**AOP in Spring Boot – 2025**

**What is CrossCutting Concern ?**

**A concern is a part of the functionally divided system**.

There are two types of concerns:

* **Core Concern**: concerns representing **single and specific functionality for primary requirements. Example: Indexing medical records for search functionality, Building an Apartment or House. Mostly Business Logic.**
* **Cross Cutting Concern**: The **concerns representing functionalities for secondary requirements.** It is **applicable throughout the application** and it affects the entire application. For example: **logging**, **security**, **auditing, Transaction management.**

**What is Separation of concerns (SoC)**

Separation of Concern means segregation of different sections of the functionalities. Separation of Concern (SoC) is a design principle which states that a software system should be divided into distinct sections, each addressing a separate concern or aspect of the system. A classic example of SoC is the Model-View-Controller (MVC) pattern used in web applications:

**What is Aspect Oriented Programming?**

**Aspect-Oriented Programming (AOP)** is a programming paradigm that aims to **separate cross-cutting concerns** from the main business logic of an application. **AOP** is a technique that allows you to **modularize concerns** that affect multiple parts of an application (called **cross-cutting concerns**) such as **Logging, Security, Transaction management, Performance monitoring**. It does so by adding behavior to existing code (an advice) without modifying the code itself.

**Why Use AOP?**

* **Improves modularity**: Keeps business logic clean and focused.
* **Reduces code duplication**: Centralizes common behavior.
* **Enhances maintainability**: Easier to update cross-cutting logic in one place.

**Aspect**

**An aspect is a common feature or a concern spread across the methods,** classes, object hierarchies. Technically a class annotated with **@Aspect** and contains **advice** (the action to take) and **pointcuts** (where to apply the advice).

**Join point**

This represents **a point in your application where you can plug-in the AOP aspect**. A point in the execution of the program (e.g., method call) where an aspect can be applied.

**Advice:** The action taken by an aspect at a join point (e.g., log before method execution).

**Pointcut:** This is a set of one or more join points where an advice should be executed. A **Pointcut** is a **set of criteria** used to determine **which methods** (or join points) should be intercepted by an aspect's advice.

**Target object:** The object being advised by one or more aspects.

**Weaving:** Weaving is the process of linking aspects with other application types or objects to create an advised object.

**AOP can also be achieved using Plain Core Java**

1. **Using Decorator Pattern**
2. **Using Dynamic Proxy (InvocationHandler)**
   1. It always requires an Interface to be present, Proxy.newInstance requires one interface. Without interface, you have to take the help of CGlib library.

**Decorator Pattern**: **Attach additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.**

**public interface Service {**

**void execute();**

**}**

**public class RealService implements Service {**

**@Override**

**public void execute() {**

**System.*out*.println("Executing real service...");**

**}**

**}**

**public class LoggingDecorator implements Service {**

**private final Service decoratedService;**

**public LoggingDecorator(Service service) {**

**this.decoratedService = service;**

**}**

**@Override**

**public void execute() {**

**System.*out*.println("Logging before execution");**

**decoratedService.execute();**

**System.*out*.println("Logging after execution");**

**}**

**}**

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Service realService = **new** RealService();

Service loggingService = **new** LoggingDecorator(realService);

loggingService.execute();

}

}

**Using InvocationHandler**

**public** **interface** Calculator {

**int** calculate(**int** a, **int** b);

}

**public** **class** RealCalculator **implements** Calculator {

@Override

**public** **int** calculate(**int** a, **int** b) {

**return** a\*b;

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Calculator calculator = **new** RealCalculator();

InvocationHandler handler = ((proxy, method, arguments) -> {

System.***out***.println("Doing something before calculation ...");

Object obj = method.invoke(calculator, arguments);

System.***out***.println("Invoking ...");

System.***out***.println("Doing something after calculation ...");

**return** obj;

});

Calculator calc = (Calculator) Proxy.*newProxyInstance*(calculator.class.getClassLoader(),

calculator.getClass().getInterfaces(), handler);

**int** val = calc.calculate(10,5);

System.***out***.println(val);

}

}

**The above code can be written as**

Calculator realCalculator = **new** RealCalculator();

Calculator proxyCalculator =

(Calculator) Proxy.*newProxyInstance*(

Calculator.**class**.getClassLoader(), **new** Class<?>[]{Calculator.**class**},

(**proxy, method, args1**) -> {

System.***out***.println("Before calculation...");

Object result = **method.invoke**(realCalculator, **args1**);

System.***out***.println("After calculation...");

**return** result;

}

);

**int** result = proxyCalculator.calculate(10, 5);

System.***out***.println("Result: " + result);

**What is the difference between JDK Dynamic Proxy and CGLIB Proxy?**

**DK Dynamic Proxy**: **Used when the target class implements an interface**. It creates a proxy that implements the same interface as the target class.

**CGLIB Proxy**: **Used when the target class does not implement any interfaces**. It creates a subclass of the target class and intercepts method calls on the subclass.

**Spring pom.xml Dependency**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-aop</artifactId>**

**</dependency>**

**Spring AOP**

1. **@Before** : Runs before the method execution.
2. **@After** : Runs after the method execution (regardless of outcome).
3. **@Around** : Runs before and after the method execution.
4. **@AfterReturning** : Runs after the method returns a result.
5. **@AfterThrowing** : Runs if the method throws an exception.

**@Before(value = “”, argNames = “”)**

**@Before(value = "execution(\* com.ddlab.rnd.service.impl.CheckServiceImpl.createPerson(..))")**

**public void beforeAllMethods(JoinPoint joinPoint)**

**@After(value = “”, argNames = “”)**

**@After("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")**

**public void afterExecutingAllMethods(JoinPoint joinPoint)**

**@Around(value = “”, argNames = “”)**

**@Around("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")**

**public Object aroundMethod(ProceedingJoinPoint pjp) throws Throwable { }**

**@AfterReturning(value = “”, argNames = “”, pointcut = “”, returning = “”)**

**@AfterReturning("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")**

**public void afterReturningAllMethods(JoinPoint joinPoint)**

**or**

**You can not use value with returning, you have to use pointcut and returning**

**@AfterReturning(pointcut = "execution(\* com.ddlab.rnd.service.CrudService.\*(..))",**

**returning = "returnObj")**

**public void afterReturningAllMethods(JoinPoint joinPoint, Object returnObj) {}**

**@AfterThrowing(value = “”,argNames = “”,pointcut = “”, throwing = “”)**

**@AfterThrowing(pointcut = "execution(\* com.ddlab.rnd.service.CrudService.\*(..))",**

**throwing = "ex1")**

**public void afterThrowingAllMethods(JoinPoint joinPoint, Exception ex1) throws Throwable {}**

**Example Service Implementation Class is given below.**

@Service

**public** **class** CrudService {

**public** Employee **m1**(Employee e, String param1, **int** param2) {

System.***out***.println("---------Service Method BEGIN----------");

System.***out***.println("Employee Details : " + e);

System.***out***.println("---------Service Method END----------");

**return** e;

}

}

**Basic AOP Class given below.**

**@Aspect**

**@Component**

**public** **class** AOPAspectComponent {

@Before("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")

@After("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")

@AfterReturning("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")

@AfterThrowing("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")

**public** **void** hookup(JoinPoint joinPoint) {

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ALL ANNOTATIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

**Signature signature = joinPoint.getSignature();** // org.aspectj.lang.Signature

System.***out***.println("Method Name : " + signature.getName()); // m1

Object[] args = **joinPoint.getArgs();**

**for** (Object obj : args) {

System.***out***.println("Method Arguments : " + obj);// Employee(id=123, name=John), Param-1

}

**Class returnType = ((MethodSignature) signature).getReturnType();**

System.***out***.println("Method Return Type: "+returnType); //class com.ddlab.rnd.entity.Employee

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ALL ANNOTATIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

**@AfterReturning**(pointcut="execution(\* com.ddlab.rnd.service.CrudService.\*(..))",returning = "emp")

**public** **void** afterReturning(JoinPoint joinPoint, **Employee emp**) {

System.***out***.println("What did it return: "+emp); //Employee(id=1234, name=John Abraham)

}

**}**

**In the service layer, method is throwing exception, you can hook like the following ways.**

**public** Employee m1(Employee e, String param1, **int** param2) {

System.***out***.println("---------Service Method BEGIN----------");

System.***out***.println("Employee Details : " + e);

**if**(param1 == **null**) **throw** **new** NullPointerException("Param1 cannot be null");

System.***out***.println("---------Service Method END----------");

Employee newEmp = **new** Employee(1234, "John Abraham");

**return** newEmp;

}

**Corresponding AOP method**

**@AfterThrowing**("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")

**public** **void** afterThrowing(JoinPoint joinPoint) {

System.***out***.println("Throwing exception ..."); 🡸 This will be printed

}

**Another variation of the above method.**

**@AfterThrowing**(value = "execution(\* com.ddlab.rnd.service.CrudService.\*(..))", throwing = "ex")

**public** **void** afterThrowing(JoinPoint joinPoint, **Exception ex**) {

System.***out***.println("Throwing exception ..."+ex.getMessage());

}

**In the above case, it is necessary to have argument as Exception and mention as throwing = “ex”**

**Usage of @Around annotation**

**@Around("execution(\* com.ddlab.rnd.service.CrudService.\*(..))")**

**public** Object performAroundAllMethods(**ProceedingJoinPoint pjp**) **throws** Throwable {

Object returnedObject = **null**;

System.***out***.println("---------- Before Execution in Around -------------");

**try** {

System.***out***.println("Method Name : " + pjp.getSignature().getName());

**Object[] args = pjp.getArgs();**

**for** (Object obj : args) {

System.***out***.println("Argument : " + obj);

}

**Signature signature = pjp.getSignature();**

**Class returnType = ((MethodSignature) signature).getReturnType();**

System.***out***.println("Return Type : "+returnType);

Object targetObject = pjp.getTarget();

System.***out***.println("Target Object : "+targetObject);

**returnedObject = pjp.proceed();** // You have to return the object

} **finally** {

System.***out***.println("Performed other operations in finally ...");

}

System.***out***.println("---------- After Execution in Around -------------");

**return returnedObject;// If you don't return, calling class will get null value**

}

}

**How to hook a particular method**

**@Configuration**

**@Aspect**

**public** **class** AOPInterceptors {

**@Before(value = "execution(\* com.ddlab.rnd.service.impl.CheckServiceImpl.createPerson(..))")**

**public** **void** checkBefore(JoinPoint jp) {

System.***out***.println(".............. Before Execution ................ ");

System.***out***.println("Method Name : " + jp.getSignature().getName());

Object[] args = jp.getArgs();

**for** (Object obj : args) {

System.***out***.println("Argument : " + obj);

}

}

}

**Note: You have to use ProceedingJoinPoint only in case of @Around annotation otherwise it will throw,** **Caused by: java.lang.IllegalArgumentException: ProceedingJoinPoint is only supported for around advice**

Test Class is given below.

**public** **class** TestBefore {

**public** **static** **void** check(ApplicationContext applicationContext) {

CrudService service = applicationContext.getBean(CrudService.**class**);

Person p = **new** Person();

p.setFirstName("John");

p.setLastName("Abraham");

p.setAge(23);

p.setId(111);

Person newPerson = service.createUpdateObtain(p, "vidya", 33);

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.***out***.println("In Test Finally newPerson : " + newPerson);

service.performSlowOperation();

}

}

**How to create your own custom AOP annotation**

Let us create a custom annotation called **@TrackExecutionTime.** The code is given below.

@Target(ElementType.***METHOD***)

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** TrackExecutionTime { 🡸 Create this annotation

}

@Aspect

@Component

**public** **class** TrackTimeAspect {

@Around("@annotation(TrackExecutionTime)")

**public** Object logExecutionTime(ProceedingJoinPoint joinPoint) **throws** Throwable {

**Instant start = Instant.now();**

Object proceed = joinPoint.proceed();

**Instant end = Instant.now();**

**long duration = Duration.*between*(start, end).toMinutes();**

System.***out***.println(joinPoint.getSignature() + " executed in minutes:" + duration);

**return** proceed;

}

}

Use it in the following method.

@TrackExecutionTime

**public** **void** performSlowOperation() {

System.***out***.println("Operation Started ....");

**try** {

TimeUnit.***SECONDS***.sleep(5);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Operation Completed ....");

}

Spring Boot Main Class is given below.

@SpringBootApplication

**public** **class** MainApplication {

**public** **static** **void** main(String[] args) {

ApplicationContext applicationContext = SpringApplication.*run*(MainApplication.**class**, args);

TestBefore.*check*(applicationContext);

}

}

**Usage of PointCut (PointCut Designators) PCD**

* **execution**() - match a joinPoint method’s signature
* **within**() - match all the JoinPoint methods in a given class, package, or sub-package

Other existing types of pointcut expressions: **args()**, **target()**, **this(),** **@args()**, **@within()**, **@target()**, **@annotation()**

**How is @Pointcut different from @Before, @After etc**

In simple words whatever you specify inside @Before or @After is a pointcut expression. This can be extracted out into a separate method using @Pointcut annotation for better understanding, modularity and better control.

Example is given below.

@Pointcut("target(com.ddlab.rnd.service.CrudService)")

**public** **void** hookAllMethodsWithinAClass() {

System.***out***.println("------------ Inside Pointcut -----------------");

}

@Before("hookAllMethodsWithinAClass()")

**public** **void** callPointCut(JoinPoint joinPoint) {

System.***out***.println(".............. Before Execution ................ ");

System.***out***.println("Method Name : " + joinPoint.getSignature().getName());

Object[] args = joinPoint.getArgs();

**for** (Object obj : args) {

System.***out***.println("Argument : " + obj);

}

}

Pointcut expressions can be combined using **&&**, **||** and **!** operators:

@Pointcut("@target(org.springframework.stereotype.Repository)")

public void repositoryMethods() {}

@Pointcut("execution(\* \*..create\*(Long,..))")

public void firstLongParamMethods() {}

@Pointcut("repositoryMethods() && firstLongParamMethods()")

public void entityCreationMethods() {}

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)

throws-pattern?)

Some examples of common pointcut expressions are given below.

* the execution of any public method:

execution(public \* \*(..))

* the execution of any method with a name beginning with "set":

execution(\* set\*(..))

* the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

* the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

* the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

*'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.*

* any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

*'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

*'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.*

Note that the pointcut given in this example is different to execution(\* \*(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

* any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

*'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

@within(org.springframework.transaction.annotation.Transactional)

*'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

*'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

@args(com.xyz.security.Classified)

*'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.*

* any join point (method execution only in Spring AOP) on a Spring bean named 'tradeService':

bean(tradeService)

* any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression '\*Service':

bean(\*Service)

A pointcut expression can appear as a value of the @Pointcut annotation:

@Pointcut("within(@org.springframework.stereotype.Repository \*)")

**public** **void** **repositoryClassMethods**() {}

The method declaration is called the **pointcut signature**. It provides a name that advice annotations can use to refer to that pointcut:

@Around("repositoryClassMethods()")

**public** Object **measureMethodExecutionTime**(ProceedingJoinPoint pjp) **throws** Throwable {

...

}