**🌿 Spring Boot AOP (Aspect Oriented Programming)**

**🧠 Basic Idea**

In a normal Spring Boot project, we usually have **business logic classes** like StudentService, PaymentService, OrderService, etc.

These classes handle **main business rules**, for example:

* StudentService → save student details
* PaymentService → process payment

But, along with these main rules, we often have **extra work** that is **repeated across many classes**, such as:

* Logging method calls
* Security checks
* Starting and committing transactions
* Measuring execution time
* Handling exceptions

If we write this extra code inside every class, it leads to **code duplication** and **less maintainability**.

To solve this, we use **AOP (Aspect Oriented Programming)**.

**🧩 What AOP Does**

AOP allows us to **separate the additional tasks (cross-cutting concerns)** into **separate classes (called Aspects)**.  
Then, we tell Spring **when and where** these tasks should run — before, after, or around certain methods.

Think of it like this:

🎭 *Your business methods act on the stage,*  
*and AOP works behind the curtain, controlling what happens before, after, or around the performance.*

**🔍 Key Concept: Cross-Cutting Concerns**

**Cross-cutting concerns** are pieces of logic that are **not part of the main business logic**, but are needed by **many modules**.

**Examples:**

| **Concern** | **Description** | **Example** |
| --- | --- | --- |
| Logging | Keep track of method calls | Log when a student is saved |
| Security | Check authentication or authorization | Verify user before processing payment |
| Transaction Management | Start/commit/rollback database transactions | Wrap DB operations in a transaction |
| Performance | Measure execution time | Find slow parts of code |
| Data Encoding/Decoding | Encrypt or decrypt data | Encrypt passwords before saving |

Instead of repeating this logic inside every class, we move it to a **separate class (Aspect)** and attach it dynamically where needed.

**⚙️ Main Components of AOP**

**1️⃣ Aspect**

An **Aspect** is a **special class** that holds the logic of a cross-cutting concern.  
It contains one or more **Advice methods**.

In Spring Boot, an Aspect class is marked using @Aspect annotation and registered as a Spring Bean using @Component.

**Example:**

@Aspect

@Component

public class LoggingAspect {

// Advices will be written here

}

Here, LoggingAspect is providing additional service (logging) to the project.

**2️⃣ Advice**

An **Advice** is a **method inside the Aspect class** that defines *what extra logic to run* and *when to run it*.

Each advice runs at a specific point in the execution of your program — such as *before*, *after*, or *around* a method call.

**🕓 Types of Advices (With Deep Explanation)**

Let’s discuss each Advice with an **easy-to-understand example + real-world analogy** 👇

**a) Before Advice (@Before)**

**When it runs:**  
Before the target method (business method) executes.

**Execution order:**  
1️⃣ Advice method  
2️⃣ Target (business) method

**Use Case Example:**  
Checking if a user is logged in before allowing access to a service.

**Analogy:**  
Before you enter an exam hall, the invigilator checks your ID card.  
Only after that, you can sit and write the exam.

**Code Example:**

@Before("execution(\* com.example.service.StudentService.\*(..))")

public void checkLogin() {

System.out.println("Checking user login before method execution...");

}

So, before any method in StudentService runs, the checkLogin() advice will execute.

**b) After Advice (@After)**

**When it runs:**  
After the target method finishes — no matter whether it finishes successfully or throws an exception.

**Execution order:**  
1️⃣ Target (business) method  
2️⃣ Advice method

**Use Case Example:**  
Logging that a method was executed, or closing a connection after a task is done.

**Analogy:**  
After you submit your exam paper, the teacher gives you a “done” stamp — even if your answers were wrong, the stamp happens *after* submission.

**Code Example:**

@After("execution(\* com.example.service.StudentService.\*(..))")

public void logAfterMethod() {

System.out.println("Method execution completed.");

}

**c) Around Advice (@Around)**

**When it runs:**  
Before **and** after the target method.  
It literally “wraps around” the target method call.

**Execution order:**  
1️⃣ Advice (first part) — before calling the method  
2️⃣ Target (business) method  
3️⃣ Advice (second part) — after method finishes

**Use Case Example:**  
To measure execution time or to manage transactions.

**Analogy:**  
Imagine you are timing a race:

* You start the stopwatch (before advice)
* The runner runs (method executes)
* You stop the stopwatch (after advice)
* You print total time (after logic)

**Code Example:**

@Around("execution(\* com.example.service.StudentService.\*(..))")

public Object measureExecutionTime(ProceedingJoinPoint pjp) throws Throwable {

long start = System.currentTimeMillis();

System.out.println("Method execution starting...");

Object result = pjp.proceed(); // execute the actual method

System.out.println("Method execution finished...");

long end = System.currentTimeMillis();

System.out.println("Execution Time: " + (end - start) + " ms");

return result;

}

So here:

* The **first part** (before proceed()) runs before the target method.
* The **second part** (after proceed()) runs after the target method.

**🧠 Why AOP is Powerful**

Without AOP:

public void saveStudent(Student s) {

System.out.println("Checking security...");

System.out.println("Starting transaction...");

// business logic

System.out.println("Committing transaction...");

System.out.println("Logging info...");

}

You repeat these lines in multiple methods.  
Your code becomes messy and hard to maintain.

With AOP:

* These extra lines go into an Aspect class.
* Your business methods stay clean and focused.

So your business class becomes:

public void saveStudent(Student s) {

// Only business logic here

}

AOP automatically adds the other concerns at runtime.

**🔗 How AOP Works in Spring Boot (Simple Flow)**

1️⃣ You mark a class with @Aspect.  
2️⃣ Inside it, you write advice methods (Before, After, Around, etc.).  
3️⃣ You define where to apply them using a **Pointcut expression**, e.g.:

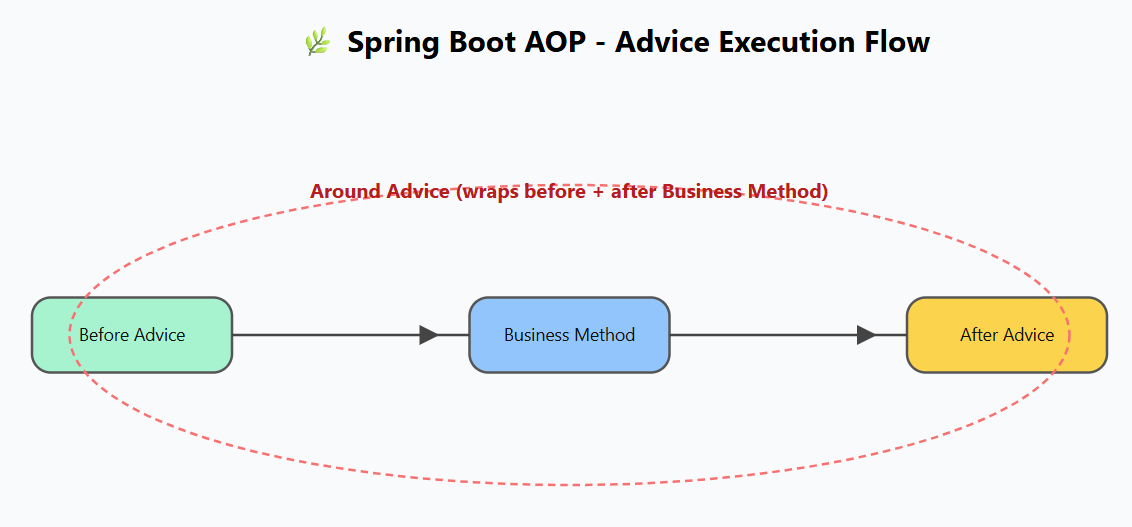
@Before("execution(\* com.example.service.\*.\*(..))")

This means “run this advice before any method in any class inside service package.”

4️⃣ Spring’s **AOP Proxy mechanism** dynamically wraps your beans so that your advices run automatically at the right time.

**✅ Summary Table**

| **Type of Advice** | **Runs When** | **Execution Order** | **Typical Use Case** | **Analogy** |
| --- | --- | --- | --- | --- |
| **Before** | Before method | Advice → Method | Security, validation | ID check before exam |
| **After** | After method (success/fail) | Method → Advice | Logging, cleanup | Stamp after exam |
| **Around** | Before & After | Advice (before) → Method → Advice (after) | Transaction, performance | Stopwatch around a race |



**d) After Returning Advice (@AfterReturning)**

**When it runs:**  
After the business method executes **successfully** — that is, **only if no exception occurs**.

**Execution order:**  
1️⃣ Business Method (runs first)  
2️⃣ Advice Method (runs second, but only if successful)

**Use Case Example:**  
You want to log a success message **only when** a transaction is completed without errors.

**Analogy:**  
Imagine you pay your bill online.  
If the payment succeeds, you get a *“Payment Successful”* message (that’s your After Returning Advice).  
If payment fails, that message will never appear.

**Example Code:**

@AfterReturning("execution(\* com.example.service.PaymentService.makePayment(..))")

public void logSuccess() {

System.out.println("✅ Payment completed successfully!");

}

So, this advice **only runs** when makePayment() runs successfully without any exception.

**e) After Throwing Advice (@AfterThrowing)**

**When it runs:**  
After the business method **throws an exception** (fails).

**Execution order:**  
1️⃣ Business Method (runs first, throws exception)  
2️⃣ Advice Method (runs next, only because of that exception)

**Use Case Example:**  
To log errors or send alerts when something goes wrong in your method.

**Analogy:**  
Imagine an ATM machine.  
If the transaction fails (for example, “insufficient balance”), you immediately get an error message on the screen — that message is like an *After Throwing Advice.*

**Example Code:**

@AfterThrowing("execution(\* com.example.service.PaymentService.makePayment(..))")

public void logError() {

System.out.println("❌ Payment failed due to an error!");

}

So, this advice **only runs** when makePayment() throws an exception.

| **Annotation** | **Advice Type** | **When It Runs** | **Works When (Success / Failure)** | **Typical Use Case** | **Example / Analogy** |
| --- | --- | --- | --- | --- | --- |
| **@Before** | **Before Advice** | Runs **before** the business method starts | ✅ Works on both success and failure (because it runs *before* method execution) | Checking authentication, input validation, logging entry | Security guard checks your ID **before** letting you enter |
| **@After** | **After (Finally) Advice** | Runs **after** the business method finishes (no matter what) | ✅ Runs on **success** and ❌ **failure** (always executes like finally block) | Cleanup tasks, closing DB connection, releasing resources | Like saying “Goodbye” after an event — no matter it was successful or not |
| **@AfterReturning** | **After Returning Advice** | Runs **after** the business method executes **successfully** | ✅ Runs **only on success** ❌ Does not run on exception | Log “operation successful”, send confirmation message | Display “Payment Successful” message after a valid transaction |
| **@AfterThrowing** | **After Throwing Advice** | Runs **after** the business method **throws an exception** | ❌ Does not run on success ✅ Runs **only on failure** | Logging errors, sending failure alerts | Showing “Payment Failed” message when transaction fails |
| **@Around** | **Around Advice** | Runs **before and after** the business method | ✅ Runs on both success and failure (since it surrounds the method execution) | Measure execution time, start & end transactions | Start stopwatch before a race, stop it after — even if runner trips |
| **@Pointcut** | **PointCut Expression** | Used to **select methods** for advices (does not execute itself) | 🚫 Not applicable — it only defines where advices should apply | Define reusable method selection rules | Like marking which classes/methods need special attention |
| **@Aspect** | **Aspect Declaration** | Marks a class as an Aspect (container of advices) | 🚫 Not applicable — it’s a class-level annotation | Define cross-cutting concern (LoggingAspect, SecurityAspect) | Like declaring “This class will handle all security checks” |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

### 🧩 ****@Pointcut — Selecting Where Advice Should Work****

* @Pointcut is **like a filter** that tells **where** (on which methods) your advice should run.
* It doesn’t execute anything by itself. It only **marks or identifies** the methods that need extra behavior (like logging, security, etc.).
* You can think of it as **a rule or condition** to select specific business methods.

**Example:**

@Pointcut("execution(\* com.orcas.service.PaymentService.\*(..))")

public void paymentMethods() {}

Here, we are saying — “Apply my advice on **all methods inside PaymentService class**.”

So if we later write:

@Before("paymentMethods()")

public void checkSecurity() {

System.out.println("Security check before payment");

}

Then this advice will automatically run before all methods of PaymentService.

🧠 **Analogy:**  
Imagine you’re a teacher checking homework.  
You say — “I’ll check **only math notebooks**.”  
That rule (“only math notebooks”) is your **@Pointcut**.

**🧠 Complete Example — Understanding @Pointcut + @Before**

We’ll create **two classes**:

1. 🏦 PaymentService → our **Business (Target) Class**
2. 🛡️ SecurityAspect → our **Aspect Class** (contains Advice + Pointcut)

**🏦 Business (Target) Class**

📄 **File:** PaymentService.java  
📦 **Package:** com.orcas.service

package com.orcas.service;

import org.springframework.stereotype.Service;

@Service

public class PaymentService {

public void makePayment() {

System.out.println("💰 Payment processing started...");

}

public void refundPayment() {

System.out.println("💸 Refund processing started...");

}

}

**Explanation:**

* This is our **Target Class**.
* It contains normal business logic (payment, refund, etc.).
* No AOP logic is written here.
* These are the methods we want to **secure** or **add extra behavior** to.

**🛡️ Aspect Class**

📄 **File:** SecurityAspect.java  
📦 **Package:** com.orcas.aspect

package com.orcas.aspect;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Aspect // Tells Spring this class contains advices

@Component // So Spring can detect it during component scan

public class SecurityAspect {

// 🧩 Step 1: Define PointCut (select which methods need extra behavior)

@Pointcut("execution(\* com.orcas.service.PaymentService.\*(..))")

public void paymentMethods() {

// This is an empty method — used only as a reference for PointCut

}

// ⚙️ Step 2: Define Advice (actual extra logic)

@Before("paymentMethods()")

public void checkSecurity() {

System.out.println("🔒 Security check before executing payment method...");

}

}

**Explanation (Line by Line):**

1. @Aspect → Marks this class as an **Aspect**.  
   Spring knows this class contains **AOP logic**.
2. @Component → So Spring can **create and manage** this class as a bean automatically.
3. @Pointcut("execution(\* com.orcas.service.PaymentService.\*(..))")
   * Means: Apply this PointCut to **all methods** inside PaymentService class.
   * It matches:
     + All methods (\*)
     + Inside package com.orcas.service
     + Inside class PaymentService
     + With any parameters ((..))
4. The empty method paymentMethods() is just a **marker name** for the PointCut.  
   We’ll use this name inside other annotations.
5. @Before("paymentMethods()")
   * Tells Spring to run checkSecurity() **before** any method in PaymentService runs.
   * This is called a **Before Advice**.
6. checkSecurity()
   * Actual implementation of Advice.
   * Prints a message **before** the payment or refund methods start.

**🧩 What Happens When You Run It**

If you call:

paymentService.makePayment();

**Output:**

🔒 Security check before executing payment method...

💰 Payment processing started...

If you call:

paymentService.refundPayment();

**Output:**

🔒 Security check before executing payment method...

💸 Refund processing started...

**🪄 Behind the Scenes (What Spring Does Internally)**

1. Spring creates a **Proxy object** for PaymentService.  
   This proxy adds the security logic from SecurityAspect.
2. When you call makePayment() or refundPayment(),  
   the call first goes to the **proxy**, not the original method directly.
3. The proxy runs:
   * the @Before advice (checkSecurity())
   * then the actual method (makePayment())

**🧠 In Simple Words:**

* PaymentService → your **original worker** who performs payment.
* SecurityAspect → your **security guard** who checks before the worker starts.
* @Pointcut → tells the guard **which workers to watch**.
* @Before → tells **when** to check (before starting work).