Aspect-Oriented Programming (AOP)

in Java with Spring Boot

Tharindu_Gamlakshage
Tharindushantha4@gmail.com
GitHub:- TharinduShantha98
Software Engineer

Contents

What is AOP? 3

Cross-Cutting Concerns? ***3***

Simplified with a Chart ***3***

Why Do We Need AOP? 3

Key Concepts in AOP 5

How AOP Works Internally 5

✅ ***1. Dynamic Proxies (Proxy-Based AOP)*** ***5***

✅ ***2. AspectJ Weaving*** ***6***

Dynamic Proxies example: 7

Annotations Breakdown ***8***

What is AOP?

Aspect-Oriented Programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.

Cross-Cutting Concerns?

These are functionalities that span across multiple modules or layers of an application but are not the core responsibility of those modules.

Examples include:

- Logging
- Security
- Transaction management
- Error handling
- Monitoring / metrics

Instead of scattering this logic throughout your codebase, AOP lets you encapsulate it into aspects, which can be applied declaratively.

Simplified with a Chart

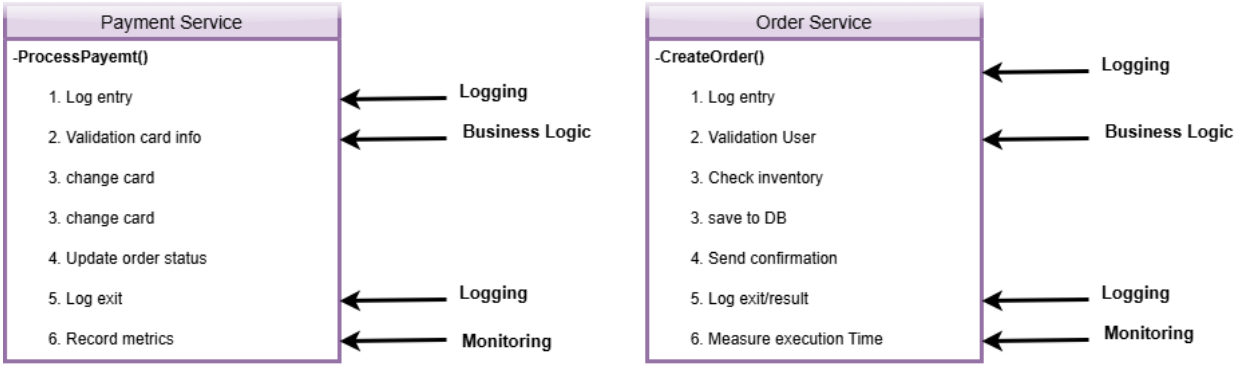
Imagine a typical application where your main code handles business logic, but you also need to handle cross-cutting concerns like logging, security, or performance tracking.

Why Do We Need AOP?

Let’s imagine we want to log every method call in our application. Without AOP, you'd need to manually add logging statements in each method

Without AOP: Scattered Logic

Here's what your code might look like without AOP:

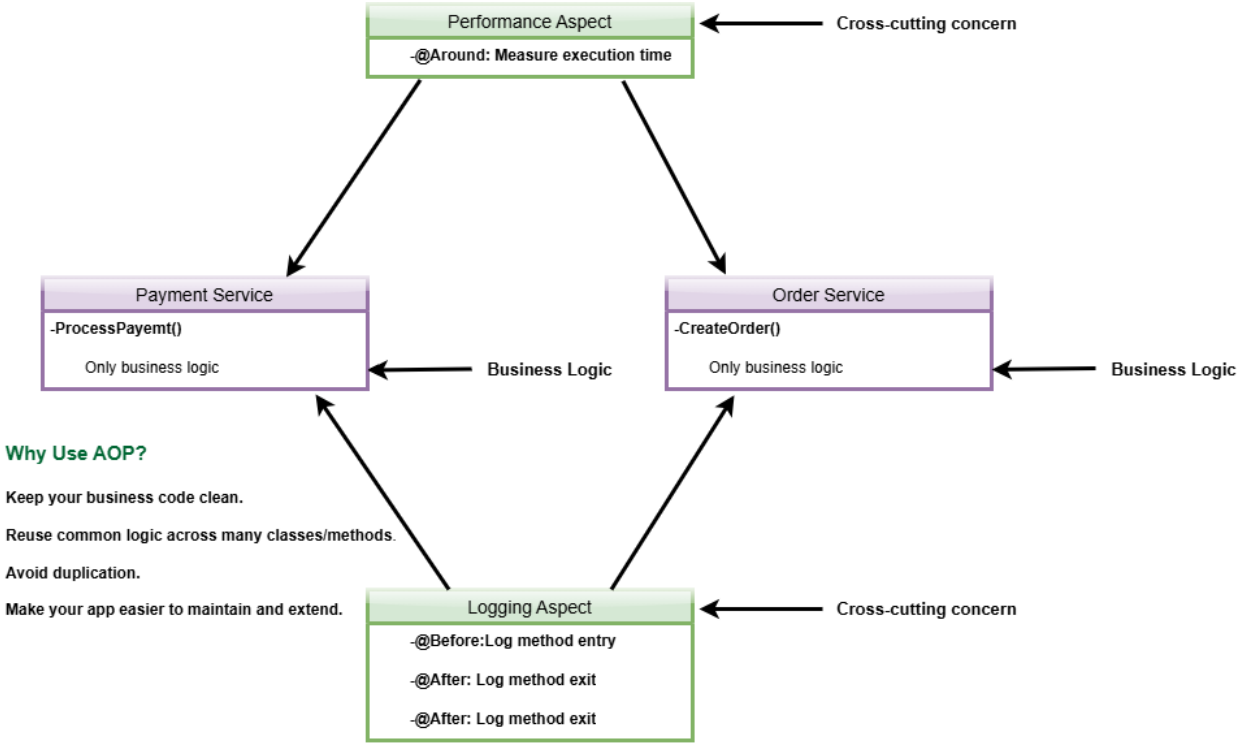


This approach leads to:

- Code duplication
- Tight coupling
- Harder maintenance

With AOP: Centralized Concerns

Now let's apply AOP. We extract the common parts into an aspect.



With AOP, we can extract this repetitive logic into a reusable module — an aspect — and apply it across methods without changing their implementation.

Key Concepts in AOP

TERM	DESCRIPTION
Aspect	A class that encapsulates cross-cutting logic (e.g.,**@Aspect** annotated class)
Join Point	A specific point in the execution of a program, such as method execution or exception throwing.
Pointcut	An expression that selects join points where advice should run.
Advice	The action taken by an aspect at a particular join point. Types:**@Before**,**@After**,**@AfterReturning**,**@AfterThrowing**,**@Around**.
Weaving	The process of applying aspects to a target object to create an advised object. Done at compile time or runtime.

How AOP Works Internally

Spring AOP uses dynamic proxies or AspectJ weaving under the hood.

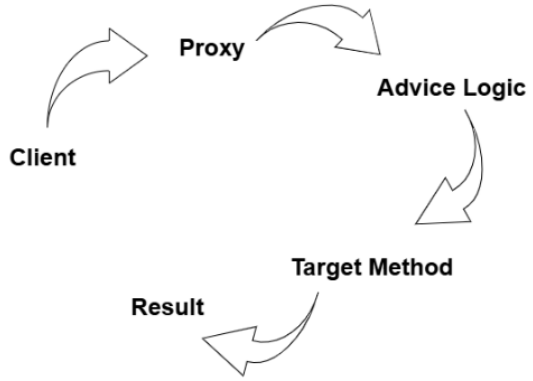
This means that Spring AOP supports two different mechanisms to apply aspects (like logging, security, etc.) around your method calls. These mechanisms are:

☑ 1. Dynamic Proxies (Proxy-Based AOP)

🔍 **What is a Dynamic Proxy?**

A dynamic proxy is a Java feature that allows you to create a proxy object at runtime. This proxy can intercept method calls and execute custom logic before or after the actual method runs.

Here's a simple diagram showing proxy-based AOP:



✅ **Pros**

- Simple and built-in with Spring Boot (**spring-boot-starter-aop**)
- No special building steps required

⚠️ **Cons**

- Only works on Spring-managed beans
- Only applies to public methods
- Cannot intercept constructor calls or internal method calls (within the same class)
- Limited to method execution join points

☑️ 2. AspectJ Weaving

🔍 **What is AspectJ Weaving?**

Unlike proxy-based AOP, AspectJ weaving modifies the bytecode of your classes directly — either during compilation or when they're loaded into the JVM.

There are two common types used with Spring:

> Compile-Time Weaving: Aspects are woven into **.class** files during compilation using the AspectJ compiler (**ajc**).
>
> Load-Time Weaving (LTW): Aspects are woven as classes are loaded by the JVM using a Java agent.

✅ Pros

Can intercept private/protected methods

Supports constructors, fields, static methods

Works outside of Spring context (on third-party libraries)

Handles internal method calls within the same class

⚠️ Cons

More complex setup (especially compile-time weaving)

Slightly higher runtime overhead (especially LTW)

Harder to debug and test

MECHANISM	USED BY	BEST FOR	LIMITATIONS
Dynamic Proxies	Spring AOP	Most typical use cases (logging, simple tracing, etc.)	Most typical use cases (logging, simple tracing, etc.)
AspectJ Weaving	Advanced AOP	Deep interception (constructors, private methods, third-party code)	More complex setup and performance impact

Dynamic Proxies example:

```java
package com.ticket.api.ticketsales_core.aspect;

import ...

@Slf4j  TharinduShantha98 +1
@Aspect
@Component
public class LoggingAspect {

    private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);  3 usages
    private static final ThreadLocal<String> requestIdHolder = new ThreadLocal<>();  no usages
    private HttpServletRequest request;  no usages

    @Before("execution(* com.ticket.api.ticketsales_core..*(..)) &&" +  no usages  TharinduShantha98
            " !within(com.ticket.api.ticketsales_core.filter..*)")
    public void logBeforeMethod(JoinPoint joinPoint) {
        logger.debug("Entering method: {} with arguments: {}", joinPoint.getSignature(), joinPoint.getArgs());
    }

    @AfterReturning(pointcut = "execution(* com.ticket.api.ticketsales_core..*(..)) " +  no usages  TharinduShantha98
            "&& !within(com.ticket.api.ticketsales_core.filter..*)", returning = "result")
    public void logAfterMethod(JoinPoint joinPoint, Object result) {
        logger.debug("Exiting method: {} with result: {}", joinPoint.getSignature(), result);
    }

    @AfterThrowing(pointcut = "execution(* com.ticket.api.ticketsales_core..*(..)) " +  no usages  TharinduShantha98
            "&& !within(com.ticket.api.ticketsales_core.filter..*)", throwing = "exception")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) {
        logger.error("Exception in method: {} with cause: {}", joinPoint.getSignature(), exception.getCause() != null ? except
    }
}
```

Annotations Breakdown

@Aspect: Marks this class as an AOP aspect.

@Component: Registers this class as a Spring bean.

@Slf4j: Lombok annotation for generating a Logger.

@Before: Runs before the matched method.

@AfterReturning: Runs after the method returns successfully.

@AfterThrowing; Runs if the method throws an exception.

Thank you