hello-spring.
2. **Update pom.xml:**
   * Open the pom.xml file in your project root.
   * Add the spring-context dependency, which includes the core Spring IoC container.

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.example.spring</groupId>  
 <artifactId>hello-spring</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <maven.compiler.source>17</maven.compiler.source>  
 <maven.compiler.target>17</maven.compiler.target>  
 <spring.version>6.1.0</spring.version> <!-- Use a recent stable Spring version -->  
 </properties>  
  
 <dependencies>  
 <!-- Spring Core Dependency -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 </dependencies>  
</project>

1. **Download Dependencies:**
   * Right click on project and locate Maven option -> Click Sync Project to download maven dependencies (ntelliJ IDEA)
   * Option may be different for other IDE
   * Or use “mvn clean install” command using terminal in Project home folder.

#### STEP 2: Create a Simple Message Service

We'll create a simple service that provides a "Hello World" message.

* Create a new package com.example.spring.service in src/main/java/.
* Create a Java class named MessageService.java inside this package:

// src/main/java/com/example/spring/service/MessageService.java  
package com.example.spring.service;  
  
import org.springframework.stereotype.Service; // Import the @Service annotation  
  
/\*\*  
 \* This class is a simple service that provides a greeting message.  
 \* The @Service annotation tells Spring that this is a service component  
 \* and should be managed by the Spring IoC container.  
 \* Spring will automatically create an instance (bean) of this class.  
 \*/  
@Service // Marks this class as a Spring service component  
public class MessageService {  
  
 /\*\*  
 \* Returns a simple "Hello, Spring World!" message.  
 \* @return The greeting message.  
 \*/  
 public String getMessage() {  
 return "Hello, Spring World!";  
 }  
}

#### STEP 3: Create Spring Configuration

We need a configuration class to tell Spring where to find our components (like MessageService).

* Create a new package com.example.spring.config in src/main/java/.
* Create a Java class named AppConfig.java inside this package:

// src/main/java/com/example/spring/config/AppConfig.java  
package com.example.spring.config;  
  
import org.springframework.context.annotation.ComponentScan; // Import @ComponentScan  
import org.springframework.context.annotation.Configuration; // Import @Configuration  
  
/\*\*  
 \* This class is the Spring configuration for our application.  
 \* @Configuration indicates that this class contains bean definitions.  
 \* @ComponentScan tells Spring to scan the specified base package  
 \* (and its sub-packages) for Spring-annotated components like @Service,  
 \* @Component, @Repository, @Controller, etc., and register them as beans.  
 \*/  
@Configuration // Marks this class as a Spring configuration class  
@ComponentScan(basePackages = "com.example.spring") // Scans for components in "com.example.spring" and its sub-packages  
public class AppConfig {  
 // No explicit @Bean methods are needed here for MessageService  
 // because @ComponentScan will find @Service MessageService automatically.  
}

#### STEP 4: Create the Main Application Class

This is where we'll start the Spring IoC container and retrieve our MessageService bean.

* Create a new package com.example.spring.app in src/main/java/.
* Create a Java class named HelloWorldApp.java inside this package:

// src/main/java/com/example/spring/app/HelloWorldApp.java  
package com.example.spring.app;  
  
import com.example.spring.config.AppConfig; // Import our configuration class  
import com.example.spring.service.MessageService; // Import our service class  
import org.springframework.context.annotation.AnnotationConfigApplicationContext; // Import Spring's application context  
  
/\*\*  
 \* The main application class to demonstrate a basic Spring "Hello World".  
 \* It initializes the Spring IoC container and retrieves a bean to use it.  
 \*/  
public class HelloWorldApp {  
 public static void main(String[] args) {  
 System.out.println("--- Starting Spring IoC Container ---");  
  
 // 1. Create a Spring IoC Container (ApplicationContext)  
 // AnnotationConfigApplicationContext is used to load Java-based configurations (@Configuration classes).  
 // When 'AppConfig.class' is passed, Spring reads it, performs component scanning,  
 // and creates all the necessary beans (including MessageService).  
 try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {  
  
 System.out.println("\n--- Retrieving and Using Spring Bean ---");  
 // 2. Retrieve the MessageService bean from the container.  
 // Spring provides the instance of MessageService that it created and managed.  
 MessageService messageService = context.getBean(MessageService.class);  
  
 // 3. Use the retrieved bean to get the message.  
 String message = messageService.getMessage();  
 System.out.println("Received message from Spring bean: " + message);  
  
 } // The 'try-with-resources' ensures the context is closed automatically.  
 System.out.println("\n--- Spring IoC Container Shut Down ---");  
 System.out.println("Congratulations! You've successfully run your first Spring application!");  
 }  
}

### How to Run the Application:

1. **Run from your IDE:**
   * Open the HelloWorldApp.java file in your IDE.
   * Right-click within the file and select "Run 'HelloWorldApp.main()'" or click the green play button next to the main method.

You should see output similar to this in your console:

--- Starting Spring IoC Container ---  
  
--- Retrieving and Using Spring Bean ---  
Received message from Spring bean: Hello, Spring World!  
  
--- Spring IoC Container Shut Down ---  
Congratulations! You've successfully run your first Spring application!

This output demonstrates that Spring successfully initialized its container, found your MessageService (due to @Service and @ComponentScan), created an instance of it, and provided it when you requested the bean. This is the essence of Dependency Injection and Inversion of Control!

# Activity 1.2: Basic Setup & "Hello World" Application using Spring Boot

This activity will guide you through creating a simple "Hello World" application using Spring Boot, demonstrating how it simplifies project setup and execution compared to traditional Spring.

### Introduction: What is Spring Boot?

Spring Boot is a project built on top of the Spring Framework that aims to simplify the development of production-ready Spring applications. It provides:

* **Auto-configuration:** Automatically configures your Spring application based on the dependencies you've added.
* **Embedded Servers:** Can embed servers like Tomcat, Jetty, or Undertow directly into your executable JAR, so you don't need to deploy WAR files.
* **Opinionated Defaults:** Provides sensible default configurations to reduce the need for manual setup.
* **Stand-alone Applications:** Allows you to create stand-alone applications that can be run with just java -jar.

In essence, Spring Boot helps you "just run" your Spring applications with minimal fuss.

### STEP 1: Spring Boot Project Setup

The easiest and recommended way to create a Spring Boot project is by using **Spring Initializr**.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project (or Gradle, if preferred)
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.springboot (or your preferred group ID)
   * **Artifact:** hello-springboot (or your preferred artifact ID)
   * **Name:** hello-springboot
   * **Description:** Hello World Spring Boot Application
   * **Package Name:** com.example.springboot.app
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher (recommended for modern Spring Boot).
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** This starter pulls in Spring Core, Spring MVC, and an embedded Tomcat server, which is perfect for our simple "Hello World" web application (even though we won't expose a web endpoint, it's a common starter).
   * **Lombok:** (Optional but highly recommended) Reduces boilerplate code for getters, setters, constructors, etc.
4. **Generate and Download:** Click the "Generate" button. This will download a .zip file containing your new Spring Boot project.
5. **Import into IDE:** Unzip the downloaded file and import the project into your favorite IDE (IntelliJ IDEA, Eclipse, VS Code with Java extensions). Your IDE will typically recognize it as a Maven project and set it up automatically.
   * You will find a main application class already created for you, typically named HelloSpringbootApplication.java in the com.example.springboot.app package.

### STEP 2: Create a Simple Message Service

We'll create the same simple service as before that provides a "Hello World" message.

* Create a new package com.example.springboot.service in src/main/java/ (if it doesn't already exist).
* Create a Java class named MessageService.java inside this package:

// src/main/java/com/example/springboot/service/MessageService.java  
package com.example.springboot.service;  
  
import org.springframework.stereotype.Service; // Import the @Service annotation  
  
/\*\*  
 \* This class is a simple service that provides a greeting message.  
 \* The @Service annotation tells Spring that this is a service component  
 \* and should be managed by the Spring IoC container.  
 \* Spring Boot's auto-configuration and component scanning will automatically  
 \* detect and register this as a bean.  
 \*/  
@Service // Marks this class as a Spring service component  
public class MessageService {  
  
 /\*\*  
 \* Returns a simple "Hello, Spring Boot World!" message.  
 \* @return The greeting message.  
 \*/  
 public String getMessage() {  
 return "Hello, Spring Boot World!";  
 }  
}

### STEP 3: Update the Main Spring Boot Application Class

Spring Boot automatically generates a main class with @SpringBootApplication. We'll modify its main method to retrieve and use our MessageService.

* Open the HelloSpringbootApplication.java file (or whatever your main application class is named, typically in com.example.springboot.app package).+
* Modify it as follows:

// src/main/java/com/example/springboot/app/HelloSpringbootApplication.java  
package com.example.springboot.app;  
  
import com.example.springboot.service.MessageService; // Import our service class  
import org.springframework.boot.SpringApplication; // Import SpringApplication  
import org.springframework.boot.autoconfigure.SpringBootApplication; // Import @SpringBootApplication  
import org.springframework.context.ConfigurableApplicationContext; // Import ConfigurableApplicationContext  
  
/\*\*  
 \* The main entry point for our Spring Boot application.  
 \* @SpringBootApplication is a convenience annotation that adds:  
 \* - @Configuration: Tags the class as a source of bean definitions.  
 \* - @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings,  
 \* other beans, and various property settings.  
 \* - @ComponentScan: Tells Spring to look for other components, configurations, and services  
 \* in the 'com.example.springboot.app' package and its sub-packages.  
 \*/  
@SpringBootApplication  
public class HelloSpringbootApplication {  
  
 public static void main(String[] args) {  
 System.out.println("--- Starting Spring Boot Application ---");  
  
 // 1. Run the Spring Boot application.  
 // This method bootstraps the application, creates the Spring ApplicationContext,  
 // performs auto-configuration, and starts any embedded servers (if applicable).  
 ConfigurableApplicationContext context = SpringApplication.run(HelloSpringbootApplication.class, args);  
  
 System.out.println("\n--- Retrieving and Using Spring Bean ---");  
 // 2. Retrieve the MessageService bean from the container.  
 // Spring Boot's @ComponentScan (part of @SpringBootApplication)  
 // automatically found and registered our MessageService.  
 MessageService messageService = context.getBean(MessageService.class);  
  
 // 3. Use the retrieved bean to get the message.  
 String message = messageService.getMessage();  
 System.out.println("Received message from Spring Boot bean: " + message);  
  
 // 4. Close the application context gracefully.  
 // In a real web application, the server would keep running.  
 // For a simple command-line app, we close the context.  
 context.close();  
 System.out.println("\n--- Spring Boot Application Shut Down ---");  
 System.out.println("Congratulations! You've successfully run your first Spring Boot application!");  
 }  
}

### STEP 4: Run the Spring Boot Application

Running a Spring Boot application is incredibly simple.

1. **Run from your IDE:**
   * Open the HelloSpringbootApplication.java file in your IDE.
   * Right-click within the file and select "Run 'HelloSpringbootApplication.main()'" or click the green play button next to the main method.
2. **Run from Terminal (Maven Spring Boot Plugin):**
   * Open your terminal or command prompt.
   * Navigate to the root directory of your hello-springboot project (where pom.xml is located).
   * Run the Spring Boot Maven plugin command:  
     mvn spring-boot:run
   * This command will compile your project, package it, and then run the Spring Boot application.

You should see output similar to this in your console (Spring Boot's logs will be more verbose, but you'll find your custom messages within them):

. \_\_\_\_ \_ \_\_ \_ \_  
 /\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \  
( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \  
 \\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )  
 ' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /  
 =========|\_|==============|\_\_\_/=/\_/\_/\_/  
 :: Spring Boot :: (v3.x.x)  
  
... (Spring Boot startup logs) ...  
  
--- Starting Spring Boot Application ---  
  
--- Retrieving and Using Spring Bean ---  
Received message from Spring Boot bean: Hello, Spring Boot World!  
  
--- Spring Boot Application Shut Down ---  
Congratulations! You've successfully run your first Spring Boot application!  
  
... (Spring Boot shutdown logs) ...

This demonstrates how Spring Boot automates much of the configuration and setup, allowing you to focus directly on your application's logic.

# Activity 2.1: Spring DI Fundamentals

This activity will guide you through building a simple Spring application to understand Dependency Injection (DI), Spring's Inversion of Control (IoC) container, component scanning, autowiring, and bean scopes using Java-based configuration.

### Wiring Beans (Annotations & Java Config)

"Wiring" refers to the process of connecting beans with their dependencies. Spring provides several ways to do this. We will focus on **Annotations** and **Java Configuration**.

#### STEP 1: Project Setup (Maven)

1. **Create a New Maven Project:**
   * Open your IDE (IntelliJ IDEA, Eclipse, VS Code).
   * Create a new Maven project.
   * Set GroupId to com.example.di and ArtifactId to di-demo.
   * Package : com.example.di.demo
2. **Update pom.xml:**
   * Add the spring-context dependency.

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.example.di</groupId>  
 <artifactId>di-demo</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <maven.compiler.source>17</maven.compiler.source>  
 <maven.compiler.target>17</maven.compiler.target>  
 <spring.version>6.1.0</spring.version> <!-- Use a recent stable Spring version -->  
 </properties>  
  
 <dependencies>  
 <!-- Spring Core and Context for DI -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 </dependencies>  
</project>

1. **Download Dependencies:**  
   
2. **Create Package Structure:**
   * com.example.di.demo
   * com.example.di.demo.service
   * com.example.di.demo.component
   * com.example.di.demo.config

#### STEP 2: Define Components and Services

Let's create some simple classes to demonstrate DI.

* **MessageSender.java** (Interface in com.example.di.demo.component)  
  // src/main/java/com/example/di/demo/component/MessageSender.java  
  package com.example.di.component;  
    
  public interface MessageSender {  
   void sendMessage(String message);  
  }
* **EmailSender.java** (Annotation-based component in com.example.di.demo.component)  
  // src/main/java/com/example/di/demo/component/EmailSender.java  
  package com.example.di.demo.component;  
    
  import org.springframework.stereotype.Component;  
    
  @Component("emailSender") // Marks as Spring component with a specific name  
  public class EmailSender implements MessageSender {  
   @Override  
   public void sendMessage(String message) {  
   System.out.println("Email Sent: " + message);  
   }  
  }
* **SmsSender.java** (Annotation-based component in com.example.di.demo.component)  
  // src/main/java/com/example/di/demo/component/SmsSender.java  
  package com.example.di.demo.component;  
    
  import org.springframework.stereotype.Component;  
    
  @Component("smsSender")  
  public class SmsSender implements MessageSender {  
   @Override  
   public void sendMessage(String message) {  
   System.out.println("SMS Sent: " + message);  
   }  
  }
* **NotificationService.java** (Service with DI in com.example.di.demo.service)  
  // src/main/java/com/example/di/demo/service/NotificationService.java  
  package com.example.di.demo.service;  
    
  import com.example.di.demo.component.MessageSender;  
  import org.springframework.beans.factory.annotation.Autowired;  
  import org.springframework.beans.factory.annotation.Qualifier;  
  import org.springframework.stereotype.Service;  
    
  @Service // Marks as Spring service component  
  public class NotificationService {  
    
   private MessageSender sender; // Dependency  
    
   // --- Constructor Injection (Recommended for mandatory dependencies) ---  
   @Autowired // Tells Spring to inject dependencies via this constructor  
   // @Qualifier("emailSender") Specifies which MessageSender bean to inject  
   public NotificationService(@Qualifier("emailSender") MessageSender sender) {  
   this.sender = sender;  
   System.out.println("NotificationService created via Constructor Injection with " + sender.getClass().getSimpleName());  
   }  
    
   // --- Setter Injection (Optional dependencies, or when changing at runtime) ---  
   // Uncomment to try Setter Injection instead of or in addition to constructor injection  
   /\*  
   private MessageSender secondarySender;  
    
   @Autowired  
   @Qualifier("smsSender")  
   public void setSecondarySender(MessageSender secondarySender) {  
   this.secondarySender = secondarySender;  
   System.out.println("Secondary sender set via Setter Injection with " + secondarySender.getClass().getSimpleName());  
   }  
   \*/  
    
   // --- Field Injection (Least recommended, but common for quick demos) ---  
   // Uncomment to try Field Injection (and comment out constructor/setter @Autowired)  
   /\*  
   @Autowired  
   @Qualifier("emailSender")  
   private MessageSender fieldInjectedSender;  
   \*/  
    
   public void sendNotification(String message) {  
   sender.sendMessage(message);  
   // if (secondarySender != null) secondarySender.sendMessage("Secondary: " + message);  
   // if (fieldInjectedSender != null) fieldInjectedSender.sendMessage("Field Injected: " + message);  
   }  
  }

#### STEP 3: Spring Configuration (Annotations & Java Config)

We'll use @ComponentScan for annotation-based wiring and @Bean methods for Java Config wiring.

* **AppConfig.java** (in com.example.di.demo.config)  
  // src/main/java/com/example/di/demo/config/AppConfig.java

package com.example.di.demo.config;

import com.example.di.demo.component.MessageSender;

import com.example.di.demo.component.SmsSender; // Import SmsSender for Java Config bean

import com.example.di.demo.service.NotificationService; // Import NotificationService for Java Config bean

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope; // For bean scopes

import org.springframework.beans.factory.annotation.Qualifier; // Import Qualifier

@Configuration // Marks this class as a Spring configuration source

@ComponentScan(basePackages = "com.example.di.demo") // Scans for @Component, @Service, etc.

public class AppConfig {

// --- Java Config Method for Wiring Beans ---

// This method explicitly defines a bean named "javaConfigSmsSender"

// Spring will call this method to get an instance of SmsSender.

@Bean("javaConfigSmsSender")

public MessageSender smsSenderViaJavaConfig() {

System.out.println("Creating SmsSender via Java Config @Bean method.");

return new SmsSender();

}

// This method defines a NotificationService bean that uses the smsSenderViaJavaConfig.

// Spring automatically resolves the dependency 'sender' by looking for a matching bean.

@Bean("javaConfigNotificationService")

// @Scope("prototype") // Uncomment to demonstrate prototype scope for this specific bean

public NotificationService notificationServiceViaJavaConfig(@Qualifier("javaConfigSmsSender") MessageSender sender) { // Added @Qualifier

System.out.println("Creating NotificationService via Java Config @Bean method.");

// We are explicitly passing the 'javaConfigSmsSender' bean here.

return new NotificationService(sender);

}

// --- Demonstrating Bean Scopes with @Bean ---

// By default, @Bean methods create singleton beans.

// To make it prototype, use @Scope("prototype")

@Bean("prototypeSmsSender")

@Scope("prototype") // Each request for this bean will return a new instance

public MessageSender prototypeSmsSender() {

System.out.println("Creating a NEW prototype SmsSender.");

return new SmsSender();

}

}

#### STEP 4: Main Application Class

This class will start the Spring IoC container and demonstrate retrieving beans wired using both annotations and Java Config, and observe bean scopes.

* **DiDemoApplication.java** (in com.example.di.demo)  
  // src/main/java/com/example/di/demo/DiDemoApplication.java

package com.example.di.demo;

import com.example.di.demo.component.MessageSender;

import com.example.di.demo.config.AppConfig;

import com.example.di.demo.service.NotificationService;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class DiDemoApplication {

public static void main(String[] args) {

System.out.println("--- Starting Spring IoC Container ---");

// Initialize Spring container with Java-based configuration

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

System.out.println("\n--- Demonstrating Annotation-based Wiring (Autowired & ComponentScan) ---");

// Retrieve the NotificationService bean.

// Spring finds it via @ComponentScan and injects EmailSender via @Autowired @Qualifier("emailSender").

// Explicitly request by bean name to avoid NoUniqueBeanDefinitionException

NotificationService annotationNotificationService = (NotificationService) context.getBean("notificationService"); // Changed to retrieve by name

annotationNotificationService.sendNotification("Hello from Annotation-wired service!");

System.out.println("\n--- Demonstrating Java Config Wiring (@Bean methods) ---");

// Retrieve the NotificationService bean wired via Java Config.

// It's explicitly named "javaConfigNotificationService" in AppConfig.

NotificationService javaConfigNotificationService = (NotificationService) context.getBean("javaConfigNotificationService");

javaConfigNotificationService.sendNotification("Hello from Java Config-wired service!");

System.out.println("\n--- Demonstrating Bean Scopes ---");

// --- Singleton Scope (Default for @Component/@Service and @Bean) ---

// Retrieve the EmailSender bean. It's a singleton by default.

MessageSender emailSender1 = (MessageSender) context.getBean("emailSender");

MessageSender emailSender2 = (MessageSender) context.getBean("emailSender");

System.out.println("EmailSender instances are the same (Singleton)? " + (emailSender1 == emailSender2)); // Should be true

// --- Prototype Scope (Explicitly defined with @Scope("prototype")) ---

// Retrieve the prototypeSmsSender bean. Each call gets a new instance.

MessageSender prototypeSmsSender1 = (MessageSender) context.getBean("prototypeSmsSender");

MessageSender prototypeSmsSender2 = (MessageSender) context.getBean("prototypeSmsSender");

System.out.println("PrototypeSmsSender instances are the same (Prototype)? " + (prototypeSmsSender1 == prototypeSmsSender2)); // Should be false

}

System.out.println("\n--- Spring IoC Container Shut Down ---");

System.out.println("Activity Complete! You've explored DI basics, wiring methods, and bean scopes.");

}

}

### How to Run the Application:

1. **Run from your IDE:**
   * Open the DiDemoApplication.java file in your IDE.
   * Right-click within the file and select "Run 'DiDemoApplication.main()'" or click the green play button next to the main method.

You will see console output demonstrating the creation of beans, the injection of dependencies using both annotation-based and Java-based configuration, and the difference in behavior between singleton and prototype scoped beans.

# Activity 3.1: Spring ORM JPA “Product Management”

This activity will guide you through building a simple Spring Boot application to manage products using Spring ORM JPA and an in-memory H2 database.

### Goal:

To understand how Spring ORM JPA simplifies data access by creating a Product entity, a ProductRepository interface, and interacting with it through a service layer.

### Step 1: Project Setup with Spring Initializr

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure your Project:**

* **Project:** Maven Project (or Gradle Project if preferred)
* **Language:** Java
* **Spring Boot:** Choose the latest stable version (e.g., 3.3.1 or newer).
* **Project Metadata:**
* **Group:** com.example
* **Artifact:** spring-data-demo
* **Name:** spring-data-demo
* **Package Name:** com.example.springdata
* **Packaging:** Jar
* **Java:** 17 (or your preferred LTS version)

1. **Add Dependencies:** In the "Dependencies" section, search for and add the following:

* Spring Web (for building RESTful APIs)
* Spring Data JPA (for ORM and repository abstraction)
* H2 Database (an in-memory database for easy setup and testing)
* Lombok (optional, but highly recommended for reducing boilerplate code like getters/setters)

1. **Generate and Download:** Click the "Generate" button. This will download a .zip file containing your project.
2. **Unzip and Open:** Unzip the downloaded file and open the project in your favorite IDE (IntelliJ IDEA, VS Code, Eclipse, etc.).

### Step 2: Create the Product Entity

The Product entity will represent a row in our database table.

1. **Create Package:** Inside src/main/java/com/example/springdata, create a new package named entity.
2. **Create Product.java:** Inside the entity package, create a new Java class named Product.java and add the following code:  
   // src/main/java/com/example/springdata/entity/Product.java  
   package com.example.springdata.entity;  
     
   import jakarta.persistence.Entity; // JPA annotation to mark this class as an entity  
   import jakarta.persistence.GeneratedValue; // For automatic ID generation  
   import jakarta.persistence.GenerationType; // Strategy for ID generation  
   import jakarta.persistence.Id; // Marks the primary key field  
   import lombok.AllArgsConstructor; // Lombok: Generates a constructor with all fields  
   import lombok.Data; // Lombok: Generates getters, setters, toString, equals, hashCode  
   import lombok.NoArgsConstructor; // Lombok: Generates a no-argument constructor  
     
   @Entity // Marks this class as a JPA entity, mapping it to a database table  
   @Data // Lombok annotation to auto-generate getters, setters, toString(), equals(), and hashCode()  
   @NoArgsConstructor // Lombok annotation to auto-generate a no-argument constructor  
   @AllArgsConstructor // Lombok annotation to auto-generate a constructor with all fields  
   public class Product {  
     
       @Id // Marks this field as the primary key  
       @GeneratedValue(strategy = GenerationType.IDENTITY) // Configures ID generation strategy (auto-increment)  
       private Long id;  
       private String name;  
       private String description;  
       private double price;  
       private int quantity;  
   }

* **Explanation:**
* @Entity: Tells JPA that this class is a database entity.
* @Id: Marks id as the primary key.
* @GeneratedValue(strategy = GenerationType.IDENTITY): Configures the database to automatically generate the id for new products.
* @Data, @NoArgsConstructor, @AllArgsConstructor: Lombok annotations that drastically reduce boilerplate code by generating constructors, getters, setters, toString(), equals(), and hashCode() methods automatically.

### Step 3: Create the ProductRepository Interface

This interface will extend Spring Data JPA's JpaRepository to provide automatic CRUD operations and derived query methods.

1. **Create Package:** Inside src/main/java/com.example/springdata, create a new package named repository.
2. **Create ProductRepository.java:** Inside the repository package, create a new Java interface named ProductRepository.java and add the following code:  
   // src/main/java/com/example/springdata/repository/ProductRepository.java  
   package com.example.springdata.repository;  
     
   import com.example.springdata.entity.Product; // Import the Product entity  
   import org.springframework.data.jpa.repository.JpaRepository; // Spring Data JPA's core repository interface  
   import org.springframework.data.jpa.repository.Query; // Import for @Query annotation  
   import org.springframework.data.repository.query.Param; // Import for @Param annotation  
   import org.springframework.stereotype.Repository; // Optional: for clarity, marks this as a repository component  
     
   import java.util.List;  
   import java.util.Optional;  
     
   @Repository // Indicates that this interface is a "repository" component  
   // JpaRepository<Entity, ID\_Type> provides methods for CRUD operations and more  
   public interface ProductRepository extends JpaRepository<Product, Long> {  
     
       // --- Derived Query Methods ---  
     
       // Find products by name (Spring Data generates the query automatically)  
       Optional<Product> findByName(String name);  
     
       // Find products by price less than a given value  
       List<Product> findByPriceLessThan(double price);  
     
       // Find products by quantity greater than a given value  
       List<Product> findByQuantityGreaterThan(int quantity);  
     
       // Find products by name containing a specific string (case-insensitive)  
       List<Product> findByNameContainingIgnoreCase(String keyword);  
     
       // --- Custom Query with @Query (JPQL) ---  
     
       // Find products with price between a min and max value  
       // Using named parameters (:minPrice, :maxPrice) for better readability  
       @Query("SELECT p FROM Product p WHERE p.price BETWEEN :minPrice AND :maxPrice")  
       List<Product> findProductsByPriceRange(@Param("minPrice") double minPrice, @Param("maxPrice") double maxPrice);  
     
       // Find products by description containing a keyword, ordered by name  
       @Query("SELECT p FROM Product p WHERE p.description LIKE %:keyword% ORDER BY p.name ASC")  
       List<Product> findProductsByDescriptionContainingOrderByPriceAsc(@Param("keyword") String keyword);  
     
       // Find products with quantity less than a value, and price greater than a value  
       @Query("SELECT p FROM Product p WHERE p.quantity < ?1 AND p.price > ?2")  
       List<Product> findProductsLowStockHighPrice(int maxQuantity, double minPrice);  
   }

* **Explanation:**
* @Repository: A stereotype annotation that makes this interface a Spring component.
* JpaRepository<Product, Long>: By extending this, ProductRepository automatically inherits methods like save(), findById(), findAll(), deleteById(), etc., for the Product entity with Long as its ID type.
* findByName(), findByPriceLessThan(), etc.: These are "derived query methods." Spring Data automatically generates the SQL queries based on the method names and the entity's properties. You don't write any SQL!
* @Query: This new annotation allows you to write custom JPQL (Java Persistence Query Language) or native SQL queries directly.
* SELECT p FROM Product p WHERE p.price BETWEEN :minPrice AND :maxPrice: This is a JPQL query. p refers to the Product entity. :minPrice and :maxPrice are named parameters, which are mapped to method arguments using @Param.
* ?1, ?2: These are indexed parameters, referring to the first and second method arguments respectively.

### Step 4: Create the ProductService

The service layer will encapsulate business logic and interact with the ProductRepository.

1. **Create Package:** Inside src/main/java/com.example/springdata, create a new package named service.
2. **Create ProductService.java:** Inside the service package, create a new Java class named ProductService.java and add the following code:  
   // src/main/java/com/example/springdata/service/ProductService.java  
   package com.example.springdata.service;  
     
   import com.example.springdata.entity.Product; // Import Product entity  
   import com.example.springdata.repository.ProductRepository; // Import ProductRepository  
   import org.springframework.stereotype.Service; // Marks this class as a service component  
   import org.springframework.transaction.annotation.Transactional; // For transaction management  
     
   import java.util.List;  
   import java.util.Optional;  
     
   @Service // Indicates that this class is a "service" component  
   public class ProductService {  
     
       private final ProductRepository productRepository; // Inject the repository  
     
       // Constructor-based dependency injection (recommended)  
       public ProductService(ProductRepository productRepository) {  
           this.productRepository = productRepository;  
       }  
     
       // --- CRUD Operations ---  
     
       @Transactional // Ensures this method runs within a database transaction  
       public Product createProduct(Product product) {  
           // Business logic can be added here before saving  
           return productRepository.save(product); // Save the product  
       }  
     
       public Optional<Product> getProductById(Long id) {  
           return productRepository.findById(id); // Find product by ID  
       }  
     
       public List<Product> getAllProducts() {  
           return productRepository.findAll(); // Get all products  
       }  
     
       @Transactional  
       public Product updateProduct(Long id, Product updatedProduct) {  
           // Find the existing product  
           Optional<Product> existingProductOptional = productRepository.findById(id);  
     
           if (existingProductOptional.isPresent()) {  
               Product existingProduct = existingProductOptional.get();  
               // Update fields  
               existingProduct.setName(updatedProduct.getName());  
               existingProduct.setDescription(updatedProduct.getDescription());  
               existingProduct.setPrice(updatedProduct.getPrice());  
               existingProduct.setQuantity(updatedProduct.getQuantity());  
               // Save the updated product (Spring Data will update if ID exists)  
               return productRepository.save(existingProduct);  
           } else {  
               throw new RuntimeException("Product not found with ID: " + id);  
           }  
       }  
     
       @Transactional  
       public void deleteProduct(Long id) {  
           productRepository.deleteById(id); // Delete product by ID  
       }  
     
       // --- Using Derived Query Methods from Repository ---  
     
       public Optional<Product> getProductByName(String name) {  
           return productRepository.findByName(name);  
       }  
     
       public List<Product> getProductsCheaperThan(double price) {  
           return productRepository.findByPriceLessThan(price);  
       }  
     
       public List<Product> getProductsInStockGreaterThan(int quantity) {  
           return productRepository.findByQuantityGreaterThan(quantity);  
       }  
     
       public List<Product> searchProductsByName(String keyword) {  
           return productRepository.findByNameContainingIgnoreCase(keyword);  
       }  
     
       // --- Using Custom Query Methods with @Query (JPQL) ---  
     
       public List<Product> getProductsByPriceRange(double minPrice, double maxPrice) {  
           return productRepository.findProductsByPriceRange(minPrice, maxPrice);  
       }  
     
       public List<Product> getProductsByDescriptionContaining(String keyword) {  
           return productRepository.findProductsByDescriptionContainingOrderByPriceAsc(keyword);  
       }  
     
       public List<Product> getProductsLowStockHighPrice(int maxQuantity, double minPrice) {  
           return productRepository.findProductsLowStockHighPrice(maxQuantity, minPrice);  
       }  
   }

* **Explanation:**
* @Service: Marks this class as a Spring service, making it eligible for component scanning and dependency injection.
* @Autowired (implicitly by constructor injection): Spring automatically injects an instance of ProductRepository into ProductService.
* @Transactional: Ensures that the methods are executed within a database transaction. If an unchecked exception occurs, the transaction will be rolled back.
* New methods getProductsByPriceRange, getProductsByDescriptionContaining, and getProductsLowStockHighPrice are added to expose the custom JPQL queries defined in the repository.

### Step 5: Configure H2 Database

We need to tell Spring Boot to use the H2 in-memory database and enable its console for easy viewing.

1. **Open application.properties:** Navigate to src/main/resources/application.properties.
2. **Add Configuration:** Add the following lines to the file:  
   # src/main/resources/application.properties  
     
   # H2 Database Configuration  
   spring.h2.console.enabled=true  
   spring.h2.console.path=/h2-console  
   spring.datasource.url=jdbc:h2:mem:testdb  
   spring.datasource.driverClassName=org.h2.Driver  
   spring.datasource.username=sa  
   spring.datasource.password=password  
     
   # JPA Properties  
   spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
   spring.jpa.hibernate.ddl-auto=update # Automatically create/update schema based on entities  
   spring.jpa.show-sql=true # Show SQL queries in console  
   spring.jpa.properties.hibernate.format\_sql=true # Format SQL for readability

* **Explanation:**
* spring.h2.console.enabled=true: Enables the H2 web console.
* spring.h2.console.path=/h2-console: Sets the URL path for the console.
* spring.datasource.url=jdbc:h2:mem:testdb: Configures H2 as an in-memory database named testdb. The data will be lost when the application stops.
* spring.jpa.hibernate.ddl-auto=update: Hibernate (the JPA implementation) will automatically create or update the database schema based on your Product entity. For production, validate or none is often preferred.
* spring.jpa.show-sql=true: Logs the SQL statements executed by JPA.

### Step 6: Create a Simple REST Controller (Optional, but useful for testing)

This controller will expose REST endpoints to interact with our ProductService.

1. **Create Package:** Inside src/main/java/com.example/springdata, create a new package named controller.
2. **Create ProductController.java:** Inside the controller package, create a new Java class named ProductController.java and add the following code:  
   // src/main/java/com/example/springdata/controller/ProductController.java  
   package com.example.springdata.controller;  
     
   import com.example.springdata.entity.Product; // Import Product entity  
   import com.example.springdata.service.ProductService; // Import ProductService  
   import org.springframework.http.HttpStatus; // HTTP status codes  
   import org.springframework.http.ResponseEntity; // For building HTTP responses  
   import org.springframework.web.bind.annotation.\*; // REST annotations  
     
   import java.util.List;  
   import java.util.Optional;  
     
   @RestController // Marks this class as a REST controller  
   @RequestMapping("/api/products") // Base URL path for all endpoints in this controller  
   public class ProductController {  
     
       private final ProductService productService; // Inject the service  
     
       public ProductController(ProductService productService) {  
           this.productService = productService;  
       }  
     
       // POST /api/products - Create a new product  
       @PostMapping  
       public ResponseEntity<Product> createProduct(@RequestBody Product product) {  
           Product createdProduct = productService.createProduct(product);  
           return new ResponseEntity<>(createdProduct, HttpStatus.CREATED); // Return 201 Created  
       }  
     
       // GET /api/products/{id} - Get a product by ID  
       @GetMapping("/{id}")  
       public ResponseEntity<Product> getProductById(@PathVariable Long id) {  
           Optional<Product> product = productService.getProductById(id);  
           return product.map(p -> new ResponseEntity<>(p, HttpStatus.OK)) // Return 200 OK if found  
                         .orElse(new ResponseEntity<>(HttpStatus.NOT\_FOUND)); // Return 404 Not Found if not  
       }  
     
       // GET /api/products - Get all products  
       @GetMapping  
       public ResponseEntity<List<Product>> getAllProducts() {  
           List<Product> products = productService.getAllProducts();  
           return new ResponseEntity<>(products, HttpStatus.OK); // Return 200 OK  
       }  
     
       // PUT /api/products/{id} - Update an existing product  
       @PutMapping("/{id}")  
       public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product product) {  
           try {  
               Product updated = productService.updateProduct(id, product);  
               return new ResponseEntity<>(updated, HttpStatus.OK); // Return 200 OK  
           } catch (RuntimeException e) {  
               return new ResponseEntity<>(HttpStatus.NOT\_FOUND); // Return 404 if product not found  
           }  
       }  
     
       // DELETE /api/products/{id} - Delete a product  
       @DeleteMapping("/{id}")  
       public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {  
           productService.deleteProduct(id);  
           return new ResponseEntity<>(HttpStatus.NO\_CONTENT); // Return 204 No Content  
       }  
     
       // --- Endpoints using Derived Query Methods ---  
     
       // GET /api/products/search/name?name={productName}  
       @GetMapping("/search/name")  
       public ResponseEntity<Product> getProductByName(@RequestParam String name) {  
           Optional<Product> product = productService.getProductByName(name);  
           return product.map(p -> new ResponseEntity<>(p, HttpStatus.OK))  
                         .orElse(new ResponseEntity<>(HttpStatus.NOT\_FOUND));  
       }  
     
       // GET /api/products/search/cheaper-than?price={maxPrice}  
       @GetMapping("/search/cheaper-than")  
       public ResponseEntity<List<Product>> getProductsCheaperThan(@RequestParam double price) {  
           List<Product> products = productService.getProductsCheaperThan(price);  
           return new ResponseEntity<>(products, HttpStatus.OK);  
       }  
     
       // GET /api/products/search/in-stock-greater-than?quantity={minQuantity}  
       @GetMapping("/search/in-stock-greater-than")  
       public ResponseEntity<List<Product>> getProductsInStockGreaterThan(@RequestParam int quantity) {  
           List<Product> products = productService.getProductsInStockGreaterThan(quantity);  
           return new ResponseEntity<>(products, HttpStatus.OK);  
       }  
     
       // GET /api/products/search/keyword?keyword={searchKeyword}  
       @GetMapping("/search/keyword")  
       public ResponseEntity<List<Product>> searchProductsByName(@RequestParam String keyword) {  
           List<Product> products = productService.searchProductsByName(keyword);  
           return new ResponseEntity<>(products, HttpStatus.OK);  
       }  
     
       // --- New Endpoints using Custom @Query (JPQL) Methods ---  
     
       // GET /api/products/custom/price-range?minPrice={min}&maxPrice={max}  
       @GetMapping("/custom/price-range")  
       public ResponseEntity<List<Product>> getProductsByPriceRange(  
               @RequestParam double minPrice, @RequestParam double maxPrice) {  
           List<Product> products = productService.getProductsByPriceRange(minPrice, maxPrice);  
           return new ResponseEntity<>(products, HttpStatus.OK);  
       }  
     
       // GET /api/products/custom/description-keyword?keyword={keyword}  
       @GetMapping("/custom/description-keyword")  
       public ResponseEntity<List<Product>> getProductsByDescriptionContaining(  
               @RequestParam String keyword) {  
           List<Product> products = productService.getProductsByDescriptionContaining(keyword);  
           return new ResponseEntity<>(products, HttpStatus.OK);  
       }  
     
       // GET /api/products/custom/low-stock-high-price?maxQuantity={maxQ}&minPrice={minP}  
       @GetMapping("/custom/low-stock-high-price")  
       public ResponseEntity<List<Product>> getProductsLowStockHighPrice(  
               @RequestParam int maxQuantity, @RequestParam double minPrice) {  
           List<Product> products = productService.getProductsLowStockHighPrice(maxQuantity, minPrice);  
           return new ResponseEntity<>(products, HttpStatus.OK);  
       }  
   }

* **Explanation:**
* @RestController: Combines @Controller and @ResponseBody, meaning methods return data directly (e.g., JSON) rather than view names.
* @RequestMapping("/api/products"): Sets the base path for all endpoints in this controller.
* @PostMapping, @GetMapping, @PutMapping, @DeleteMapping: Map HTTP methods to specific controller methods.
* @RequestBody: Maps the HTTP request body (e.g., JSON) to a Java object (Product).
* @PathVariable: Extracts a variable from the URL path (e.g., {id}).
* @RequestParam: Extracts a parameter from the query string (e.g., ?name=value).
* ResponseEntity: Provides full control over the HTTP response (status code, headers, body).
* New endpoints under /api/products/custom/ are added to test the JPQL queries.

### Step 7: Run the Application and Test

1. **Run Spring Boot Application:**

* Locate the main application class: src/main/java/com/example/springdata/SpringDataDemoApplication.java.
* Right-click on it and select "Run 'SpringDataDemoApplication.main()'" (or use your IDE's run button).
* Look for "Started SpringDataDemoApplication" in the console output.

1. **Access H2 Console:**

* Once the application is running, open your web browser and go to: http://localhost:8080/h2-console
* **JDBC URL:** Ensure it's jdbc:h2:mem:testdb (matching your application.properties).
* **User Name:** sa
* **Password:** password
* Click "Connect". You should see the H2 console, and if you execute SELECT \* FROM PRODUCT;, you'll see an an empty table (or data if you've already added some).

1. **Test with a Tool (e.g., Postman, Insomnia, or curl):**

* **Create Product (POST):**
* **URL:** http://localhost:8080/api/products
* **Method:** POST
* **Headers:** Content-Type: application/json
* **Body (Raw JSON):**  
  {  
      "name": "Laptop",  
      "description": "High-performance laptop",  
      "price": 1200.00,  
      "quantity": 50  
  }
* Send the request. You should get a 201 Created response with the created product details (including the generated ID).
* **Create Another Product (POST):**  
  {  
      "name": "Mouse",  
      "description": "Wireless ergonomic mouse",  
      "price": 25.50,  
      "quantity": 200  
  }
* **Create a Third Product (POST):**  
  {  
      "name": "Keyboard",  
      "description": "Mechanical gaming keyboard",  
      "price": 150.00,  
      "quantity": 75  
  }
* **Get All Products (GET):**
* **URL:** http://localhost:8080/api/products
* **Method:** GET
* Send the request. You should get a 200 OK response with a list of all products.
* **Get Product by ID (GET):** (Replace 1 with the actual ID from your POST response)
* **URL:** http://localhost:8080/api/products/1
* **Method:** GET
* Send the request. You should get a 200 OK response with the specific product.
* **Update Product (PUT):** (Replace 1 with the actual ID)
* **URL:** http://localhost:8080/api/products/1
* **Method:** PUT
* **Headers:** Content-Type: application/json
* **Body (Raw JSON):**  
  {  
      "id": 1,  
      "name": "Gaming Laptop",  
      "description": "Ultra-performance gaming laptop with RTX 4080",  
      "price": 1800.00,  
      "quantity": 45  
  }
* Send the request. You should get a 200 OK response with the updated product.
* **Search Products by Name (GET - Derived Query):**
* **URL:** http://localhost:8080/api/products/search/name?name=Mouse
* **Method:** GET
* Send the request. You should get the product named "Mouse".
* **Search Products Cheaper Than (GET - Derived Query):**
* **URL:** http://localhost:8080/api/products/search/cheaper-than?price=100.00
* **Method:** GET
* Send the request. You should get products with a price less than $100.00.
* **Search Products In Stock Greater Than (GET - Derived Query):**
* **URL:** http://localhost:8080/api/products/search/in-stock-greater-than?quantity=60
* **Method:** GET
* Send the request. You should get products with quantity greater than 60.
* **Search Products by Keyword (GET - Derived Query):**
* **URL:** http://localhost:8080/api/products/search/keyword?keyword=gaming
* **Method:** GET
* Send the request. You should get products whose name or description contains "gaming" (case-insensitive).
* **Delete Product (DELETE):** (Replace 1 with the actual ID)
* **URL:** http://localhost:8080/api/products/1
* **Method:** DELETE
* Send the request. You should get a 204 No Content response.
* Verify by trying to GET the product by ID again, which should now return 404 Not Found.

### Step 8: Test Custom Queries with @Query (JPQL)

Now, let's test the new endpoints that use our custom JPQL queries. Make sure you have some products created (you can re-run the POST requests from Step 7).

* **Test Products by Price Range (GET - Custom JPQL Query):**
* **URL:** http://localhost:8080/api/products/custom/price-range?minPrice=100.00&maxPrice=1000.00
* **Method:** GET
* Send the request. You should get products whose price is between $100 and $1000 (e.g., "Keyboard").
* **Test Products by Description Keyword (GET - Custom JPQL Query):**
* **URL:** http://localhost:8080/api/products/custom/description-keyword?keyword=wireless
* **Method:** GET
* Send the request. You should get products whose description contains "wireless" (e.g., "Mouse").
* **Test Products Low Stock High Price (GET - Custom JPQL Query):**
* **URL:** http://localhost:8080/api/products/custom/low-stock-high-price?maxQuantity=100&minPrice=50.00
* **Method:** GET
* Send the request. This should return products with quantity less than 100 AND price greater than $50 (e.g., "Keyboard" if its quantity is below 100)

# Activity 4.1: Spring AOP “Logging with Aspects”

This activity will guide you through understanding and implementing Aspect-Oriented Programming (AOP) in a Spring Boot application. We will use AOP to add logging functionality to a UserService without modifying its core business logic.

### STEP 1: Project Setup (Spring Boot)

We'll use Spring Initializr to set up a new Spring Boot project.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.aop.app
   * **Artifact:** aop-demo
   * **Name:** aop-demo
   * **Description:** Spring AOP Logging Demo
   * **Package Name:** com.example.aop
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** For a basic REST controller to trigger our service.
   * **Spring AOP:** Essential for AOP functionality.

**Note:** If you cannot find "Spring AOP" directly in the search, you can generate the project without it and add it manually in the next step.

* + **Lombok:** (Optional but recommended) Reduces boilerplate code.

1. **Generate and Download:** Click the "Generate" button. Download the .zip file.
2. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).
3. **Manually Add Spring AOP Dependency to pom.xml (If not added via Initializr):**

Open the pom.xml file in your project root.

Locate the <dependencies> section.

Add the following dependency. This starter pulls in all necessary AOP-related libraries.

<dependency>

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

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

</dependency>

After adding the dependency, refresh your Maven project in your IDE (e.g., in IntelliJ, right-click on pom.xml and select "Maven" -> "Sync Project"). This will download the new dependency.

### STEP 2: Define the User Service (Target Object)

This will be our core business logic, which we want to add logging to using AOP.

* Create a new package com.example.aop.model in src/main/java/.

Create a simple User.java model class:  
// src/main/java/com/example/aop/model/User.java  
package com.example.aop.model;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
/\*\*  
 \* Simple User model class.  
 \*/  
@Data // Lombok: Generates getters, setters, toString, equals, hashCode  
@NoArgsConstructor // Lombok: Generates no-argument constructor  
@AllArgsConstructor // Lombok: Generates constructor with all fields  
public class User {  
    private Long id;  
    private String name;  
    private String email;  
}

This will be our core business logic, which we want to add logging to using AOP.

* Create a new package com.example.aop.service in src/main/java/.
* Create an interface UserService.java in this package:  
  // src/main/java/com/example/aop/service/UserService.java  
  package com.example.aop.service;  
    
  import com.example.aop.model.User; // Assuming User model will be created later  
    
  import java.util.List;  
    
  public interface UserService {  
   User createUser(String name, String email);  
   User getUserById(Long id);  
   List<User> getAllUsers();  
   void deleteUser(Long id);  
   User updateUserEmail(Long id, String newEmail);  
   void throwExceptionMethod(); // Method to test @AfterThrowing  
  }
* Create a class UserServiceImpl.java implementing the interface in the same package:  
  // src/main/java/com/example/aop/service/UserServiceImpl.java  
  package com.example.aop.service;  
    
  import com.example.aop.model.User;  
  import org.springframework.stereotype.Service;  
    
  import java.util.ArrayList;  
  import java.util.List;  
  import java.util.Map;  
  import java.util.concurrent.ConcurrentHashMap;  
  import java.util.concurrent.atomic.AtomicLong;  
    
  /\*\*  
   \* Simple in-memory implementation of UserService.  
   \* This is our "target object" that AOP will advise.  
   \*/  
  @Service // Marks this as a Spring service component  
  public class UserServiceImpl implements UserService {  
    
   private final Map<Long, User> users = new ConcurrentHashMap<>();  
   private final AtomicLong idCounter = new AtomicLong();  
    
   @Override  
   public User createUser(String name, String email) {  
   Long id = idCounter.incrementAndGet();  
   User user = new User(id, name, email);  
   users.put(id, user);  
   System.out.println("UserService: Created user " + user.getName());  
   return user;  
   }  
    
   @Override  
   public User getUserById(Long id) {  
   User user = users.get(id);  
   if (user == null) {  
   System.out.println("UserService: User with ID " + id + " not found.");  
   } else {  
   System.out.println("UserService: Retrieved user " + user.getName());  
   }  
   return user;  
   }  
    
   @Override  
   public List<User> getAllUsers() {  
   System.out.println("UserService: Retrieving all users.");  
   return new ArrayList<>(users.values());  
   }  
    
   @Override  
   public void deleteUser(Long id) {  
   User removedUser = users.remove(id);  
   if (removedUser != null) {  
   System.out.println("UserService: Deleted user with ID " + id);  
   } else {  
   System.out.println("UserService: User with ID " + id + " not found for deletion.");  
   }  
   }  
    
   @Override  
   public User updateUserEmail(Long id, String newEmail) {  
   User user = users.get(id);  
   if (user != null) {  
   user.setEmail(newEmail);  
   System.out.println("UserService: Updated email for user " + user.getName() + " to " + newEmail);  
   } else {  
   System.out.println("UserService: User with ID " + id + " not found for update.");  
   }  
   return user;  
   }  
    
   @Override  
   public void throwExceptionMethod() {  
   System.out.println("UserService: Attempting to throw an exception...");  
   throw new RuntimeException("Simulated exception from throwExceptionMethod!");  
   }  
  }

### STEP 3: Declare an Aspect (LoggingAspect)

This class will contain our logging advice.

* Create a new package com.example.aop.aspect in src/main/java/.
* Create a Java class named LoggingAspect.java inside this package:  
  // src/main/java/com/example/aop/aspect/LoggingAspect.java  
  package com.example.aop.aspect;  
    
  import org.aspectj.lang.JoinPoint; // Represents a method execution  
  import org.aspectj.lang.ProceedingJoinPoint; // For @Around advice  
  import org.aspectj.lang.annotation.\*; // AOP annotations  
  import org.springframework.stereotype.Component;  
    
  /\*\*  
   \* This class defines our logging aspect.  
   \* @Aspect: Marks this class as an Aspect.  
   \* @Component: Makes this class a Spring-managed bean, so Spring can detect it as an aspect.  
   \*/  
  @Aspect  
  @Component  
  public class LoggingAspect {  
    
   // --- Pointcut Expressions in Spring AOP ---  
   // A pointcut expression defines where the advice should be applied.  
   // It uses AspectJ pointcut designators.  
    
   // Pointcut to match all methods in UserService (any return type, any method name, any parameters)  
   // within the 'com.example.aop.service' package.  
   @Pointcut("execution(\* com.example.aop.service.UserService.\*(..))")  
   public void userServiceMethods() {}  
    
   // Pointcut to match any method in any class within the 'com.example.aop.service' package  
   // that has 'create' in its name.  
   @Pointcut("execution(\* com.example.aop.service.\*.create\*(..))")  
   public void createMethods() {}  
    
   // Pointcut to match any method in UserService that accepts a Long as its first parameter.  
   @Pointcut("execution(\* com.example.aop.service.UserService.\*(Long, ..))")  
   public void methodsWithLongId() {}  
    
   // --- Types of Advice ---  
    
   /\*\*  
   \* @Before Advice: Executes before the advised method.  
   \* It logs the method signature and arguments before execution.  
   \*/  
   @Before("userServiceMethods()") // Apply this advice before any method in UserService  
   public void logBefore(JoinPoint joinPoint) {  
   String methodName = joinPoint.getSignature().toShortString();  
   Object[] args = joinPoint.getArgs();  
   System.out.println("AOP @Before: Executing " + methodName + " with args: " + java.util.Arrays.toString(args));  
   }  
    
   /\*\*  
   \* @After Advice: Executes after the advised method, regardless of success or exception.  
   \*/  
   @After("userServiceMethods()") // Apply this advice after any method in UserService  
   public void logAfter(JoinPoint joinPoint) {  
   String methodName = joinPoint.getSignature().toShortString();  
   System.out.println("AOP @After: Finished execution of " + methodName);  
   }  
    
   /\*\*  
   \* @AfterReturning Advice: Executes only after the advised method returns successfully.  
   \* 'returning' attribute captures the return value.  
   \*/  
   @AfterReturning(pointcut = "userServiceMethods()", returning = "result")  
   public void logAfterReturning(JoinPoint joinPoint, Object result) {  
   String methodName = joinPoint.getSignature().toShortString();  
   System.out.println("AOP @AfterReturning: Method " + methodName + " returned: " + result);  
   }  
    
   /\*\*  
   \* @AfterThrowing Advice: Executes only if the advised method throws an exception.  
   \* 'throwing' attribute captures the thrown exception.  
   \*/  
   @AfterThrowing(pointcut = "userServiceMethods()", throwing = "exception")  
   public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) {  
   String methodName = joinPoint.getSignature().toShortString();  
   System.out.println("AOP @AfterThrowing: Method " + methodName + " threw exception: " + exception.getMessage());  
   }  
    
   /\*\*  
   \* @Around Advice: Executes around the advised method.  
   \* It gives full control over method execution (can prevent it, call it multiple times, etc.).  
   \* Must take a ProceedingJoinPoint as a parameter.  
   \*/  
   @Around("execution(\* com.example.aop.service.UserService.getAllUsers(..))")  
   public Object logAroundGetAllUsers(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  
   String methodName = proceedingJoinPoint.getSignature().toShortString();  
   long startTime = System.currentTimeMillis();  
   System.out.println("AOP @Around (Before): Starting " + methodName);  
    
   Object result = null;  
   try {  
   result = proceedingJoinPoint.proceed(); // Execute the target method  
   } catch (Throwable e) {  
   System.out.println("AOP @Around (Exception): " + methodName + " threw " + e.getMessage());  
   throw e; // Re-throw the exception  
   } finally {  
   long endTime = System.currentTimeMillis();  
   System.out.println("AOP @Around (After): Finished " + methodName + " in " + (endTime - startTime) + "ms");  
   }  
   return result;  
   }  
  }

### STEP 4: Enable AOP in Spring Boot Application

Spring Boot automatically enables AOP if spring-boot-starter-aop (which is pulled in by spring-aop dependency) is on the classpath. No explicit @EnableAspectJAutoProxy is usually needed.

* Open your main application class (e.g., AopDemoApplication.java in com.example.aop.app).
* Ensure it has @SpringBootApplication.  
  // src/main/java/com/example/aop/app/AopDemoApplication.java  
  package com.example.aop.app;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
    
  @SpringBootApplication  
  public class AopDemoApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(AopDemoApplication.class, args);  
   }  
  }

### STEP 5: Create a REST Controller to Trigger Service Methods

We'll create a simple REST controller to easily call our UserService methods and observe the AOP advice in action.

* Create a new package com.example.aop.controller in src/main/java/.
* Create a Java class named UserController.java inside this package:  
  // src/main/java/com/example/aop/controller/UserController.java  
  package com.example.aop.controller;  
    
  import com.example.aop.model.User;  
  import com.example.aop.service.UserService;  
  import org.springframework.beans.factory.annotation.Autowired;  
  import org.springframework.http.ResponseEntity;  
  import org.springframework.web.bind.annotation.\*;  
    
  import java.util.List;  
    
  @RestController  
  @RequestMapping("/users")  
  public class UserController {  
    
   @Autowired  
   private UserService userService; // Spring will inject the advised proxy of UserServiceImpl  
    
   @PostMapping  
   public ResponseEntity<User> createUser(@RequestParam String name, @RequestParam String email) {  
   User user = userService.createUser(name, email);  
   return ResponseEntity.ok(user);  
   }  
    
   @GetMapping("/{id}")  
   public ResponseEntity<User> getUser(@PathVariable Long id) {  
   User user = userService.getUserById(id);  
   return user != null ? ResponseEntity.ok(user) : ResponseEntity.notFound().build();  
   }  
    
   @GetMapping  
   public ResponseEntity<List<User>> getAllUsers() {  
   List<User> users = userService.getAllUsers();  
   return ResponseEntity.ok(users);  
   }  
    
   @DeleteMapping("/{id}")  
   public ResponseEntity<Void> deleteUser(@PathVariable Long id) {  
   userService.deleteUser(id);  
   return ResponseEntity.noContent().build();  
   }  
    
   @PutMapping("/{id}/email")  
   public ResponseEntity<User> updateUserEmail(@PathVariable Long id, @RequestParam String newEmail) {  
   User user = userService.updateUserEmail(id, newEmail);  
   return user != null ? ResponseEntity.ok(user) : ResponseEntity.notFound().build();  
   }  
    
   @GetMapping("/exception")  
   public ResponseEntity<String> testException() {  
   try {  
   userService.throwExceptionMethod();  
   return ResponseEntity.ok("Method executed without exception.");  
   } catch (RuntimeException e) {  
   return ResponseEntity.internalServerError().body("Method threw an exception: " + e.getMessage());  
   }  
   }  
  }

### STEP 6: Run the Application and Observe AOP in Action

1. **Run the Spring Boot Application:**
   * Open your main application class (AopDemoApplication.java).
   * Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
2. Test Endpoints (using Postman/Insomnia or curl):  
   As you send requests, carefully observe the console output from your Spring Boot application. You will see the AOP advice (logging messages) interleaved with your UserService's own output.
   * **Create User:**
     + URL: http://localhost:8080/users?name=Alice&email=alice@example.com
     + Method: POST
     + Observe @Before, @After, @AfterReturning advice.
   * **Get All Users:**
     + URL: http://localhost:8080/users
     + Method: GET
     + Observe the @Around advice's "Before" and "After" messages, including execution time.
   * **Get User by ID:**
     + URL: http://localhost:8080/users/1 (use an ID of a user you created)
     + Method: GET
     + Observe @Before, @After, @AfterReturning advice.
   * **Update User Email:**
     + URL: http://localhost:8080/users/1/email?newEmail=alice.new@example.com
     + Method: PUT
     + Observe @Before, @After, @AfterReturning advice.
   * **Delete User:**
     + URL: http://localhost:8080/users/1
     + Method: DELETE
     + Observe @Before, @After, @AfterReturning advice.
   * **Test Exception Handling (AfterThrowing):**
     + URL: http://localhost:8080/users/exception
     + Method: GET
     + Observe @Before, @After, and specifically @AfterThrowing advice.

You have successfully implemented and observed Spring AOP in action! You've seen how aspects can cleanly separate logging concerns from your core UserService logic, demonstrating the power of AOP for building modular and maintainable applications.

# Activity 5.1: Spring Expression Language (SpEL)

This activity will guide you through understanding and using Spring Expression Language (SpEL) in a Spring Boot application. SpEL is a powerful expression language that supports querying and manipulating an object graph at runtime.

### STEP 1: Project Setup (Spring Boot)

We'll use Spring Initializr to set up a new Spring Boot project.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.spel
   * **Artifact:** spel-demo
   * **Name:** spel-demo
   * **Description:** Spring SpEL Demonstration
   * **Package Name:** com.example.spel
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** For a basic REST controller to trigger our SpEL examples.
   * **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

### STEP 2: Define Configuration Properties

We'll use application.properties to define some values that we'll access using SpEL.

1. **Open application.properties:**
   * Navigate to src/main/resources/ and open application.properties.
2. **Add Custom Properties:**
   * Add the following properties:

# application.properties

app.name=SpEL Demo Application

app.version=1.0.0

app.owner=John Doe

app.default.message=This is a default message.

app.admin.email=admin@example.com

# List of items

app.items=Apple,Banana,Orange,Grape,Pineapple

# Define the regex patterns as properties

app.version.pattern=(1\\.0\\.0|2\\.0\\.0)

app.admin.email.pattern=admin@example\\.com

### STEP 3: Create a SpEL Demo Bean

This bean will demonstrate various SpEL features using the @Value annotation.

Create a new package com.example.spel.bean in src/main/java/.

Create a Java class named SpelDemoBean.java inside this package:  
// src/main/java/com/example/spel/bean/SpelDemoBean.java

package com.example.spel.bean;

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

import org.springframework.stereotype.Component;

import java.util.List;

import java.util.Map;

import java.util.Properties;

/\*\*

\* This bean demonstrates various SpEL features using the @Value annotation.

\*/

@Component // Marks this as a Spring-managed component

public class SpelDemoBean {

// --- 1. Basic SpEL Syntax and Usage ---

// Injecting a simple string literal

@Value("#{'Hello SpEL!'}")

private String greeting;

// Injecting a number literal

@Value("#{100}")

private int number;

// Injecting a boolean literal

@Value("#{true}")

private boolean isTrue;

// Injecting a value from application.properties

@Value("${app.name}")

private String appName;

// Injecting a value from system properties

@Value("#{systemProperties['java.version']}")

private String javaVersion;

// --- 2. Operators in SpEL ---

// Arithmetic operations

@Value("#{10 \* 5 + 2}")

private int arithmeticResult; // Expected: 52

// Relational operations

@Value("#{10 > 5}")

private boolean greaterThan; // Expected: true

// Logical operations

@Value("#{true and false}")

private boolean logicalAnd; // Expected: false

// Conditional (Ternary) operator - Corrected: Use '${app.owner}' to resolve property first

@Value("#{'${app.owner}' == 'John Doe' ? 'Owner is John' : 'Owner is not John'}")

private String ownerCheck;

// Elvis operator (null check)

// If 'nonExistentProperty' is null, use 'Default Value'

@Value("#{systemProperties['nonExistentProperty'] ?: 'Default Value'}")

private String elvisOperatorExample;

// Safe Navigation operator (null check for properties) - Corrected: Use '${app.owner}' to resolve property first

// If '${app.owner}' property exists, get its length, otherwise null

@Value("#{'${app.owner}'?.length()}")

private Integer ownerNameLength;

// Type operator (calling static methods/fields)

@Value("#{T(java.lang.Math).PI}")

private double piValue;

// String concatenation - Corrected: Use property placeholders for app.name and app.version

@Value("#{'App: ' + '${app.name}' + ' v' + '${app.version}'}")

private String appInfo;

// --- 3. Collections and Projections ---

// Injecting a list from properties (comma-separated)

@Value("#{'${app.items}'.split(',')}")

private List<String> appItemsList;

// Selection: Filter items longer than 5 characters

// The '${app.items}' resolves to a string, then .split(',') creates a list of strings.

// #this refers to each element in the list. This syntax is correct.

@Value("#{ '${app.items}'.split(',').?[#this.length() > 5] }")

private List<String> longItems; // Expected: [Banana, Orange, Pineapple]

// Projection: Convert all items to uppercase

// The '${app.items}' resolves to a string, then .split(',') creates a list of strings.

// #this refers to each element in the list. This syntax is correct.

@Value("#{ '${app.items}'.split(',').![#this.toUpperCase()] }")

private List<String> uppercaseItems; // Expected: [APPLE, BANANA, ORANGE, GRAPE, PINEAPPLE]

// Accessing elements by index

@Value("#{'${app.items}'.split(',')[0]}")

private String firstItem; // Expected: Apple

// --- 4. SpEL in Spring Framework Features (Demonstrated in Controller) ---

// We'll show an example of SpEL in a @RequestMapping annotation in the controller.

// Method to print all injected values

public void printSpelValues() {

System.out.println("\n--- SpEL Demo Bean Values ---");

System.out.println("Greeting: " + greeting);

System.out.println("Number: " + number);

System.out.println("Is True: " + isTrue);

System.out.println("App Name: " + appName);

System.out.println("Java Version: " + javaVersion);

System.out.println("Arithmetic Result (10 \* 5 + 2): " + arithmeticResult);

System.out.println("Greater Than (10 > 5): " + greaterThan);

System.out.println("Logical AND (true and false): " + logicalAnd);

System.out.println("Owner Check: " + ownerCheck);

System.out.println("Elvis Operator Example: " + elvisOperatorExample);

System.out.println("Owner Name Length: " + ownerNameLength);

System.out.println("PI Value: " + piValue);

System.out.println("App Info: " + appInfo);

System.out.println("App Items List: " + appItemsList);

System.out.println("Long Items (length > 5): " + longItems);

System.out.println("Uppercase Items: " + uppercaseItems);

System.out.println("First Item: " + firstItem);

System.out.println("-----------------------------");

}

}

### STEP 4: Create a REST Controller to Trigger SpEL Examples

We'll create a simple REST controller to trigger the SpelDemoBean and demonstrate SpEL in a @RequestMapping annotation.

Create a new package com.example.spel.controller in src/main/java/.

Create a Java class named SpelController.java inside this package:  
// src/main/java/com/example/spel/controller/SpelController.java

package com.example.spel.controller;

import com.example.spel.bean.SpelDemoBean;

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

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

import org.springframework.http.ResponseEntity;

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

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

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

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

import java.util.regex.Pattern; // Import Pattern for regex matching

/\*\*

\* REST Controller to demonstrate SpEL usage.

\*/

@RestController

@RequestMapping("/spel")

public class SpelController {

@Autowired

private SpelDemoBean spelDemoBean;

// Inject regex patterns from application.properties using @Value

@Value("${app.version.pattern}")

private String versionPatternString;

@Value("${app.admin.email.pattern}")

private String adminEmailPatternString;

// Compile patterns for efficient matching

private final Pattern versionPattern;

private final Pattern adminEmailPattern;

// Constructor to compile regex patterns

public SpelController(@Value("${app.version.pattern}") String versionPatternString,

@Value("${app.admin.email.pattern}") String adminEmailPatternString) {

this.versionPatternString = versionPatternString;

this.adminEmailPatternString = adminEmailPatternString;

this.versionPattern = Pattern.compile(versionPatternString);

this.adminEmailPattern = Pattern.compile(adminEmailPatternString);

}

/\*\*

\* Endpoint to trigger the printing of all SpEL values from SpelDemoBean.

\* GET /spel/demo

\*/

@GetMapping("/demo")

public ResponseEntity<String> runSpelDemo() {

spelDemoBean.printSpelValues();

return ResponseEntity.ok("SpEL demo values printed to console. Check server logs.");

}

/\*\*

\* Demonstrates how to use SpEL to inject properties into the controller

\* and then perform validation within the method.

\*

\* GET /spel/check-version/{version}

\* Example: /spel/check-version/1.0.0 (Matches)

\* Example: /spel/check-version/2.0.0 (Matches)

\* Example: /spel/check-version/3.0.0 (Will return a Bad Request, as it doesn't match the pattern)

\*/

@GetMapping("/check-version/{version}") // Simple path variable

public ResponseEntity<String> checkAppVersion(@PathVariable("version") String version) {

if (versionPattern.matcher(version).matches()) {

return ResponseEntity.ok("Version matched: " + version);

} else {

return ResponseEntity.badRequest().body("Version '" + version + "' does not match expected pattern: " + versionPatternString);

}

}

/\*\*

\* Another example of injecting a property and performing validation.

\*

\* GET /spel/is-admin/{email}

\* Example: /spel/is-admin/admin@example.com (Matches)

\* Example: /spel/is-admin/user@example.com (Will return a Bad Request)

\*/

@GetMapping("/is-admin/{email}") // Simple path variable

public ResponseEntity<String> isAdmin(@PathVariable("email") String email) {

if (adminEmailPattern.matcher(email).matches()) {

return ResponseEntity.ok("User " + email + " is an admin.");

} else {

return ResponseEntity.badRequest().body("User '" + email + "' is NOT an admin (does not match admin pattern: " + adminEmailPatternString + ").");

}

}

}

### STEP 5: Main Spring Boot Application Class

This is the standard Spring Boot application entry point.

* Open your main application class (e.g., SpelDemoApplication.java in com.example.spel.app).  
  // src/main/java/com/example/spel/app/SpelDemoApplication.java  
  package com.example.spel.app;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
    
  @SpringBootApplication  
  public class SpelDemoApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(SpelDemoApplication.class, args);  
   }  
  }

### STEP 6: Run the Application and Test SpEL

1. **Run the Spring Boot Application:**
   * Open your main application class (SpelDemoApplication.java).
   * Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
2. Test Endpoints (using Postman/Insomnia or curl):  
   Observe the console output from your Spring Boot application as you send requests.
   * **Run SpEL Demo Bean Values:**
     + URL: http://localhost:8080/spel/demo
     + Method: GET
     + **Observe:** The console output will show all the values injected into SpelDemoBean via @Value using various SpEL expressions. This demonstrates basic syntax, operators, and collection operations.
   * **Check App Version (SpEL in @RequestMapping):**
     + URL: http://localhost:8080/spel/check-version/1.0.0
     + Method: GET
     + Expected: 200 OK with "Version matched: 1.0.0"
     + URL: http://localhost:8080/spel/check-version/2.0.0
     + Method: GET
     + Expected: 200 OK with "Version matched: 2.0.0"
     + URL: http://localhost:8080/spel/check-version/3.0.0
     + Method: GET
     + Expected: 400 Bad Request because 3.0.0 does not match the SpEL pattern.
   * **Check Admin Email (SpEL in @RequestMapping):**
     + URL: http://localhost:8080/spel/is-admin/admin@example.com
     + Method: GET
     + Expected: 200 OK with "User admin@example.com is an admin."
     + URL: http://localhost:8080/spel/is-admin/another@example.com
     + Method: GET
     + Expected: 400 Bad Request with "User another@example.com is NOT an admin."

You have successfully explored various use cases of Spring Expression Language (SpEL)! You've seen how SpEL can be used for:

* Injecting dynamic values into beans using @Value.
* Performing arithmetic, relational, and logical operations.
* Handling null values safely with Elvis and safe navigation operators.
* Filtering and transforming collections with selection and projection.
* Integrating directly into Spring Framework features like @RequestMapping annotations.

**Activity 6.1: Spring MVC “Building a Simple Web App with JSP”**

This activity will guide you through building a basic web application using Spring MVC, focusing on its core components, configuration, request handling, data flow, using JSP (JavaServer Pages) for the view layer, and adding custom error handling and a user registration form.

### STEP 1: Project Setup (Spring Boot)

We'll use Spring Initializr to set up a new Spring Boot project, which simplifies Spring MVC configuration significantly.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.mvc
   * **Artifact:** mvc-jsp-demo
   * **Name:** mvc-jsp-demo
   * **Description:** Spring MVC with JSP Demo
   * **Package Name:** com.example.mvc
   * **Packaging:** War (Important for JSP, as it requires a WAR deployment for traditional servlet containers)
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** This is the core dependency for Spring MVC.
   * **Lombok:** (Optional but recommended) Reduces boilerplate code.
   * **Validation:** This dependency provides the necessary APIs for Jakarta Bean Validation (e.g., @NotBlank, @Email, @Size).
   * **Note:** You might not find "Tomcat Embed Jasper" or "Jakarta Servlet JSP JSTL" directly in the search results on start.spring.io. If so, proceed to generate the project and add them manually in the next step.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).
6. **Manually Add JSP-related Dependencies and Build Configuration to pom.xml:**
   * Open the pom.xml file in your project root.
   * Locate the <dependencies> section.
   * Add the following dependencies. These are crucial for JSP support when packaging as a WAR. Ensure the scope is provided for tomcat-embed-jasper if you plan to deploy to an external Tomcat, otherwise, it can be omitted for embedded Tomcat. For simplicity in Spring Boot's embedded Tomcat, provided is common for WARs.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
  
 <!-- NEW: Dependency for Jakarta Bean Validation API and Hibernate Validator implementation -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
 </dependency>  
  
 <!-- Dependency for JSP compilation (Jasper) -->  
 <dependency>  
 <groupId>org.apache.tomcat.embed</groupId>  
 <artifactId>tomcat-embed-jasper</artifactId>  
 <scope>provided</scope> <!-- Important for WAR packaging -->  
 </dependency>  
  
 <!-- Dependency for JSTL (JSP Standard Tag Library) -->  
 <dependency>  
 <groupId>jakarta.servlet.jsp.jstl</groupId>  
 <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>  
 <version>3.0.0</version> <!-- Use a compatible version, e.g., 3.0.0 for Jakarta EE 9/10 -->  
 </dependency>  
 <dependency>  
 <groupId>org.glassfish.web</groupId>  
 <artifactId>jakarta.servlet.jsp.jstl</artifactId>  
 <version>3.0.1</version> <!-- Use a compatible implementation version -->  
 </dependency>  
  
 <!-- Test Dependency -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>

* + After adding these sections, refresh your Maven project in your IDE (e.g., in IntelliJ, right-click on pom.xml and select "Maven" -> "Sync project"). This will download the new dependencies and apply the build configuration.

### STEP 2: Configure Spring MVC for JSP and Custom Error Page

For Spring Boot to correctly locate and render JSP files, and to handle custom error pages, we need to add configuration in application.properties.

1. **Open application.properties:**
   * Navigate to src/main/resources/ and open application.properties.
2. **Add JSP View Resolver and Custom Error Path Properties:**
   * Add the following lines. These tell Spring where your JSP files are located and what file extension they have, and also configure a custom error path.

# application.properties  
spring.mvc.view.prefix=/WEB-INF/jsp/  
spring.mvc.view.suffix=.jsp  
  
# Custom error page configuration  
server.error.path=/error-custom

1. **Create JSP Directory Structure:**
   * In src/main/webapp/ (you might need to create webapp if it doesn't exist, as it's typical for WAR projects), create the following directory structure:
     + src/main/webapp/WEB-INF/jsp/

### STEP 3: Create Model Classes (for Message and User Form)

We'll create two simple Java classes: one for general messages and another to represent user data collected from a form.

* Create a new package com.example.mvc.model in src/main/java/.
* Create a Java class named Message.java inside this package:  
  // src/main/java/com/example/mvc/model/Message.java  
  package com.example.mvc.model;  
    
  import lombok.AllArgsConstructor;  
  import lombok.Data;  
  import lombok.NoArgsConstructor;  
    
  /\*\*  
   \* Simple POJO (Plain Old Java Object) to act as our Model data.  
   \* Lombok annotations simplify getter/setter and constructor creation.  
   \*/  
  @Data // Generates getters, setters, toString, equals, hashCode  
  @NoArgsConstructor // Generates a no-argument constructor  
  @AllArgsConstructor // Generates a constructor with all fields  
  public class Message {  
   private String title;  
   private String content;  
  }
* Create a Java class named UserForm.java inside the same com.example.mvc.model package:  
  // src/main/java/com/example/mvc/model/UserForm.java  
  package com.example.mvc.model;  
    
  import lombok.Data;  
  import lombok.NoArgsConstructor;  
  import lombok.AllArgsConstructor;  
  import jakarta.validation.constraints.Email; // For email validation  
  import jakarta.validation.constraints.NotBlank; // For non-blank fields  
  import jakarta.validation.constraints.Size; // For size constraints  
    
  /\*\*  
   \* POJO to bind data from the user registration form.  
   \* Includes validation annotations (requires 'spring-boot-starter-validation' if you want to use @Valid in controller,  
   \* but for this simple demo, we'll just show the annotations).  
   \*/  
  @Data  
  @NoArgsConstructor  
  @AllArgsConstructor  
  public class UserForm {  
   @NotBlank(message = "Name is required")  
   @Size(min = 2, max = 50, message = "Name must be between 2 and 50 characters")  
   private String name;  
    
   @NotBlank(message = "Email is required")  
   @Email(message = "Email should be valid")  
   private String email;  
    
   @NotBlank(message = "Password is required")  
   @Size(min = 6, message = "Password must be at least 6 characters long")  
   private String password;  
  }  
    
  *Note: For @Valid to work and trigger these validation messages, you would typically need to add spring-boot-starter-validation dependency to your pom.xml. For this simple demo, we are just showing the annotations.*

### STEP 4: Create Spring MVC Controllers

We will have two controllers: one for home and about pages, and another for user-related actions (registration form and display). We'll also add a global exception handler.

* Create a new package com.example.mvc.controller in src/main/java/.
* Create a Java class named HomeController.java inside this package:  
  // src/main/java/com/example/mvc/controller/HomeController.java  
  package com.example.mvc.controller;  
    
  import com.example.mvc.model.Message; // Import our Message model  
  import org.springframework.stereotype.Controller; // Use @Controller for MVC  
  import org.springframework.ui.Model; // Used to pass data to the view  
  import org.springframework.web.bind.annotation.GetMapping; // For handling GET requests  
  import org.springframework.web.bind.annotation.RequestParam; // For request parameters  
  import org.springframework.web.servlet.ModelAndView; // Alternative for returning model and view  
    
  /\*\*  
   \* Spring MVC Controller to handle web requests.  
   \* @Controller: Marks this class as a Spring MVC controller.  
   \*/  
  @Controller  
  public class HomeController {  
    
   /\*\*  
   \* Handles GET requests to the root URL ("/").  
   \* Demonstrates Data Flow and Model Management using the 'Model' interface.  
   \*  
   \* @param name Optional request parameter.  
   \* @param model The Spring Model interface to add attributes for the view.  
   \* @return The logical view name "welcome". Spring's ViewResolver will resolve this to /WEB-INF/jsp/welcome.jsp.  
   \*/  
   @GetMapping("/")  
   public String welcome(@RequestParam(name = "name", required = false, defaultValue = "Guest") String name, Model model) {  
   // Add data to the model. This data will be available in the JSP.  
   model.addAttribute("greeting", "Hello, " + name + "!");  
   model.addAttribute("appTitle", "My Spring MVC App");  
    
   // Create a Message object and add it to the model  
   Message welcomeMessage = new Message("Welcome!", "This is a message from the Spring MVC Controller.");  
   model.addAttribute("messageObject", welcomeMessage);  
    
   // Return the logical view name  
   return "welcome"; // This will resolve to /WEB-INF/jsp/welcome.jsp  
   }  
    
   /\*\*  
   \* Handles GET requests to "/about".  
   \* Demonstrates Data Flow and Model Management using 'ModelAndView'.  
   \*  
   \* @return A ModelAndView object containing model data and the view name.  
   \*/  
   @GetMapping("/about")  
   public ModelAndView about() {  
   ModelAndView mav = new ModelAndView("about"); // Sets the logical view name to "about"  
   mav.addObject("pageTitle", "About Us"); // Add data to the model  
   mav.addObject("description", "This is a simple Spring MVC application demonstrating JSP views.");  
   return mav;  
   }  
    
   /\*\*  
   \* Handles GET requests to "/error-example".  
   \* This method will intentionally throw an exception to demonstrate basic exception handling.  
   \* (More advanced exception handling would involve @ExceptionHandler or ControllerAdvice).  
   \* @return  
   \*/  
   @GetMapping("/error-example")  
   public String errorExample() {  
   System.out.println("Triggering a simulated error...");  
   throw new RuntimeException("A simulated error occurred in the controller!");  
   }  
  }
* Create a Java class named UserController.java inside the same com.example.mvc.controller package:  
  // src/main/java/com/example/mvc/controller/UserController.java  
  package com.example.mvc.controller;  
    
  import com.example.mvc.model.UserForm; // Import our UserForm model  
  import org.springframework.stereotype.Controller;  
  import org.springframework.ui.Model;  
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.ModelAttribute; // For binding form data  
  import org.springframework.web.bind.annotation.PostMapping; // For handling POST requests  
  import org.springframework.web.bind.annotation.RequestMapping;  
  import org.springframework.web.servlet.mvc.support.RedirectAttributes; // For flash attributes  
    
  /\*\*  
   \* Controller for user-related operations, demonstrating form handling.  
   \*/  
  @Controller  
  @RequestMapping("/users") // Base path for user-related endpoints  
  public class UserController {  
    
   /\*\*  
   \* Displays the user registration form.  
   \* GET /users/register  
   \* @param model The model to add the UserForm object to for form binding.  
   \* @return The logical view name "register-form".  
   \*/  
   @GetMapping("/register")  
   public String showRegistrationForm(Model model) {  
   model.addAttribute("userForm", new UserForm()); // Add an empty UserForm object to bind to the form  
   model.addAttribute("pageTitle", "User Registration");  
   return "register-form"; // This will resolve to /WEB-INF/jsp/register-form.jsp  
   }  
    
   /\*\*  
   \* Handles the submission of the user registration form.  
   \* POST /users/register  
   \* @param userForm The UserForm object bound from the request parameters.  
   \* @param redirectAttributes Used to add flash attributes for redirection.  
   \* @return A redirect view to display user details.  
   \*/  
   @PostMapping("/register")  
   public String processRegistration(@ModelAttribute("userForm") UserForm userForm,  
   RedirectAttributes redirectAttributes) {  
   // In a real application, you would save userForm data to a database.  
   // For this demo, we'll just pass it to the next page using flash attributes.  
   System.out.println("User Registered: " + userForm.getName() + ", " + userForm.getEmail());  
    
   redirectAttributes.addFlashAttribute("registeredUser", userForm);  
   redirectAttributes.addFlashAttribute("message", "Registration successful!");  
    
   return "redirect:/users/user-details"; // Redirect to the user details page  
   }  
    
   /\*\*  
   \* Displays the details of the registered user.  
   \* GET /users/user-details  
   \* @param model The model to retrieve flash attributes from.  
   \* @return The logical view name "user-details".  
   \*/  
   @GetMapping("/user-details")  
   public String showUserDetails(@ModelAttribute("registeredUser") UserForm registeredUser, Model model) {  
   // 'registeredUser' is automatically populated from flash attributes if redirected from /register  
   // If accessed directly, registeredUser might be null or empty, so handle accordingly.  
   if (registeredUser.getName() == null) {  
   // If accessed directly without redirection, redirect back to form or show error  
   return "redirect:/users/register";  
   }  
   model.addAttribute("pageTitle", "User Details");  
   return "user-details"; // This will resolve to /WEB-INF/jsp/user-details.jsp  
   }  
  }
* Create a new package com.example.mvc.exception in src/main/java/.
* Create a Java class named GlobalExceptionHandler.java inside this package:  
  // src/main/java/com/example/mvc/exception/GlobalExceptionHandler.java  
  package com.example.mvc.exception;  
    
  import org.springframework.web.bind.annotation.ControllerAdvice;  
  import org.springframework.web.bind.annotation.ExceptionHandler;  
  import org.springframework.web.servlet.ModelAndView;  
  import jakarta.servlet.http.HttpServletRequest;  
    
  /\*\*  
   \* Global exception handler for the application.  
   \* @ControllerAdvice: Enables this class to provide centralized exception handling  
   \* across multiple controllers.  
   \*/  
  @ControllerAdvice  
  public class GlobalExceptionHandler {  
    
   /\*\*  
   \* Handles all RuntimeExceptions thrown by any controller method.  
   \* @param request The current HTTP request.  
   \* @param ex The exception that was thrown.  
   \* @return A ModelAndView object pointing to our custom error page.  
   \*/  
   @ExceptionHandler(RuntimeException.class)  
   public ModelAndView handleRuntimeException(HttpServletRequest request, RuntimeException ex) {  
   ModelAndView mav = new ModelAndView("custom-error"); // Logical view name for our error page  
   mav.addObject("exception", ex); // Pass the exception object to the view  
   mav.addObject("url", request.getRequestURL()); // Pass the URL that caused the error  
   mav.addObject("timestamp", new java.util.Date()); // Add a timestamp  
   mav.addObject("status", 500); // Set a generic HTTP status  
   mav.addObject("errorMessage", "An unexpected error occurred: " + ex.getMessage());  
   System.err.println("Global Exception Handler caught: " + ex.getMessage() + " at URL: " + request.getRequestURL());  
   return mav;  
   }  
    
   // You can add more @ExceptionHandler methods for specific exception types  
   // @ExceptionHandler(NoHandlerFoundException.class)  
   // public ModelAndView handleNoHandlerFoundException(...) { ... }  
  }

### STEP 5: Create JSP View Files (View Technologies)

Now, let's create the JSP files that will render the data prepared by our controller, including new JSPs for the user form and error page.

1. **Create welcome.jsp:**
   * In the src/main/webapp/WEB-INF/jsp/ directory, create a file named welcome.jsp.

<%-- src/main/webapp/WEB-INF/jsp/welcome.jsp --%>  
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>${appTitle}</title> <%-- Accessing model attribute directly --%>  
 <style>  
 body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
 .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; }  
 h1 { color: #0056b3; }  
 p { line-height: 1.6; }  
 .message-box { border: 1px solid #ddd; padding: 15px; margin-top: 20px; background-color: #e9f7ef; border-radius: 5px; }  
 .message-title { font-weight: bold; color: #28a745; }  
 .message-content { font-style: italic; }  
 a { color: #007bff; text-decoration: none; }  
 a:hover { text-decoration: underline; }  
 </style>  
</head>  
<body>  
 <div class="container">  
 <h1>${greeting}</h1> <%-- Accessing model attribute --%>  
 <p>Welcome to your first Spring MVC application with JSP!</p>  
  
 <div class="message-box">  
 <p class="message-title">Message Title: <c:out value="${messageObject.title}"/></p>  
 <p class="message-content">Message Content: <c:out value="${messageObject.content}"/></p>  
 </div>  
  
 <p><a href="/about">Learn more about this app</a></p>  
 <p><a href="/users/register">Register as a new user</a></p>  
 <p><a href="/error-example">Trigger a simulated error</a></p>  
 </div>  
</body>  
</html>

1. **Create about.jsp:**
   * In the src/main/webapp/WEB-INF/jsp/ directory, create a file named about.jsp.

<%-- src/main/webapp/WEB-INF/jsp/about.jsp --%>  
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  
<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>${pageTitle}</title> <%-- Accessing model attribute from ModelAndView --%>  
 <style>  
 body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
 .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; }  
 h1 { color: #0056b3; }  
 p { line-height: 1.6; }  
 a { color: #007bff; text-decoration: none; }  
 a:hover { text-decoration: underline; }  
 </style>  
</head>  
<body>  
 <div class="container">  
 <h1>${pageTitle}</h1>  
 <p>${description}</p>  
 <p><a href="/">Go back to Home</a></p>  
 </div>  
</body>  
</html>

1. **Create register-form.jsp (for User Info Form):**
   * In the src/main/webapp/WEB-INF/jsp/ directory, create a file named register-form.jsp.

<%-- src/main/webapp/WEB-INF/jsp/register-form.jsp --%>  
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>${pageTitle}</title>  
 <style>  
 body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
 .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 500px; margin: auto; }  
 h1 { color: #0056b3; text-align: center; }  
 .form-group { margin-bottom: 15px; }  
 .form-group label { display: block; margin-bottom: 5px; font-weight: bold; }  
 .form-group input[type="text"],  
 .form-group input[type="email"],  
 .form-group input[type="password"] {  
 width: calc(100% - 22px); /\* Account for padding and border \*/  
 padding: 10px;  
 border: 1px solid #ddd;  
 border-radius: 4px;  
 box-sizing: border-box; /\* Include padding and border in the element's total width and height \*/  
 }  
 .form-group .error { color: red; font-size: 0.9em; margin-top: 5px; }  
 .button-container { text-align: center; margin-top: 20px; }  
 .button-container button {  
 background-color: #28a745;  
 color: white;  
 padding: 10px 20px;  
 border: none;  
 border-radius: 5px;  
 cursor: pointer;  
 font-size: 1em;  
 transition: background-color 0.3s ease;  
 }  
 .button-container button:hover { background-color: #218838; }  
 a { color: #007bff; text-decoration: none; }  
 a:hover { text-decoration: underline; }  
 </style>  
</head>  
<body>  
 <div class="container">  
 <h1>${pageTitle}</h1>  
 <%-- Spring's form tag library is used here. modelAttribute links to the UserForm object in the model. --%>  
 <form:form action="/users/register" method="post" modelAttribute="userForm">  
 <div class="form-group">  
 <label for="name">Name:</label>  
 <form:input path="name" type="text" id="name"/>  
 <%-- <form:errors path="name" cssClass="error"/> --%>  
 </div>  
 <div class="form-group">  
 <label for="email">Email:</label>  
 <form:input path="email" type="email" id="email"/>  
 <%-- <form:errors path="email" cssClass="error"/> --%>  
 </div>  
 <div class="form-group">  
 <label for="password">Password:</label>  
 <form:input path="password" type="password" id="password"/>  
 <%-- <form:errors path="password" cssClass="error"/> --%>  
 </div>  
 <div class="button-container">  
 <button type="submit">Register</button>  
 </div>  
 </form:form>  
 <p style="text-align: center; margin-top: 20px;"><a href="/">Go back to Home</a></p>  
 </div>  
</body>  
</html>

1. **Create user-details.jsp (for displaying User Info):**
   * In the src/main/webapp/WEB-INF/jsp/ directory, create a file named user-details.jsp.

<%-- src/main/webapp/WEB-INF/jsp/user-details.jsp --%>  
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>${pageTitle}</title>  
 <style>  
 body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
 .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 500px; margin: auto; }  
 h1 { color: #0056b3; text-align: center; }  
 .detail-item { margin-bottom: 10px; }  
 .detail-item strong { display: inline-block; width: 80px; }  
 .success-message { color: green; font-weight: bold; text-align: center; margin-bottom: 20px; }  
 a { color: #007bff; text-decoration: none; }  
 a:hover { text-decoration: underline; }  
 </style>  
</head>  
<body>  
 <div class="container">  
 <h1>${pageTitle}</h1>  
  
 <c:if test="${not empty message}">  
 <p class="success-message">${message}</p>  
 </c:if>  
  
 <c:choose>  
 <c:when test="${not empty registeredUser}">  
 <div class="detail-item"><strong>Name:</strong> <c:out value="${registeredUser.name}"/></div>  
 <div class="detail-item"><strong>Email:</strong> <c:out value="${registeredUser.email}"/></div>  
 <%-- Password is not displayed for security reasons --%>  
 </c:when>  
 <c:otherwise>  
 <p style="text-align: center;">No user details found. Please <a href="/users/register">register</a> first.</p>  
 </c:otherwise>  
 </c:choose>  
  
 <p style="text-align: center; margin-top: 20px;"><a href="/">Go back to Home</a></p>  
 </div>  
</body>  
</html>

1. **Create custom-error.jsp (for Custom Error Page):**
   * In the src/main/webapp/WEB-INF/jsp/ directory, create a file named custom-error.jsp.

<%-- src/main/webapp/WEB-INF/jsp/custom-error.jsp --%>  
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" isErrorPage="true"%>  
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>Error Occurred</title>  
 <style>  
 body { font-family: Arial, sans-serif; margin: 20px; background-color: #fcebeb; color: #333; }  
 .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; border: 1px solid #e0b4b4; }  
 h1 { color: #d9534f; text-align: center; }  
 h2 { color: #f0ad4e; font-size: 1.2em; }  
 p { line-height: 1.6; }  
 .error-details { background-color: #fdf7f7; border: 1px solid #f5c6cb; padding: 15px; border-radius: 5px; margin-top: 20px; word-wrap: break-word; }  
 a { color: #007bff; text-decoration: none; }  
 a:hover { text-decoration: underline; }  
 </style>  
</head>  
<body>  
 <div class="container">  
 <h1>Oops! Something went wrong.</h1>  
 <p style="text-align: center;">We're sorry, but an unexpected error occurred.</p>  
  
 <c:if test="${not empty errorMessage}">  
 <p class="error-details"><strong>Error Message:</strong> <c:out value="${errorMessage}"/></p>  
 </c:if>  
 <c:if test="${not empty status}">  
 <p class="error-details"><strong>Status:</strong> <c:out value="${status}"/></p>  
 </c:if>  
 <c:if test="${not empty url}">  
 <p class="error-details"><strong>Request URL:</strong> <c:out value="${url}"/></p>  
 </c:if>  
 <c:if test="${not empty timestamp}">  
 <p class="error-details"><strong>Timestamp:</strong> <c:out value="${timestamp}"/></p>  
 </c:if>  
 <c:if test="${not empty exception}">  
 <p class="error-details"><strong>Exception Type:</strong> <c:out value="${exception['class'].name}"/></p>  
 <p class="error-details"><strong>Details:</strong> <c:out value="${exception.message}"/></p>  
 </c:if>  
  
 <p style="text-align: center; margin-top: 20px;"><a href="/">Go back to Home</a></p>  
 </div>  
</body>  
</html>

### STEP 6: Main Spring Boot Application Class

This is the standard Spring Boot application entry point. No changes are typically needed here for basic MVC setup.

* Open your main application class (e.g., MvcJspDemoApplication.java in com.example.mvc).  
  // src/main/java/com/example/mvc/app/MvcJspDemoApplication.java  
  package com.example.mvc;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
  import org.springframework.boot.builder.SpringApplicationBuilder;  
  import org.springframework.boot.web.servlet.support.SpringBootServletInitializer; // Needed for WAR deployment  
    
  /\*\*  
   \* Main Spring Boot application class.  
   \* Extends SpringBootServletInitializer for WAR deployment to a traditional servlet container.  
   \*/  
  @SpringBootApplication  
  public class MvcJspDemoApplication extends SpringBootServletInitializer {  
    
   public static void main(String[] args) {  
   SpringApplication.run(MvcJspDemoApplication.class, args);  
   }  
    
   // This method is crucial for configuring the application when deployed as a WAR file  
   // to an external servlet container (like Tomcat).  
   @Override  
   protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
   return application.sources(MvcJspDemoApplication.class);  
   }  
  }

### STEP 7: Run the Application and Test

1. **Run from your IDE (Embedded Tomcat):**
   * Open your main application class (MvcJspDemoApplication.java).
   * Run it as a Java Application from your IDE. Spring Boot will start an embedded Tomcat server.
2. **Access the Application:**
   * Open your web browser and navigate to:
     + http://localhost:8080/ (for the welcome page)
     + http://localhost:8080/?name=Alice (to see the name parameter in action)
     + http://localhost:8080/about (for the about page)
     + http://localhost:8080/users/register (to access the user registration form)
     + Fill out the form and submit it. You should be redirected to /users/user-details showing the submitted info.
     + http://localhost:8080/error-example (to trigger a simulated error and see your custom error page)
   * Observe the content rendered by the JSP files and how the model data is displayed.

You have successfully built a simple web application using Spring MVC with JSP views! You've learned about:

* The core components of Spring MVC (DispatcherServlet, Controllers, Views).
* Configuring Spring MVC for JSP.
* Handling requests with @GetMapping and @RequestParam.
* Managing data flow using Model and ModelAndView.
* Using JSP as a view technology to render dynamic content.
* Implementing a custom error page for better user experience.
* Creating a form to collect user information and display it on a subsequent page.

This provides a solid foundation for further exploration of Spring MVC's capabilities.

# Activity 7.1: Spring Security “Basic Authentication & Authorization”

This activity will guide you through implementing basic security in a Spring Boot web application using Spring Security. We'll cover authentication, authorization, web security configuration, and method security.

### STEP 1: Project Setup (Spring Boot)

We'll use Spring Initializr to set up a new Spring Boot project with the necessary security dependencies.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.security
   * **Artifact:** security-demo
   * **Name:** security-demo
   * **Description:** Spring Security Demo
   * **Package Name:** com.example.security
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** For building a simple web application.
   * **Spring Security:** The core Spring Security dependency.
   * **Thymeleaf:** (Optional, but good for simple web views without JSPs) A modern server-side Java template engine. We'll use it for simple HTML pages.
   * **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

### STEP 2: Configure Web Security (Authentication & Authorization)

This is where we define security rules, user details, and password encoding.

* Create a new package com.example.security.config in src/main/java/.
* Create a Java class named SecurityConfig.java inside this package:  
  // src/main/java/com/example/security/config/SecurityConfig.java  
  package com.example.security.config;  
    
  import org.springframework.context.annotation.Bean;  
  import org.springframework.context.annotation.Configuration;  
  import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity; // For method security  
  import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
  import org.springframework.security.core.userdetails.User;  
  import org.springframework.security.core.userdetails.UserDetails;  
  import org.springframework.security.core.userdetails.UserDetailsService;  
  import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder; // For password encoding  
  import org.springframework.security.crypto.password.PasswordEncoder;  
  import org.springframework.security.provisioning.InMemoryUserDetailsManager; // For in-memory users  
  import org.springframework.security.web.SecurityFilterChain;  
    
  /\*\*  
   \* Spring Security configuration class.  
   \* @EnableWebSecurity: Enables Spring Security's web security features.  
   \* @EnableMethodSecurity: Enables method-level security annotations like @PreAuthorize.  
   \*/  
  @Configuration  
  @EnableWebSecurity  
  @EnableMethodSecurity // Enables @PreAuthorize, @PostAuthorize, @Secured, @RolesAllowed  
  public class SecurityConfig {  
    
   /\*\*  
   \* Configures the SecurityFilterChain, defining authorization rules for HTTP requests.  
   \* This is the core of Web Security Configuration.  
   \*  
   \* @param http HttpSecurity object to configure security.  
   \* @return A SecurityFilterChain instance.  
   \* @throws Exception if configuration fails.  
   \*/  
   @Bean  
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
   http  
   .authorizeHttpRequests(authorize -> authorize  
   // Publicly accessible paths  
   .requestMatchers("/", "/home", "/public").permitAll()  
   // Paths requiring specific roles (Authorization)  
   .requestMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN") // /user paths require USER or ADMIN role  
   .requestMatchers("/admin/\*\*").hasRole("ADMIN") // /admin paths require ADMIN role  
   // All other requests require authentication  
   .anyRequest().authenticated()  
   )  
   .formLogin(form -> form  
   .loginPage("/login") // Custom login page URL  
   .permitAll() // Allow everyone to access the login page  
   )  
   .logout(logout -> logout  
   .permitAll() // Allow everyone to logout  
   );  
   // CSRF Protection is enabled by default in Spring Security.  
   // For simple forms, Spring Security automatically adds a CSRF token.  
   // For REST APIs, you might disable it or handle it differently.  
   // .csrf(csrf -> csrf.disable()); // Uncomment to disable CSRF (e.g., for stateless REST APIs)  
    
   return http.build();  
   }  
    
   /\*\*  
   \* Configures an in-memory UserDetailsService for Authentication.  
   \* In a real application, this would typically retrieve users from a database.  
   \*  
   \* @return A UserDetailsService with predefined users.  
   \*/  
   @Bean  
   public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder) {  
   // Define a 'user' with password 'password' and role 'USER'  
   UserDetails user = User.builder()  
   .username("user")  
   .password(passwordEncoder.encode("password")) // Encode password  
   .roles("USER")  
   .build();  
    
   // Define an 'admin' with password 'admin' and role 'ADMIN'  
   UserDetails admin = User.builder()  
   .username("admin")  
   .password(passwordEncoder.encode("admin")) // Encode password  
   .roles("ADMIN", "USER") // Admin also has USER role  
   .build();  
    
   // Return an InMemoryUserDetailsManager with these users  
   return new InMemoryUserDetailsManager(user, admin);  
   }  
    
   /\*\*  
   \* Configures a PasswordEncoder.  
   \* BCryptPasswordEncoder is recommended for strong password hashing (Password Encoding).  
   \*  
   \* @return A BCryptPasswordEncoder instance.  
   \*/  
   @Bean  
   public PasswordEncoder passwordEncoder() {  
   return new BCryptPasswordEncoder();  
   }  
  }

### STEP 3: Create Controllers and Views

We'll create simple controllers and Thymeleaf HTML files to demonstrate different access levels.

* Create a new package com.example.security.controller in src/main/java/.
* Create a Java class named WebController.java inside this package:  
  // src/main/java/com/example/security/controller/WebController.java  
  package com.example.security.controller;  
    
  import org.springframework.security.access.prepost.PreAuthorize; // For Method Security  
  import org.springframework.stereotype.Controller;  
  import org.springframework.ui.Model;  
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.RequestMapping;  
    
  /\*\*  
   \* Web Controller for serving HTML pages and demonstrating access control.  
   \*/  
  @Controller  
  public class WebController {  
    
   @GetMapping("/home")  
   public String home() {  
   return "home"; // Resolves to src/main/resources/templates/home.html  
   }

@GetMapping("/")

public String defaultPage() {

return "home"; // Resolves to src/main/resources/templates/home.html

}

@GetMapping("/public")  
 public String publicPage() {  
 return "public-page"; // Resolves to src/main/resources/templates/public-page.html  
 }  
  
 @GetMapping("/user/dashboard")  
 public String userDashboard(Model model) {  
 model.addAttribute("message", "Welcome, User! This is your dashboard.");  
 return "user-dashboard"; // Resolves to src/main/resources/templates/user-dashboard.html  
 }  
  
 @GetMapping("/admin/panel")  
 public String adminPanel(Model model) {  
 model.addAttribute("message", "Welcome, Admin! This is your control panel.");  
 return "admin-panel"; // Resolves to src/main/resources/templates/admin-panel.html  
 }  
  
 @GetMapping("/login")  
 public String login() {  
 return "login"; // Resolves to src/main/resources/templates/login.html  
 }  
  
 /\*\*  
 \* Demonstrates Method Security using @PreAuthorize.  
 \* Only users with the 'ADMIN' role can access this method.  
 \*/  
 @PreAuthorize("hasRole('ADMIN')") // Method Security: Requires ADMIN role  
 @GetMapping("/admin/secure-action")  
 public String secureAdminAction(Model model) {  
 model.addAttribute("message", "You accessed a highly secure admin action!");  
 return "admin-panel"; // Redirect back to admin panel with a message  
 }  
  
 /\*\*  
 \* Demonstrates Method Security using @PreAuthorize.  
 \* Only users with the 'USER' role (or ADMIN, since ADMIN has USER role) can access this method.  
 \*/  
 @PreAuthorize("hasRole('USER')") // Method Security: Requires USER role  
 @GetMapping("/user/secure-action")  
 public String secureUserAction(Model model) {  
 model.addAttribute("message", "You accessed a secure user action!");  
 return "user-dashboard"; // Redirect back to user dashboard with a message  
 }  
}

* Create a new package com.example.security.rest in src/main/java/.
* Create a Java class named SecureRestController.java inside this package:  
  // src/main/java/com/example/security/rest/SecureRestController.java  
  package com.example.security.rest;  
    
  import org.springframework.security.access.prepost.PreAuthorize;  
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.RequestMapping;  
  import org.springframework.web.bind.annotation.RestController;  
    
  /\*\*  
   \* REST Controller to demonstrate Security in REST APIs.  
   \* Spring Security applies to REST endpoints just like regular web endpoints.  
   \*/  
  @RestController  
  @RequestMapping("/api")  
  public class SecureRestController {  
    
   @GetMapping("/public-data")  
   public String getPublicData() {  
   return "This data is accessible to anyone.";  
   }  
    
   @GetMapping("/user-data")  
   @PreAuthorize("hasRole('USER')") // Requires USER role via Method Security  
   public String getUserData() {  
   return "This data is accessible to authenticated users with USER role.";  
   }  
    
   @GetMapping("/admin-data")  
   @PreAuthorize("hasRole('ADMIN')") // Requires ADMIN role via Method Security  
   public String getAdminData() {  
   return "This data is accessible only to users with ADMIN role.";  
   }  
  }
* Create Thymeleaf HTML files in src/main/resources/templates/:
  1. **home.html**:  
     <!DOCTYPE html>  
     <html lang="en" xmlns:th="http://www.thymeleaf.org">  
     <head>  
      <meta charset="UTF-8">  
      <title>Home Page</title>  
      <style>  
      body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
      .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; text-align: center; }  
      h1 { color: #0056b3; }  
      a { color: #007bff; text-decoration: none; margin: 0 10px; }  
      a:hover { text-decoration: underline; }  
      .nav-links { margin-top: 20px; }  
      </style>  
     </head>  
     <body>  
      <div class="container">  
      <h1>Welcome to the Spring Security Demo!</h1>  
      <p>Explore different pages based on your roles.</p>  
      <div class="nav-links">  
      <a th:href="@{/public}">Public Page</a>  
      <a th:href="@{/user/dashboard}">User Dashboard</a>  
      <a th:href="@{/admin/panel}">Admin Panel</a>  
      <a th:href="@{/login}">Login</a>  
      <form th:action="@{/logout}" method="post" style="display:inline;">  
      <button type="submit" style="background: none; border: none; color: #dc3545; cursor: pointer; font-size: 1em; text-decoration: underline;">Logout</button>  
      </form>  
      </div>  
      </div>  
     </body>  
     </html>
  2. **public-page.html**:  
     <!DOCTYPE html>  
     <html lang="en" xmlns:th="http://www.thymeleaf.org">  
     <head>  
      <meta charset="UTF-8">  
      <title>Public Page</title>  
      <style>  
      body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
      .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; text-align: center; }  
      h1 { color: #28a745; }  
      a { color: #007bff; text-decoration: none; margin: 0 10px; }  
      a:hover { text-decoration: underline; }  
      </style>  
     </head>  
     <body>  
      <div class="container">  
      <h1>This is a Public Page</h1>  
      <p>Anyone can access this page without logging in.</p>  
      <a th:href="@{/home}">Go to Home</a>  
      </div>  
     </body>  
     </html>
  3. **user-dashboard.html**:  
     <!DOCTYPE html>  
     <html lang="en" xmlns:th="http://www.thymeleaf.org">  
     <head>  
      <meta charset="UTF-8">  
      <title>User Dashboard</title>  
      <style>  
      body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
      .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; text-align: center; }  
      h1 { color: #ffc107; }  
      p { font-size: 1.1em; color: #555; }  
      a { color: #007bff; text-decoration: none; margin: 0 10px; }  
      a:hover { text-decoration: underline; }  
      .message { margin-top: 20px; padding: 10px; background-color: #e2f0d9; border-left: 5px solid #28a745; text-align: left; }  
      </style>  
     </head>  
     <body>  
      <div class="container">  
      <h1>User Dashboard</h1>  
      <p th:text="${message}"></p>  
      <p>You have successfully logged in as a USER (or ADMIN).</p>  
      <div th:if="${#authorization.expression('hasRole(''ADMIN'')')}">  
      <p>You also have ADMIN privileges.</p>  
      </div>  
      <a th:href="@{/user/secure-action}">Access Secure User Action (Method Security)</a>  
      <a th:href="@{/home}">Go to Home</a>  
      <form th:action="@{/logout}" method="post" style="display:inline;">  
      <button type="submit" style="background: none; border: none; color: #dc3545; cursor: pointer; font-size: 1em; text-decoration: underline;">Logout</button>  
      </form>  
      </div>  
     </body>  
     </html>
  4. **admin-panel.html**:  
     <!DOCTYPE html>  
     <html lang="en" xmlns:th="http://www.thymeleaf.org">  
     <head>  
      <meta charset="UTF-8">  
      <title>Admin Panel</title>  
      <style>  
      body { font-family: Arial, sans-serif; margin: 20px; background-color: #f4f4f4; color: #333; }  
      .container { background-color: #fff; padding: 30px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); max-width: 600px; margin: auto; text-align: center; }  
      h1 { color: #dc3545; }  
      p { font-size: 1.1em; color: #555; }  
      a { color: #007bff; text-decoration: none; margin: 0 10px; }  
      a:hover { text-decoration: underline; }  
      .message { margin-top: 20px; padding: 10px; background-color: #f8d7da; border-left: 5px solid #dc3545; text-align: left; }  
      </style>  
     </head>  
     <body>  
      <div class="container">  
      <h1>Admin Panel</h1>  
      <p th:text="${message}"></p>  
      <p>You have successfully logged in as an ADMIN.</p>  
      <a th:href="@{/admin/secure-action}">Access Highly Secure Admin Action (Method Security)</a>  
      <a th:href="@{/home}">Go to Home</a>  
      <form th:action="@{/logout}" method="post" style="display:inline;">  
      <button type="submit" style="background: none; border: none; color: #dc3545; cursor: pointer; font-size: 1em; text-decoration: underline;">Logout</button>  
      </form>  
      </div>  
     </body>  
     </html>
  5. **login.html**:  
     <!DOCTYPE html>  
     <html lang="en" xmlns:th="http://www.thymeleaf.org">  
     <head>  
      <meta charset="UTF-8">  
      <title>Login</title>  
      <style>  
      body { font-family: Arial, sans-serif; background-color: #f4f4f4; display: flex; justify-content: center; align-items: center; height: 100vh; margin: 0; }  
      .login-container { background-color: #fff; padding: 40px; border-radius: 8px; box-shadow: 0 4px 8px rgba(0,0,0,0.1); width: 350px; text-align: center; }  
      h1 { color: #0056b3; margin-bottom: 25px; }  
      .form-group { margin-bottom: 20px; text-align: left; }  
      .form-group label { display: block; margin-bottom: 8px; font-weight: bold; color: #555; }  
      .form-group input[type="text"],  
      .form-group input[type="password"] {  
      width: calc(100% - 22px);  
      padding: 12px;  
      border: 1px solid #ddd;  
      border-radius: 5px;  
      font-size: 1em;  
      box-sizing: border-box;  
      }  
      .error-message { color: #dc3545; margin-bottom: 15px; font-weight: bold; }  
      .login-button {  
      background-color: #007bff;  
      color: white;  
      padding: 12px 25px;  
      border: none;  
      border-radius: 5px;  
      cursor: pointer;  
      font-size: 1.1em;  
      transition: background-color 0.3s ease;  
      width: 100%;  
      }  
      .login-button:hover { background-color: #0056b3; }  
      .info-text { margin-top: 20px; font-size: 0.9em; color: #777; }  
      .info-text strong { color: #333; }  
      </style>  
     </head>  
     <body>  
      <div class="login-container">  
      <h1>Login</h1>  
      <div th:if="${param.error}" class="error-message">  
      Invalid username or password.  
      </div>  
      <div th:if="${param.logout}" class="error-message" style="color: green;">  
      You have been logged out.  
      </div>  
      <form th:action="@{/login}" method="post">  
      <div class="form-group">  
      <label for="username">Username:</label>  
      <input type="text" id="username" name="username" required>  
      </div>  
      <div class="form-group">  
      <label for="password">Password:</label>  
      <input type="password" id="password" name="password" required>  
      </div>  
      <button type="submit" class="login-button">Log In</button>  
      </form>  
      <div class="info-text">  
      <p>Try with:</p>  
      <p><strong>Username:</strong> user, <strong>Password:</strong> password (Role: USER)</p>  
      <p><strong>Username:</strong> admin, <strong>Password:</strong> admin (Role: ADMIN)</p>  
      </div>  
      </div>  
     </body>  
     </html>

### STEP 4: Main Spring Boot Application Class

This is the standard Spring Boot application entry point.

* Open your main application class (e.g., SecurityDemoApplication.java in com.example.security).  
  // src/main/java/com/example/security/app/SecurityDemoApplication.java  
  package com.example.security;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
    
  @SpringBootApplication  
  public class SecurityDemoApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(SecurityDemoApplication.class, args);  
   }  
  }

### STEP 5: Run the Application and Test Security

1. **Run the Spring Boot Application:**
   * Open your main application class (SecurityDemoApplication.java).
   * Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
2. **Test Access Control:**
   * **Public Page:**
     + Open your browser and go to http://localhost:8080/public. You should be able to access this page without logging in.
     + Go to http://localhost:8080/home. This is also publicly accessible.
   * **Login Page:**
     + Try to go to http://localhost:8080/user/dashboard. Spring Security will redirect you to the /login page because you are not authenticated.
     + Use the provided credentials:
       - **Username:** user, **Password:** password
       - **Username:** admin, **Password:** admin
   * **User Dashboard (Authorization):**
     + Log in as user.
     + Go to http://localhost:8080/user/dashboard. You should see the user dashboard.
     + Try to go to http://localhost:8080/admin/panel. You should be redirected to an "Access Denied" page (or login page if not authenticated) because the user role does not have access to /admin/\*\* paths.
   * **Admin Panel (Authorization):**
     + Log out (click the logout button on the home/dashboard page) and log in as admin.
     + Go to http://localhost:8080/admin/panel. You should see the admin panel.
     + You can also access http://localhost:8080/user/dashboard as admin because the admin user has both ADMIN and USER roles.
   * **Method Security (@PreAuthorize):**
     + Log in as user.
     + Go to http://localhost:8080/user/secure-action. You should see the message "You accessed a secure user action!"
     + Try to go to http://localhost:8080/admin/secure-action. You should get an "Access Denied" error because the user role cannot access this method.
     + Log out and log in as admin.
     + Go to http://localhost:8080/admin/secure-action. You should now successfully access this method.
   * **Security in REST APIs:**
     + Open a new browser tab or use a tool like Postman/curl.
     + http://localhost:8080/api/public-data: Accessible without authentication.
     + http://localhost:8080/api/user-data: Requires authentication with USER or ADMIN role. If not logged in, it will redirect to login (for browser) or return 401 Unauthorized (for API clients).
     + http://localhost:8080/api/admin-data: Requires authentication with ADMIN role.

### CSRF Protection

* Spring Security enables CSRF (Cross-Site Request Forgery) protection by default.
* For form submissions (like our login form), Spring Security automatically injects a hidden input field named \_csrf with a token. You can see this in login.html form.
* When you submit the form, this token is sent with the request, and Spring Security validates it. If the token is missing or invalid, the request is rejected.
* For stateless REST APIs, CSRF protection is often disabled (.csrf(csrf -> csrf.disable()) in SecurityConfig) because tokens are typically handled differently (e.g., JWTs) or not applicable. However, for a stateful web application, keeping CSRF enabled is a good practice.

### Password Encoding

* We used BCryptPasswordEncoder in SecurityConfig. This is a strong, one-way hashing algorithm for passwords.
* When a user registers or logs in, Spring Security uses this PasswordEncoder to hash the provided password and compare it with the stored (hashed) password. It never stores or compares plain-text passwords.

You have successfully implemented a basic Spring Security setup! You've learned about:

* The fundamental concepts of authentication and authorization.
* Configuring web security rules using HttpSecurity.
* Defining in-memory users with UserDetailsService.
* Implementing method-level security with @PreAuthorize.
* The importance of PasswordEncoder for secure password storage.
* Basic security considerations for REST APIs and CSRF protection.

This activity provides a strong foundation for securing your Spring applications.

# Activity 8.1: Spring Remoting “HTTP-based Communication with RestTemplate”

This activity will guide you through implementing HTTP-based communication using Spring's RestTemplate. You will create two separate Spring Boot applications: a **Server** that exposes a REST API, and a **Client** that consumes it using RestTemplate.

In this activity, we will focus on using RestTemplate to demonstrate client-server communication over HTTP.

### STEP 1: Project Setup (Server Application - REST API)

First, let's set up the Spring Boot application that will expose our REST endpoints.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.restcomm
   * **Artifact:** rest-server
   * **Name:** rest-server
   * **Description:** Spring REST Server Demo
   * **Package Name:** com.example.restcomm.server
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** Essential for building REST controllers.
   * **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

### STEP 2: Define Data Transfer Objects (DTOs - Shared)

We'll define simple DTOs (Data Transfer Objects) that will be exchanged between the client and server. These classes represent the structure of the data.

* Create a new package com.example.restcomm.shared in src/main/java/ of your rest-server project.
* Create a Java class named GreetingRequest.java inside this package:  
  // src/main/java/com/example/restcomm/shared/GreetingRequest.java  
  package com.example.restcomm.shared;  
    
  import lombok.AllArgsConstructor;  
  import lombok.Data;  
  import lombok.NoArgsConstructor;  
    
  /\*\*  
   \* DTO for sending a greeting request.  
   \*/  
  @Data // Lombok: Generates getters, setters, toString, equals, hashCode  
  @NoArgsConstructor // Lombok: Generates no-argument constructor  
  @AllArgsConstructor // Lombok: Generates constructor with all fields  
  public class GreetingRequest {  
   private String name;  
  }
* Create a Java class named GreetingResponse.java inside the same com.example.restcomm.shared package:  
  // src/main/java/com/example/restcomm/shared/GreetingResponse.java  
  package com.example.restcomm.shared;  
    
  import lombok.AllArgsConstructor;  
  import lombok.Data;  
  import lombok.NoArgsConstructor;  
    
  /\*\*  
   \* DTO for receiving a greeting response.  
   \*/  
  @Data  
  @NoArgsConstructor  
  @AllArgsConstructor  
  public class GreetingResponse {  
   private String message;  
   private String timestamp;  
  }

### STEP 3: Create REST Controller on Server-Side

Now, let's create the REST controller that will expose endpoints for our client to consume.

* Create a new package com.example.restcomm.controller in src/main/java/ of your rest-server project.
* Create a Java class named GreetingController.java inside this package:  
  // src/main/java/com/example/restcomm/server/controller/GreetingController.java  
  package com.example.restcomm.controller;  
    
  import com.example.restcomm.shared.GreetingRequest;  
  import com.example.restcomm.shared.GreetingResponse;  
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.PostMapping;  
  import org.springframework.web.bind.annotation.RequestBody;  
  import org.springframework.web.bind.annotation.RequestParam;  
  import org.springframework.web.bind.annotation.RestController;  
    
  import java.time.LocalDateTime;  
  import java.time.format.DateTimeFormatter;  
    
  /\*\*  
   \* REST Controller exposing greeting endpoints.  
   \*/  
  @RestController // Marks this as a REST controller  
  public class GreetingController {  
    
   private static final DateTimeFormatter FORMATTER = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  
    
   /\*\*  
   \* Handles GET requests to /api/greet.  
   \* Example: GET http://localhost:8080/api/greet?name=World  
   \* @param name The name to greet, from request parameter.  
   \* @return A GreetingResponse object.  
   \*/  
   @GetMapping("/api/greet")  
   public GreetingResponse getGreeting(@RequestParam(defaultValue = "Guest") String name) {  
   System.out.println("Server: Received GET request for greeting for: " + name);  
   String message = "Hello, " + name + " from REST server (GET)!";  
   return new GreetingResponse(message, LocalDateTime.now().format(FORMATTER));  
   }  
    
   /\*\*  
   \* Handles POST requests to /api/greet.  
   \* Expects a JSON request body with a 'name' field.  
   \* Example: POST http://localhost:8080/api/greet  
   \* Body: {"name": "Alice"}  
   \* @param request The GreetingRequest object from the JSON request body.  
   \* @return A GreetingResponse object.  
   \*/  
   @PostMapping("/api/greet")  
   public GreetingResponse postGreeting(@RequestBody GreetingRequest request) {  
   System.out.println("Server: Received POST request for greeting for: " + request.getName());  
   String message = "Hello, " + request.getName() + " from REST server (POST)!";  
   return new GreetingResponse(message, LocalDateTime.now().format(FORMATTER));  
   }  
  }

### STEP 4: Main Server Application Class

This is the standard Spring Boot application entry point for the server.

* Open your main application class (e.g., RestServerApplication.java in com.example.restcomm).  
  // src/main/java/com/example/restcomm/server/RestServerApplication.java  
  package com.example.restcomm;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
    
  @SpringBootApplication  
  public class RestServerApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(RestServerApplication.class, args);  
   }  
  }

### STEP 5: Project Setup (Client Application)

Now, let's set up the Spring Boot application that will consume the REST API.

1. **Create a New Project:** Repeat STEP 1, but with the following configurations:
   * **Group:** com.example.restcomm
   * **Artifact:** rest-client
   * **Name:** rest-client
   * **Description:** Spring REST Client Demo
   * **Package Name:** com.example.restcomm.client
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
   * **Dependencies:** **Spring Web** (this includes RestTemplate), **Lombok**.
2. **Import into IDE:** Unzip and import the rest-client project.
3. **Copy the Shared DTOs:**
   * Copy GreetingRequest.java and GreetingResponse.java from rest-server/src/main/java/com/example/restcomm/shared/ to rest-client/src/main/java/com/example/restcomm/client/shared/.
   * **Important:** The package structure (com.example.restcomm.shared) must be identical in both projects.

### STEP 6: Configure RestTemplate on Client-Side

We'll configure RestTemplate as a Spring @Bean so it can be easily injected and used.

* Create a new package com.example.restcomm.client.config in src/main/java/ of your rest-client project.
* Create a Java class named ClientConfig.java inside this package:  
  // src/main/java/com/example/restcomm/client/config/ClientConfig.java  
  package com.example.restcomm.client.config;  
    
  import org.springframework.context.annotation.Bean;  
  import org.springframework.context.annotation.Configuration;  
  import org.springframework.web.client.RestTemplate; // The key class for HTTP client  
    
  /\*\*  
   \* Configuration for creating and customizing RestTemplate.  
   \*/  
  @Configuration  
  public class ClientConfig {  
    
   /\*\*  
   \* Creates a RestTemplate bean.  
   \* @return A configured RestTemplate instance.  
   \*/  
   @Bean  
   public RestTemplate restTemplate() {  
   return new RestTemplate();  
   }  
  }

### STEP 7: Create a Client Component to Consume REST API

Let's create a simple component that injects RestTemplate and makes calls to the server's REST API.

* Create a new package com.example.restcomm.client.component in src/main/java/ of your rest-client project.
* Create a Java class named RestClientRunner.java inside this package:  
  // src/main/java/com/example/restcomm/client/component/RestClientRunner.java  
  package com.example.restcomm.client.component;  
    
  import com.example.restcomm.client.shared.GreetingRequest;  
  import com.example.restcomm.client.shared.GreetingResponse;  
  import org.springframework.boot.CommandLineRunner; // Interface to run code after Spring context is loaded  
  import org.springframework.stereotype.Component;  
  import org.springframework.web.client.RestTemplate; // For making HTTP requests  
    
  /\*\*  
   \* A client component that runs after the Spring Boot application starts.  
   \* It uses RestTemplate to call the remote REST API.  
   \*/  
  @Component  
  public class RestClientRunner implements CommandLineRunner {  
    
   private final RestTemplate restTemplate; // Spring will inject the RestTemplate bean here  
   private static final String SERVER\_BASE\_URL = "http://localhost:8080/api";  
    
   public RestClientRunner(RestTemplate restTemplate) {  
   this.restTemplate = restTemplate;  
   }  
    
   @Override  
   public void run(String... args) throws Exception {  
   System.out.println("\n--- Client Application Started ---");  
   System.out.println("Attempting to call remote REST API using RestTemplate...");  
    
   // --- 1. Perform a GET request ---  
   try {  
   System.out.println("\n--- Calling GET endpoint: " + SERVER\_BASE\_URL + "/greet?name=John ---");  
   String getName = "John";  
   // Use getForObject for simple GET requests expecting an object  
   GreetingResponse getResponse = restTemplate.getForObject(  
   SERVER\_BASE\_URL + "/greet?name={name}",  
   GreetingResponse.class,  
   getName  
   );  
   System.out.println("Client received GET response: " + getResponse);  
   } catch (Exception e) {  
   System.err.println("Client encountered an error calling GET endpoint: " + e.getMessage());  
   }  
    
   // --- 2. Perform a POST request ---  
   try {  
   System.out.println("\n--- Calling POST endpoint: " + SERVER\_BASE\_URL + "/greet ---");  
   GreetingRequest postRequest = new GreetingRequest("Jane");  
   // Use postForObject for POST requests sending an object and expecting an object  
   GreetingResponse postResponse = restTemplate.postForObject(  
   SERVER\_BASE\_URL + "/greet",  
   postRequest,  
   GreetingResponse.class  
   );  
   System.out.println("Client received POST response: " + postResponse);  
   } catch (Exception e) {  
   System.err.println("Client encountered an error calling POST endpoint: " + e.getMessage());  
   }  
    
   System.out.println("\n--- Client Application Finished ---");  
   }  
  }
* **Add application.properties to the Client Application:**

1. In your remoting-client project, navigate to src/main/resources/.
2. Create a new file named application.properties (if it doesn't already exist).
3. Add the following line to this application.properties file:

# src/main/resources/application.properties

server.port=8081

1. Now, when you start the client application, it will attempt to use port 8081 instead of 8080, resolving the port conflict.

### STEP 8: Main Client Application Class

This is the standard Spring Boot application entry point for the client.

* Open your main application class (e.g., RestClientApplication.java in com.example.restcomm.client).  
  // src/main/java/com/example/restcomm/client/RestClientApplication.java  
  package com.example.restcomm.client;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
    
  @SpringBootApplication  
  public class RestClientApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(RestClientApplication.class, args);  
   }  
  }

### STEP 9: Run Both Applications and Test

To see the communication in action, you need to run both the server and client applications simultaneously.

1. **Start the Server Application:**
   * In your IDE, navigate to RestServerApplication.java (in the rest-server project).
   * Run its main method.
   * You should see Spring Boot starting the server on port 8080 (default). Keep this running.
2. **Start the Client Application:**
   * In a separate instance of your IDE (or a new run configuration), navigate to RestClientApplication.java (in the rest-client project).
   * Run its main method.
   * **Observe the console output of both applications.**

**Expected Output:**

* **Server Console (rest-server):** You should see messages indicating it received GET and POST requests, e.g.:  
  Server: Received GET request for greeting for: John  
  Server: Received POST request for greeting for: Jane
* **Client Console (rest-client):** You should see messages indicating the responses received from the server, e.g.:  
  --- Client Application Started ---  
  Attempting to call remote REST API using RestTemplate...  
    
  --- Calling GET endpoint: http://localhost:8080/api/greet?name=John ---  
  Client received GET response: GreetingResponse(message=Hello, John from REST server (GET)!, timestamp=YYYY-MM-DD HH:MM:SS)  
    
  --- Calling POST endpoint: http://localhost:8080/api/greet ---  
  Client received POST response: GreetingResponse(message=Hello, Jane from REST server (POST)!, timestamp=YYYY-MM-DD HH:MM:SS)  
    
  --- Client Application Finished ---

This confirms that the client successfully communicated with the REST API on the server using RestTemplate.

You have successfully implemented HTTP-based communication using Spring's RestTemplate! You've learned how to:

* Create a simple Spring Boot REST API on the server.
* Define DTOs for data exchange.
* Configure and use RestTemplate on the client-side to make GET and POST requests.
* Observe the interaction between a client and a server via standard HTTP.

RestTemplate is a versatile tool for consuming various web services and is a fundamental part of building microservices that communicate over HTTP.

# Activity 9.1: Spring with JMS “Messaging with ActiveMQ”

This activity will guide you through implementing messaging with Spring and JMS (Java Message Service), using Apache ActiveMQ as the message broker. You will learn how to send messages, receive messages, and implement a request-reply messaging pattern.

### STEP 1: Set Up Apache ActiveMQ Broker

Before starting our Spring Boot applications, we need a running JMS broker. Apache ActiveMQ is a popular open-source choice.

1. **Download ActiveMQ:**
   * Go to the Apache ActiveMQ download page: [https://activemq.apache.org/components/classic/download/activemq-5-18-3-release-notes](https://www.google.com/search?q=https://activemq.apache.org/components/classic/download/activemq-5-18-3-release-notes) (or find the latest stable release).
   * Download the binary distribution (e.g., apache-activemq-5.18.3-bin.zip).
2. **Extract and Start ActiveMQ:**
   * Unzip the downloaded file to a convenient location (e.g., C:\activemq or ~/activemq).
   * Open a command prompt or terminal.
   * Navigate to the bin directory inside the extracted ActiveMQ folder:
     + **Windows:** cd C:\activemq\apache-activemq-5.18.3\bin\win64 (or win32)
     + **Linux/macOS:** cd ~/activemq/apache-activemq-5.18.3/bin
   * Run the ActiveMQ start command:
     + **Windows:** activemq start
     + **Linux/macOS:** ./activemq start
   * You should see output indicating that ActiveMQ has started.
   * **Verify:** Open your web browser and go to ActiveMQ's web console: http://localhost:8161/. The default username/password is admin/admin. You should see the ActiveMQ dashboard. Keep this running throughout the activity.

### STEP 2: Project Setup (Spring Boot JMS Application)

We'll create a single Spring Boot application that acts as both a message sender and receiver to simplify the setup.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.jms
   * **Artifact:** spring-jms-demo
   * **Name:** spring-jms-demo
   * **Description:** Spring JMS Demo with ActiveMQ
   * **Package Name:** com.example.jms
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** For a simple REST controller to trigger message sending.
   * **Spring for Apache ActiveMQ 5:** This is the crucial dependency for JMS integration with ActiveMQ.
   * **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

### STEP 3: Configure JMS Connection in application.properties

We need to tell Spring Boot how to connect to our ActiveMQ broker.

1. **Open application.properties:**
   * Navigate to src/main/resources/ and open application.properties.
2. **Add ActiveMQ Connection Properties:**
   * Add the following lines:

# application.properties  
# ActiveMQ broker URL (default for ActiveMQ)  
spring.activemq.broker-url=tcp://localhost:61616  
  
# Enable JMS listener auto-startup  
spring.jms.listener.auto-startup=true  
  
# Enable sending messages to non-existent destinations (for demo simplicity)  
spring.jms.template.explicit-qos-enabled=true  
spring.jms.template.delivery-persistent=true

* + tcp://localhost:61616 is the default TCP port for ActiveMQ.
  + spring.jms.listener.auto-startup=true ensures our message listeners start automatically.

### STEP 4: Create a Message Sender (Producer)

We'll create a service that uses JmsTemplate to send messages to a queue.

* Create a new package com.example.jms.service in src/main/java/.
* Create a Java class named MessageSender.java inside this package:  
  // src/main/java/com/example/jms/service/MessageSender.java  
  package com.example.jms.service;  
    
  import org.springframework.jms.core.JmsTemplate; // Core class for sending JMS messages  
  import org.springframework.stereotype.Service;  
    
  /\*\*  
   \* Service responsible for sending messages to JMS destinations.  
   \*/  
  @Service  
  public class MessageSender {  
    
   private final JmsTemplate jmsTemplate;  
    
   // Spring automatically injects JmsTemplate  
   public MessageSender(JmsTemplate jmsTemplate) {  
   this.jmsTemplate = jmsTemplate;  
   }  
    
   /\*\*  
   \* Sends a simple text message to a specified queue.  
   \* @param destination The name of the queue.  
   \* @param message The text content of the message.  
   \*/  
   public void sendMessage(String destination, String message) {  
   System.out.println("Sending message to queue '" + destination + "': " + message);  
   jmsTemplate.convertAndSend(destination, message); // convertAndSend handles serialization  
   }  
    
   /\*\*  
   \* Sends a message and waits for a reply (Request-Reply Messaging).  
   \* @param requestDestination The queue to send the request to.  
   \* @param replyDestination The queue to expect the reply from.  
   \* @param requestMessage The request message content.  
   \* @return The reply message content.  
   \*/  
   public String sendAndReceive(String requestDestination, String replyDestination, String requestMessage) {  
   System.out.println("Sending request to '" + requestDestination + "': " + requestMessage);  
   System.out.println("Expecting reply on '" + replyDestination + "'");  
    
   // Send the message and specify the reply-to destination  
   // The convertAndSend method with MessagePostProcessor allows setting JMS headers  
   jmsTemplate.convertAndSend(requestDestination, requestMessage, message -> {  
   message.setJMSReplyTo(jmsTemplate.getConnectionFactory().createContext().createQueue(replyDestination));  
   return message;  
   });  
    
   // Receive the reply message from the reply destination  
   // receiveAndConvert blocks until a message is available or timeout occurs  
   Object reply = jmsTemplate.receiveAndConvert(replyDestination);  
    
   if (reply != null) {  
   System.out.println("Received reply from '" + replyDestination + "': " + reply);  
   return reply.toString();  
   } else {  
   System.out.println("No reply received from '" + replyDestination + "'.");  
   return "No reply";  
   }  
   }  
  }

### STEP 5: Create a Message Receiver (Consumer)

We'll create a component that listens for messages from a queue.

* Create a new package com.example.jms.consumer in src/main/java/.
* Create a Java class named MessageReceiver.java inside this package:  
  // src/main/java/com/example/jms/consumer/MessageReceiver.java  
  package com.example.jms.consumer;  
    
  import org.springframework.jms.annotation.JmsListener; // Annotation for message-driven POJOs  
  import org.springframework.stereotype.Component;  
    
  /\*\*  
   \* Component responsible for receiving messages from JMS destinations.  
   \*/  
  @Component  
  public class MessageReceiver {  
    
   /\*\*  
   \* Listens for messages on the "myQueue" destination.  
   \* This method acts as a Message-Driven POJO (MDP).  
   \* @param message The received text message.  
   \*/  
   @JmsListener(destination = "myQueue") // Listens to "myQueue"  
   public void receiveMessage(String message) {  
   System.out.println("Received message from 'myQueue': " + message);  
   }  
    
   /\*\*  
   \* Listens for request messages on "requestQueue" and sends a reply to the JMSReplyTo destination.  
   \* This demonstrates the Request-Reply Messaging pattern.  
   \* @param requestMessage The received request message.  
   \* @return The reply message.  
   \*/  
   @JmsListener(destination = "requestQueue") // Listens to "requestQueue"  
   public String handleRequest(String requestMessage) {  
   System.out.println("Received request from 'requestQueue': " + requestMessage);  
   String replyMessage = "Reply to: " + requestMessage.toUpperCase();  
   System.out.println("Sending reply: " + replyMessage);  
   return replyMessage; // Spring automatically sends this as a reply to JMSReplyTo  
   }  
  }

### STEP 6: Create a REST Controller to Trigger Messaging

We'll create a simple REST controller to easily trigger sending messages via HTTP requests.

* Create a new package com.example.jms.controller in src/main/java/.
* Create a Java class named JmsController.java inside this package:  
  // src/main/java/com/example/jms/controller/JmsController.java  
  package com.example.jms.controller;  
    
  import com.example.jms.service.MessageSender;  
  import org.springframework.http.ResponseEntity;  
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.RequestParam;  
  import org.springframework.web.bind.annotation.RestController;  
    
  /\*\*  
   \* REST Controller to trigger JMS message sending.  
   \*/  
  @RestController  
  public class JmsController {  
    
   private final MessageSender messageSender;  
    
   public JmsController(MessageSender messageSender) {  
   this.messageSender = messageSender;  
   }  
    
   /\*\*  
   \* Endpoint to send a message to "myQueue".  
   \* Example: GET http://localhost:8080/send?message=HelloJMS  
   \* @param message The message content to send.  
   \* @return A confirmation message.  
   \*/  
   @GetMapping("/send")  
   public ResponseEntity<String> sendMessage(@RequestParam String message) {  
   messageSender.sendMessage("myQueue", message);  
   return ResponseEntity.ok("Message sent to myQueue: " + message);  
   }  
    
   /\*\*  
   \* Endpoint to send a request message to "requestQueue" and receive a reply from "replyQueue".  
   \* Example: GET http://localhost:8080/request-reply?message=Ping  
   \* @param message The request message content.  
   \* @return The reply received from the consumer.  
   \*/  
   @GetMapping("/request-reply")  
   public ResponseEntity<String> sendRequestAndReceiveReply(@RequestParam String message) {  
   String reply = messageSender.sendAndReceive("requestQueue", "replyQueue", message);  
   return ResponseEntity.ok("Request sent, Reply received: " + reply);  
   }  
  }

### STEP 7: Main Spring Boot Application Class

This is the standard Spring Boot application entry point. We need to enable JMS listening.

* Open your main application class (e.g., SpringJmsDemoApplication.java in com.example.jms.app).  
  // src/main/java/com/example/jms/app/SpringJmsDemoApplication.java  
  package com.example.jms;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
  import org.springframework.jms.annotation.EnableJms; // Enables JMS listener annotation processing  
    
  /\*\*  
   \* Main Spring Boot application class for JMS demo.  
   \* @EnableJms: Activates detection of @JmsListener annotations.  
   \*/  
  @SpringBootApplication  
  @EnableJms // This annotation is crucial for @JmsListener to work  
  public class SpringJmsDemoApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(SpringJmsDemoApplication.class, args);  
   }  
  }

### STEP 8: Run the Application and Test JMS Messaging

1. **Ensure ActiveMQ is Running:**
   * Double-check that your Apache ActiveMQ broker is running (from STEP 1). You should be able to access its web console at http://localhost:8161/.
2. **Run the Spring Boot Application:**
   * Open your main application class (SpringJmsDemoApplication.java).
   * Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
   * You should see Spring Boot starting on port 8080.
3. **Test Sending and Receiving Messages:**
   * **Open your IDE's console for the spring-jms-demo application.** This is where you'll see both sent and received messages.
   * **Test Sending Messages (Point-to-Point):**
     + Open your web browser and go to: http://localhost:8080/send?message=HelloFromWeb
     + **Observe:**
       - In your application's console, you should see:  
         Sending message to queue 'myQueue': HelloFromWeb  
         Received message from 'myQueue': HelloFromWeb
       - In the ActiveMQ web console (http://localhost:8161/ -> Queues tab), you should see myQueue appear (if it didn't exist before) and its message count briefly increment and then decrement (as the message is consumed).
   * **Test Request-Reply Messaging:**
     + Open your web browser and go to: http://localhost:8080/request-reply?message=RequestForReply
     + **Observe:**
       - In your application's console, you should see:  
         Sending request to 'requestQueue': RequestForReply  
         Expecting reply on 'replyQueue'  
         Received request from 'requestQueue': RequestForReply  
         Sending reply: REPLY TO: REQUESTFORREPLY  
         Received reply from 'replyQueue': REPLY TO: REQUESTFORREPLY
       - In the ActiveMQ web console, you should see requestQueue and replyQueue appear, with messages flowing through them.

You have successfully implemented messaging with Spring JMS and Apache ActiveMQ! You've learned how to:

* Set up and run a local ActiveMQ broker.
* Configure Spring Boot to connect to ActiveMQ.
* Send messages using JmsTemplate.
* Receive messages using @JmsListener (Message-Driven POJOs).
* Implement a request-reply messaging pattern.

This activity provides a practical understanding of how Spring simplifies asynchronous communication using JMS.

# Activity 10.1: Spring JavaMail “Sending Emails in Spring Boot”

This activity will guide you through integrating email functionality into a Spring Boot application using Spring's JavaMail support. You will learn how to send simple text emails, rich HTML emails with attachments, and implement asynchronous email sending, along with basic error handling.

### STEP 1: Project Setup (Spring Boot Application)

We'll use Spring Initializr to set up a new Spring Boot project with the necessary mail dependencies.

1. **Go to Spring Initializr:** Open your web browser and navigate to <https://start.spring.io/>.
2. **Configure Your Project:**
   * **Project:** Maven Project
   * **Language:** Java
   * **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   * **Group:** com.example.mail
   * **Artifact:** spring-mail-demo
   * **Name:** spring-mail-demo
   * **Description:** Spring JavaMail Demo
   * **Package Name:** com.example.mail
   * **Packaging:** Jar
   * **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   * **Spring Web:** For a simple REST controller to trigger email sending.
   * **Java Mail Sender (Spring Boot Starter Mail):** This is the core dependency for Spring Mail integration.
   * **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

### STEP 2: Configure Mail Server Properties

We need to tell Spring Boot how to connect to your SMTP (Simple Mail Transfer Protocol) server. We'll use Gmail's SMTP settings as an example, but you can replace them with your own mail server details.

**Important Note for Gmail:** If you use Gmail, you might need to generate an "App password" instead of using your regular Gmail password, especially if you have 2-Step Verification enabled. Go to your Google Account -> Security -> How you sign in to Google -> App passwords.

1. **Open application.properties:**
   * Navigate to src/main/resources/ and open application.properties.
2. **Add Mail Server Configuration:**
   * Add the following properties. **Replace your-email@gmail.com and your-app-password with your actual Gmail address and generated app password.**

# application.properties  
# SMTP Host (e.g., Gmail's SMTP server)  
spring.mail.host=smtp.gmail.com  
# SMTP Port (587 for TLS, 465 for SSL)  
spring.mail.port=587  
# Your Gmail address  
spring.mail.username=your-email@gmail.com  
# Your Gmail App Password (or regular password if 2FA is off, not recommended)  
spring.mail.password=your-app-password  
# Enable authentication  
spring.mail.properties.mail.smtp.auth=true  
# Enable STARTTLS for secure connection  
spring.mail.properties.mail.smtp.starttls.enable=true  
# Optional: for debugging mail sessions  
# spring.mail.properties.mail.debug=true

### STEP 3: Create a Mail Service for Sending Emails

We'll create a service class that encapsulates the email sending logic using JavaMailSender.

* Create a new package com.example.mail.service in src/main/java/.
* Create a Java class named EmailService.java inside this package:  
  // src/main/java/com/example/mail/service/EmailService.java  
  package com.example.mail.service;  
    
  import jakarta.mail.MessagingException;  
  import jakarta.mail.internet.MimeMessage;  
  import org.springframework.beans.factory.annotation.Value; // Import for @Value annotation  
  import org.springframework.core.io.FileSystemResource; // For attachments  
  import org.springframework.mail.MailException; // Spring's mail exception  
  import org.springframework.mail.SimpleMailMessage; // For simple text emails  
  import org.springframework.mail.javamail.JavaMailSender; // Core Spring Mail interface  
  import org.springframework.mail.javamail.MimeMessageHelper; // For complex emails (HTML, attachments)  
  import org.springframework.scheduling.annotation.Async; // For asynchronous sending  
  import org.springframework.stereotype.Service;  
    
  import java.io.File;  
    
  /\*\*  
   \* Service class for sending various types of emails.  
   \*/  
  @Service  
  public class EmailService {  
    
   private final JavaMailSender mailSender;  
    
   @Value("${spring.mail.username}") // Inject the sender's email from properties  
   private String senderEmail;  
    
   // Spring automatically injects JavaMailSender configured from application.properties  
   public EmailService(JavaMailSender mailSender) {  
   this.mailSender = mailSender;  
   }  
    
   /\*\*  
   \* Sends a simple text email.  
   \* @param to The recipient's email address.  
   \* @param subject The subject of the email.  
   \* @param text The plain text content of the email.  
   \*/  
   public void sendSimpleEmail(String to, String subject, String text) {  
   try {  
   SimpleMailMessage message = new SimpleMailMessage();  
   message.setFrom(senderEmail); // Use the injected senderEmail  
   message.setTo(to);  
   message.setSubject(subject);  
   message.setText(text);  
   mailSender.send(message);  
   System.out.println("Simple email sent successfully to " + to);  
   } catch (MailException e) {  
   System.err.println("Failed to send simple email to " + to + ": " + e.getMessage());  
   // In a real application, you might log the error, retry, or notify an admin.  
   }  
   }  
    
   /\*\*  
   \* Sends a rich HTML email with optional attachment.  
   \* @param to The recipient's email address.  
   \* @param subject The subject of the email.  
   \* @param htmlContent The HTML content of the email.  
   \* @param attachmentPath Optional path to a file to attach.  
   \*/  
   public void sendComplexEmail(String to, String subject, String htmlContent, String attachmentPath) {  
   try {  
   MimeMessage message = mailSender.createMimeMessage();  
   // true for multipart message (e.g., HTML + attachment)  
   MimeMessageHelper helper = new MimeMessageHelper(message, true);  
    
   helper.setFrom(senderEmail); // Use the injected senderEmail  
   helper.setTo(to);  
   helper.setSubject(subject);  
   helper.setText(htmlContent, true); // true indicates HTML content  
    
   if (attachmentPath != null && !attachmentPath.isEmpty()) {  
   File attachment = new File(attachmentPath);  
   if (attachment.exists()) {  
   FileSystemResource file = new FileSystemResource(attachment);  
   helper.addAttachment(file.getFilename(), file);  
   System.out.println("Attached file: " + file.getFilename());  
   } else {  
   System.err.println("Attachment file not found: " + attachmentPath);  
   }  
   }  
    
   mailSender.send(message);  
   System.out.println("Complex email sent successfully to " + to);  
   } catch (MessagingException | MailException e) {  
   System.err.println("Failed to send complex email to " + to + ": " + e.getMessage());  
   e.printStackTrace(); // Print stack trace for more details on MessagingException  
   }  
   }  
    
   /\*\*  
   \* Sends a simple text email asynchronously.  
   \* The method will return immediately, and email sending will happen in a separate thread.  
   \* @param to The recipient's email address.  
   \* @param subject The subject of the email.  
   \* @param text The plain text content of the email.  
   \*/  
   @Async // Marks this method to be executed in a separate thread  
   public void sendSimpleEmailAsync(String to, String subject, String text) {  
   System.out.println("Attempting to send simple email asynchronously to " + to + " from thread: " + Thread.currentThread().getName());  
   sendSimpleEmail(to, subject, text); // Re-use the synchronous sending logic  
   }  
    
   /\*\*  
   \* Sends a rich HTML email with optional attachment asynchronously.  
   \* @param to The recipient's email address.  
   \* @param subject The subject of the email.  
   \* @param htmlContent The HTML content of the email.  
   \* @param attachmentPath Optional path to a file to attach.  
   \*/  
   @Async  
   public void sendComplexEmailAsync(String to, String subject, String htmlContent, String attachmentPath) {  
   System.out.println("Attempting to send complex email asynchronously to " + to + " from thread: " + Thread.currentThread().getName());  
   sendComplexEmail(to, subject, htmlContent, attachmentPath); // Re-use the synchronous sending logic  
   }  
  }

### STEP 4: Create a REST Controller to Trigger Email Sending

We'll create a simple REST controller to easily trigger email sending via HTTP requests.

* Create a new package com.example.mail.controller in src/main/java/.
* Create a Java class named MailController.java inside this package:  
  // src/main/java/com/example/mail/controller/MailController.java  
  package com.example.mail.controller;  
    
  import com.example.mail.service.EmailService;  
  import org.springframework.http.ResponseEntity;  
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.RequestParam;  
  import org.springframework.web.bind.annotation.RestController;  
    
  /\*\*  
   \* REST Controller to trigger email sending for demonstration.  
   \*/  
  @RestController  
  public class MailController {  
    
   private final EmailService emailService;  
    
   public MailController(EmailService emailService) {  
   this.emailService = emailService;  
   }  
    
   /\*\*  
   \* Endpoint to send a simple text email.  
   \* Example: GET http://localhost:8080/send-simple?to=recipient@example.com&subject=Test&body=Hello  
   \* @param to Recipient email address.  
   \* @param subject Email subject.  
   \* @param body Email body text.  
   \* @return Confirmation message.  
   \*/  
   @GetMapping("/send-simple")  
   public ResponseEntity<String> sendSimple(@RequestParam String to,  
   @RequestParam String subject,  
   @RequestParam String body) {  
   emailService.sendSimpleEmail(to, subject, body);  
   return ResponseEntity.ok("Simple email sending initiated to " + to);  
   }  
    
   /\*\*  
   \* Endpoint to send a simple text email asynchronously.  
   \* Example: GET http://localhost:8080/send-simple-async?to=recipient@example.com&subject=AsyncTest&body=HelloAsync  
   \* @param to Recipient email address.  
   \* @param subject Email subject.  
   \* @param body Email body text.  
   \* @return Confirmation message.  
   \*/  
   @GetMapping("/send-simple-async")  
   public ResponseEntity<String> sendSimpleAsync(@RequestParam String to,  
   @RequestParam String subject,  
   @RequestParam String body) {  
   emailService.sendSimpleEmailAsync(to, subject, body);  
   return ResponseEntity.ok("Asynchronous simple email sending initiated to " + to + ". Check console for thread info.");  
   }  
    
   /\*\*  
   \* Endpoint to send a complex HTML email with an optional attachment.  
   \* Example: GET http://localhost:8080/send-complex?to=recipient@example.com&subject=HTMLTest&body=<b>Hello</b><i>HTML</i>&attach=true  
   \* @param to Recipient email address.  
   \* @param subject Email subject.  
   \* @param body Email HTML content.  
   \* @param attach Whether to include an attachment (true/false).  
   \* @return Confirmation message.  
   \*/  
   @GetMapping("/send-complex")  
   public ResponseEntity<String> sendComplex(@RequestParam String to,  
   @RequestParam String subject,  
   @RequestParam String body,  
   @RequestParam(defaultValue = "false") boolean attach) {  
   String htmlContent = "<html><body><h1>" + body + "</h1><p>This is an <b>HTML</b> email from Spring Boot!</p></body></html>";  
   String attachmentPath = null;  
   if (attach) {  
   // Create a dummy file for attachment  
   // Ensure this file exists in src/main/resources or a path accessible to the application  
   // For demonstration, let's assume a file named 'dummy-attachment.txt' in resources  
   // You might need to create this file manually for the demo to work.  
   // Path will be relative to the project root or classpath.  
   attachmentPath = "src/main/resources/dummy-attachment.txt";  
   // Or for a file directly in resources:  
   // attachmentPath = getClass().getClassLoader().getResource("dummy-attachment.txt").getFile();  
   }  
   emailService.sendComplexEmail(to, subject, htmlContent, attachmentPath);  
   return ResponseEntity.ok("Complex email sending initiated to " + to + (attach ? " with attachment." : "."));  
   }  
    
   /\*\*  
   \* Endpoint to send a complex HTML email with an optional attachment asynchronously.  
   \* Example: GET http://localhost:8080/send-complex-async?to=recipient@example.com&subject=AsyncHTMLTest&body=<b>Hello</b><i>AsyncHTML</i>&attach=true  
   \* @param to Recipient email address.  
   \* @param subject Email subject.  
   \* @param body Email HTML content.  
   \* @param attach Whether to include an attachment (true/false).  
   \* @return Confirmation message.  
   \*/  
   @GetMapping("/send-complex-async")  
   public ResponseEntity<String> sendComplexAsync(@RequestParam String to,  
   @RequestParam String subject,  
   @RequestParam String body,  
   @RequestParam(defaultValue = "false") boolean attach) {  
   String htmlContent = "<html><body><h1>" + body + "</h1><p>This is an <b>HTML</b> email from Spring Boot (Async)!</p></body></html>";  
   String attachmentPath = null;  
   if (attach) {  
   attachmentPath = "src/main/resources/dummy-attachment.txt";  
   }  
   emailService.sendComplexEmailAsync(to, subject, htmlContent, attachmentPath);  
   return ResponseEntity.ok("Asynchronous complex email sending initiated to " + to + (attach ? " with attachment." : ".") + " Check console for thread info.");  
   }  
  }  
    
  **Action Required:** For the attachment example to work, create a simple text file named dummy-attachment.txt inside your src/main/resources directory. You can put any content in it, e.g., "This is a test attachment."

### STEP 5: Main Spring Boot Application Class

This is the standard Spring Boot application entry point. To enable asynchronous email sending, we need to add @EnableAsync.

* Open your main application class (e.g., SpringMailDemoApplication.java in com.example.mail).  
  // src/main/java/com/example/mail/app/SpringMailDemoApplication.java  
  package com.example.mail;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
  import org.springframework.scheduling.annotation.EnableAsync; // Enables asynchronous method execution  
    
  /\*\*  
   \* Main Spring Boot application class for JavaMail demo.  
   \* @EnableAsync: Activates Spring's asynchronous method execution capability.  
   \*/  
  @SpringBootApplication  
  @EnableAsync // This annotation is crucial for @Async to work  
  public class SpringMailDemoApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.run(SpringMailDemoApplication.class, args);  
   }  
  }

### STEP 6: Run the Application and Test Email Sending

1. **Ensure Mail Server Configuration is Correct:**
   * Double-check your application.properties for spring.mail.username and spring.mail.password. Make sure the password is an "App password" if you're using Gmail with 2-Step Verification. Incorrect credentials are the most common cause of errors.
2. **Create Dummy Attachment File:**
   * If you plan to test attachments, ensure you have created src/main/resources/dummy-attachment.txt with some content.
3. **Run the Spring Boot Application:**
   * Open your main application class (SpringMailDemoApplication.java).
   * Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
   * You should see Spring Boot starting on port 8080.
4. **Test Email Sending:**
   * **Open your IDE's console for the spring-mail-demo application.** This is where you'll see logs about email sending status and asynchronous thread information.
   * **Open the inbox of the recipient@example.com email address** you specified in the URLs.
   * **Test Sending Simple Emails:**
     + Open your web browser and go to:  
       http://localhost:8080/send-simple?to=your-recipient-email@example.com&subject=MyFirstSpringEmail&body=Hello%20from%20Spring%20Boot%20JavaMail!
     + **Observe:** Check your application console for "Simple email sent successfully" and the recipient's inbox for the email.
   * **Test Sending Rich (HTML) Emails with Attachment:**
     + Open your web browser and go to:  
       http://localhost:8080/send-complex?to=your-recipient-email@example.com&subject=HTMLWithAttachment&body=Welcome%20to%20HTML%20Email&attach=true
     + **Observe:** Check your application console and the recipient's inbox for an HTML email with the dummy-attachment.txt file.
   * **Test Asynchronous Email Sending (Simple):**
     + Open your web browser and go to:  
       http://localhost:8080/send-simple-async?to=your-recipient-email@example.com&subject=AsyncSimpleEmail&body=This%20is%20an%20async%20simple%20email.
     + **Observe:** You should immediately see "Asynchronous simple email sending initiated..." in your browser. In the console, notice that the "Attempting to send simple email asynchronously..." message appears from a different thread (e.g., task-1). The email should still arrive in the recipient's inbox.
   * **Test Asynchronous Email Sending (Complex):**
     + Open your web browser and go to:  
       http://localhost:8080/send-complex-async?to=your-recipient-email@example.com&subject=AsyncComplexEmail&body=Async%20HTML%20Test&attach=true
     + **Observe:** Similar to the simple async email, the browser response is immediate, and the email sending occurs in a background thread.

You have successfully implemented email sending functionality in a Spring Boot application using Spring JavaMail! You've learned how to:

* Configure Spring Boot for email sending using application.properties.
* Send simple plain text emails using SimpleMailMessage.
* Send rich HTML emails and include attachments using MimeMessageHelper.
* Implement asynchronous email sending using @Async for non-blocking operations.
* Understand basic error handling for mail operations.

This activity provides a solid foundation for integrating robust email capabilities into your Spring applications.