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
```

3. **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;
```

```java
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:

```java
// 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>
```

2. **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>
```

3. **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>
```

4. **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>
```

5. **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