AOP

Spring Boot AOP (Aspect-Oriented Programming)

1. Introduction to AOP

- **Definition:**

  Aspect-Oriented Programming is a programming paradigm that separates **cross-cutting concerns** (e.g., logging, security, and transactions) from the main business logic of the application.

- **Why Use AOP?**
  - Simplifies code maintenance.
  - Eliminates code duplication for repetitive tasks (e.g., logging in every method).
  - Improves modularity by separating concerns.
- **Spring AOP:**

  Spring AOP uses runtime proxies to implement aspects, enabling modular handling of cross-cutting concerns.

---

2. Key Concepts in AOP

Concept	Description
Aspect	A module for handling a cross-cutting concern. Example: LoggingAspect for logging functionality.
Join Point	A point in application execution, such as a method call, where an aspect can be applied.
Advice	Action taken by an aspect at a specific join point. (e.g., logging before a method runs).
Pointcut	A predicate expression that matches join points where advice should be applied.
Weaving	The process of linking aspects with the target object (done at runtime in Spring).

3. Types of Advice in AOP

a. Before Advice

- **Definition:** Executes **before** the target method runs.
- **Use Case:** Validate input, log method entry.
- **Example:**

```
@Before("execution(* com.example.service.MyService.perform
Task(..))")
public void logBefore() {
    System.out.println("Before Advice: Method execution st
arted.");
}
```

b. After Advice

- **Definition:** Executes **after** the target method completes, regardless of the outcome.
- **Use Case:** Cleanup resources, log method completion.
- **Example:**

```
@After("execution(* com.example.service.MyService.performT
ask(..))")
public void logAfter() {
    System.out.println("After Advice: Method execution com
pleted.");
}
```

c. AfterReturning Advice

- **Definition:** Executes **after the method returns successfully**.
- **Use Case:** Log the return value or modify it.
- **Example:**

```
@AfterReturning(pointcut = "execution(* com.example.servic
e.CalculationService.addNumbers(..))", returning = "resul
t")
public void logAfterReturning(Object result) {
    System.out.println("AfterReturning Advice: Returned va
lue - " + result);
}
```

d. AfterThrowing Advice

- **Definition:** Executes **if the method throws an exception**.
- **Use Case:** Handle or log exceptions.
- **Example:**

```
@AfterThrowing(pointcut = "execution(* com.example.servic
e.MyService.performTask(..))", throwing = "exception")
public void logAfterThrowing(Exception exception) {
    System.out.println("AfterThrowing Advice: Exception oc
curred - " + exception.getMessage());
}
```

e. Around Advice

- **Definition:** Executes **before and after** the target method.

- **Use Case:** Monitor performance, log inputs and outputs, alter method flow.
- **Example:**

```
@Around("execution(* com.example.service.MyService.performTask(..))")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("Around Advice: Before method execution.");
    Object result = joinPoint.proceed();
    System.out.println("Around Advice: After method execution.");
    return result;
}
```

4. Pointcut Expressions

Common Patterns

Expression	Description
`execution(* com.example.service.*.*(..))`	Matches all methods in `com.example.service` package.
`execution(* com.example..*.*(..))`	Matches all methods in `com.example` package and sub-packages.
`within(com.example.service..*)`	Matches all methods in the `com.example.service` package and its sub-packages.
`bean(myBean)`	Matches all methods of the bean named `myBean`.

5. Real-World Use Cases

a. Logging

Automatically log method execution details like inputs, outputs, and execution time.

```
@Aspect
@Component
public class LoggingAspect {

    @Around("execution(* com.example.service.*.*(..))")
    public Object logExecution(ProceedingJoinPoint joinPoint)
throws Throwable {
        System.out.println("Method: " + joinPoint.getSignatur
e());
        System.out.println("Input Arguments: " + java.util.Ar
rays.toString(joinPoint.getArgs()));
        Object result = joinPoint.proceed();
        System.out.println("Return Value: " + result);
        return result;
    }
}
```

b. Security

Restrict method access based on user roles.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RoleCheck {
    String role() default "USER";
}
```

```
@Aspect
@Component
public class SecurityAspect {

    @Before("@annotation(roleCheck)")
    public void checkRole(RoleCheck roleCheck) {
        if (!currentUserRole.equals(roleCheck.role())) {
            throw new SecurityException("Access Denied!");
        }
    }
}
```

c. Performance Monitoring

Measure execution time of methods.

```
@Aspect
@Component
public class PerformanceAspect {

    @Around("execution(* com.example.service.*.*(..))")
    public Object monitorPerformance(ProceedingJoinPoint join
Point) throws Throwable {
        long start = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long duration = System.currentTimeMillis() - start;
        System.out.println("Execution time: " + duration + "m
s");
        return result;
    }
```

```
}
```

6. How Spring AOP Works

- **Proxy Creation:** Spring AOP creates proxies of the beans that implement aspects.
- **Runtime Weaving:** Spring weaves aspects into the application during runtime.
- **Annotation-Based Configuration:** Simplifies AOP setup using `@Aspect` and annotations like `@Before`, `@Around`, etc.

7. Benefits of AOP

- Reduces code duplication.
- Improves modularity.
- Eases debugging and maintenance.
- Provides dynamic handling of cross-cutting concerns.

8. Common Mistakes in AOP

- Forgetting to annotate `@Aspect` classes with `@Component`.
- Incorrect Pointcut expressions leading to no advice being applied.
- Using AOP for business logic instead of cross-cutting concerns.

Real-Time Example: Transaction Management with Spring AOP

Use Case:

In a banking application, when transferring money between accounts, multiple operations are performed, such as deducting from one account and adding to

another. To ensure consistency, the entire operation must be wrapped in a **transaction**—either all operations succeed, or none.

Implementation with Spring AOP

Step 1: Add Spring Transaction Dependencies

In `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Step 2: Create the Service

```
package com.example.service;

import com.example.repository.AccountRepository;
import com.example.model.Account;
import org.springframework.beans.factory.annotation.Autowire
d;
import org.springframework.stereotype.Service;
```

```
@Service
public class BankService {

    @Autowired
    private AccountRepository accountRepository;

    public void transferMoney(Long fromAccountId, Long toAcco
untId, double amount) {
        Account fromAccount = accountRepository.findById(from
AccountId)
            .orElseThrow(() -> new IllegalArgumentException
("Invalid fromAccountId"));
        Account toAccount = accountRepository.findById(toAcco
untId)
            .orElseThrow(() -> new IllegalArgumentException
("Invalid toAccountId"));

        if (fromAccount.getBalance() < amount) {
            throw new IllegalArgumentException("Insufficient
balance!");
        }

        fromAccount.setBalance(fromAccount.getBalance() - amo
unt);
        toAccount.setBalance(toAccount.getBalance() + amoun
t);

        accountRepository.save(fromAccount);
        accountRepository.save(toAccount);

        System.out.println("Money transferred successfull
y!");
    }
}
```

Step 3: Create the Aspect

```
package com.example.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.AfterThrowing;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transaction
al;

@Aspect
@Component
public class TransactionAspect {

    @Before("execution(* com.example.service.BankService.tran
sferMoney(..))")
    @Transactional
    public void startTransaction() {
        System.out.println("Transaction started...");
    }

    @AfterThrowing("execution(* com.example.service.BankServi
ce.transferMoney(..))")
    public void rollbackTransaction() {
        System.out.println("Transaction rolled back due to an
error.");
    }
}
```

Step 4: Repository and Model

Account Model

```
package com.example.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Account {

    @Id
    private Long id;
    private String name;
    private double balance;

    // Getters and Setters
}
```

Account Repository

```
package com.example.repository;

import com.example.model.Account;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AccountRepository extends JpaRepository<Acco
unt, Long> {
}
```

Step 5: Application Configuration

Set up an in-memory database in `application.properties`:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Step 6: Test the Application

Test the `transferMoney` method:

```
@SpringBootApplication
public class AopTransactionApplication {

    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(Ao
pTransactionApplication.class, args);
        BankService bankService = context.getBean(BankServic
e.class);

        try {
            bankService.transferMoney(1L, 2L, 500.00);
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage
());
        }
    }
}
```

Output

- **Successful Transaction:**

```
Transaction started...
Money transferred successfully!
```

- **Failed Transaction (Insufficient Balance):**

```
Transaction started...
Transaction rolled back due to an error.
Exception: Insufficient balance!
```