6. **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
```

```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).