**AOP in Spring Boot – 2022**

**What is CrossCutting Concern ?**

Before understanding the Crosscutting Concern, we have to understand the Concern.

**A Concern is a part of the system divided on the basis of the functionality**. **A concern is a part of the divided functionality**. **A concern is a part of the functionally divided system**.

There are two types of concerns:

* The concerns representing **single and specific functionality for primary requirements** are known as **core concerns or Primary functionality** of the system is known as core concerns. For example: Business logic. I have to build an apartment/buy a flat that is core concern.
* The **concerns representing functionalities for secondary requirements** are referred to as crosscutting concerns or system-wide concerns. OR The **crosscutting concern is a concern which is applicable throughout the application** and it affects the entire application. For example: logging, security, auditing which are needed in almost every module of an application, hence they are cross-cutting concerns. Security for the flat, car parking is a concern applicable to entire apartment.

If writing an application for handling medical records, **the indexing of such records is a core concern**, while logging a history of changes to the record database or user database, or an authentication system, would be cross-cutting concerns since they interact with more parts of the program.

In computer science, a concern is a particular set of information that has an effect on the code of a computer program.

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

In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections. Each section addresses a separate concern, a set of information that affects the code of a computer program.

**What is Aspect Oriented Programming?**

In computing, aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding behavior to existing code (an advice) without modifying the code itself.

**Aspect**

**An aspect is a common feature or a concern that's typically scattered across methods**, classes, object hierarchies. **A modularization of a concern that cuts across multiple classes**. Transaction management is a good example of a crosscutting concern in enterprise Java applications.

**Join point**

This represents **a point in your application where you can plug-in the AOP aspect**. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework. A method in a class can be a join point.

**Advice:** This is the **actual action to be taken** either before or after the method execution.

**Pointcut:** This is a set of one or more join points where an advice should be executed.

**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** Account {  
 **void** withdraw(**int** amount);  
}

**public class** RetailAccount **implements** Account {  
 @Override  
 **public void** withdraw(**int** amount) {  
 System.***out***.println(**"Retail customer drawing "**+amount);  
 }  
}

**public class** AccountDecorator **implements** Account {  
 **private** Account **account**;  
 **public** AccountDecorator(Account account) {  
 **this**.**account** = account;  
 }  
 @Override  
 **public void** withdraw(**int** amount) {  
 System.***out***.println(**"Retail customer check ..."**);  
 **account**.withdraw(amount);  
 }  
}

**public class** TestDecorator {  
 **public static void** main(String[] args) {  
 Account account = **new** RetailAccount();  
 AccountDecorator decorator = **new** AccountDecorator(account);  
 decorator.withdraw(1000);  
 }  
}

**Using InvocationHandler**

**public class** AccountInvocationHandler **implements** InvocationHandler {  
 **private** Account **account**;  
  
 **public** AccountInvocationHandler(Account account) {  
 **this**.**account** = account;  
 }  
  
 @Override  
 **public** Object invoke(Object **proxy**, Method **method**, Object[] **args**) **throws** Throwable {  
 System.***out***.println(**"Executing before ..."**);  
 **Object result = method.invoke(account, args);** System.***out***.println(**"Executing after ..."**);  
 **return** result;  
 }  
}

**public class** TestInvocationHandler {  
  
 **public void** testUsingLambda() {  
 Account account = **new** RetailAccount();  
 **InvocationHandler aroundHandler = (proxy, method, arguments) ->** {  
 System.***out***.println(**"Do something before"**);  
 **Object obj = method.invoke(account, arguments);**  
 System.***out***.println(**"Do something after"**);  
 **return** obj;  
 };  
  
 Account act = (Account) **Proxy.*newProxyInstance*(account.getClass().getClassLoader(),  
 account.getClass().getInterfaces(), aroundHandler);** act.withdraw(1000);  
 }  
  
 **public void** usingNormalWay() {  
 Account account = **new** RetailAccount();  
 **AccountInvocationHandler aroundHandler = new AccountInvocationHandler(account);**  
 **Account act = (Account) Proxy.*newProxyInstance*(account.getClass().getClassLoader(),  
 account.getClass().getInterfaces(), aroundHandler);** act.withdraw(1000);  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestInvocationHandler().testUsingLambda();  
 }  
}

**Spring AOP**

1. **@Before** : Advice executes before a join point
2. **@AfterReturning** : Advice executes after a join point completes normally.
3. **@AfterThrowing** : Advice to be executed if a method exits by throwing an exception.
4. **@After** : Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
5. **@Around** : Advice that surrounds a join point such as a method invocation. It executes before and after a join point.

Spring pom.xml

**<dependency>**

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

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

**</dependency>**

**Service Layer**

@Service

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

**public** **void** create(Person p) {

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

System.***out***.println("Person Details : " + p);

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

}

**public** Person createUpdateObtain(Person p, String param1, **int** param2) {

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

System.***out***.println("Person Details : " + p);

Person p1 = **new** Person();

p1.setFirstName(param1);

p1.setLastName(p.getLastName());

p1.setAge(param2);

p1.setId(param2);

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

**return** p1;

}

**public** Person methodThrowsException(Person p, String param1, **int** param2) {

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

System.***out***.println("Person Details : " + p);

Person p1 = **new** Person();

p1.setLastName(p.getLastName());

p1.setFirstName(param1.substring(8, 13)); // Throws Exception

p1.setAge(param2);

p1.setId(param2);

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

**return** p1;

}

@TrackExecutionTime

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

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

**try** {

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

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

All AOP implementation is given below.

**@Aspect**

**@Configuration**

**public** **class** AOPExecutor {

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

**public** **void** performBeforeAllMethods(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);

}

}

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

**public** **void** performAfterReturningAllMethods(JoinPoint joinPoint) **throws** Throwable {

System.***out***.println(".............. After Returning Execution ................ ");

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

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

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

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

}

Object targetObject = joinPoint.getTarget();

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

}

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

**returning = "returnObj")**

**public** **void** performAfterReturningAllMethods(JoinPoint joinPoint, Object returnObj) {

System.***out***.println(".............. After Returning Execution ................ ");

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

Object[] args = joinPoint.getArgs();

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

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

}

Object targetObject = joinPoint.getTarget();

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

System.***out***.println("Return Object : "+returnObj);

}

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

**throwing = "ex1")**

**public** **void** performAfterThrowingAllMethods(JoinPoint joinPoint, Exception ex1)

**throws** Throwable {

System.***out***.println("........... After Throwing Exception ............. ");

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

Object[] args = joinPoint.getArgs();

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

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

}

Object targetObject = joinPoint.getTarget();

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

System.***out***.println("Exception Details : " + ex1);

}

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

**public** **void** performAfterExecutingAllMethods(JoinPoint joinPoint) **throws** Throwable {

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

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

Object[] args = joinPoint.getArgs();

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

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

}

Object targetObject = joinPoint.getTarget();

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

}

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

Create an annotation

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

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

**public** **@interface** TrackExecutionTime {

}

@Aspect

@Component

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

@Around("@annotation(TrackExecutionTime)")

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

**long** start = System.*currentTimeMillis*();

Object proceed = joinPoint.proceed();

**long** executionTime = System.*currentTimeMillis*() - start;

System.***out***.println(joinPoint.getSignature() + " executed in " + executionTime + "ms");

**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**

Example is given below.

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

// @Pointcut("within(com.ddlab.rnd.service..\*)")

@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)