**Spring AOP**

Spring AOP, or Aspect-Oriented Programming, is a key feature of the Spring Framework that allows developers to modularize cross-cutting concerns in their applications. Cross-cutting concerns are aspects of your application that affect multiple modules but are not part of the core business logic. Examples include logging, security, and transaction management.

**AOP Project contains two parts.**

**Business Class:**

Business class is a component of your application that contains the core business logic and functionality. It encapsulates the essential features and operations that are directly related to the primary purpose of the application.

Characteristics:

* It focuses on the primary concerns of the application, such as specific functionality or business rules.
* It implements the core features of the application, typically following the requirements of the domain.

**Aspect Class:**

Class where the transaction management,security concept,logging,Encode/Decode takes place

Characteristics:

* It addresses concerns that are common across different parts of the application.
* It typically contains advice, which is the additional behavior that needs to be executed at specific join points in the application.

**Important Terminologies:**

1)Aspect- Class where the transaction management,security concept,logging,Encode/Decode takes place

2)Advice - It is a method inside Aspect class. (Actual implementation of Aspect)

3)Pointcut - It is an expression where it will select b. Method which needs advices.

exp: @Pointcut ("execution (public \* com.app.ProductDao.\*(..))")

4)JoinPoint - Linking the business method with suitable advices (Advices+Pointcuts)

5)Target - Pure Business Class object

6)Weaving - Mixing of b.class methods and their connected advices.

7)Proxy -Final Output (class/object) is called as Proxy that contains both logic connected

**Types of Advice**

* Before Advice(@Before): Executing advice before calling b.method
* After Advice(@After): Executing advice after b.method finished
* Around Advice (@Around): Advice code made into 2 sections/parts,1st part executed before b.method, later 2nd part after b.method.
* After Returning Advice(@AfterReturning): After executing b.method,only on success execute advice.
* After Throwing Advice (@AfterThrowing): After executing b.method, if b.methodis throwing any exception, then execute advice

Demo Project was done to illustrate the AOP.

**Dependencies used:**

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

**Tx Service Class**

Firstly, we should map the business class method to a method with the pointcut expression. After that we can implement the advices like Before,After,Around,AfterReturning,AfterThrowing.

@Before("p1()") - Before the execution of the saveEmployee of EmpDao class, the method annotated with @Before should be executed.

@After("p1()") – After the execution of the saveEmployee of EmpDao class, the method annotated with @After should be executed.

The @Before and @After method will be executed at any cost.

@AfterReturning("p1()") – Only after the execution of the saveEmployee of EmpDao class, the method annotated with @AfterReturning should be executed. If any exception or error is thrown by the saveEmployee method, the advice method will not be executed.

@AfterThrowing("p1()") – After throwing the exception of the saveEmployee of EmpDao class, the method annotated with @AfterThrowing will be executed.

@Around – This advice will have two parts. The first part will be carried out before the implementation of the business class method. The second part will be executed after the implementation of the business class method. jp.proceed() method from ProceedingJoinPoint is responsible for calling business call method.

@Aspect

@Component

public class TxService {

//pointcut

@Pointcut ("execution (public void com.demo.spring.dao.EmployeeDao.saveEmployee())")

public void p1() {}

@Pointcut ("execution (\* com.demo.spring.dao.EmployeeDao.\*())")

public void p2() {}

//before

@Before("p1()”) //joinpoint

public void beginTx ()

{

System.out.println("Tx started...");

}

//after

@After("p1()")

public void sendReport()

{

System.out.println("Report sent!");

}

//afterreturning - only after success of business class

@AfterReturning("p1()")

public void commitTx()

{

System.out.println("Tx is committed");

}

@AfterThrowing(value= "p1()",throwing="th")

public void rollbackTx(Throwable th)

{

System.out.println("Tx is rollback "+th.getMessage());

}

@Around("p1()")

public void aroundTest(ProceedingJoinPoint jp)

{

System.out.println("Before b.method");

//code -->call: b.method

try

{

// Object ob=jp.proceed(); If it is non void method

jp.proceed();

} catch(Throwable e)

{

e.printStackTrace();

}

//at any case

System.out.println("After b.method");

}

//Mapping through the annotation

@Pointcut ("@annotation (com. demo.spring.anno.MyTx)")

public void p3() {}

@Before("p3()")

public void beginTx2()

{

System.out.println("Tx begin from annotation");

}

**Annotation based AOP**

* The @Retention will allow the annotation method which will be called through the joinpoint in aspect. With the help of @Target, we are specifying we are annotating a [method. @interface](mailto:method.@interafce) is the interface with name @MyTx

@Retention (RetentionPolicy.RUNTIME)

@Target (ElementType.METHOD)

public @interface MyTx {}

* We are implementing the method of the interface MyTx ie saveEmployee2()

@MyTx

public void saveEmployee2() {

System.out.println("From save Employee by annotation");

}

* Call the saveEmployee2 method in the TestRunner class.

@Override

public void run (String... args) throws Exception

{ dao.saveEmployee();

dao. saveEmployee2();

}

* The below line indicates, select the method of the @MyTx interface which is available in the com.demo.spring.anno package.

@Pointcut ("@annotation (com. demo.spring.anno.MyTx)")

public void p3() {}

Before the execution of annotation method this method should be executed.

@Before("p3()")

public void beginTx2()

{

System.out.println("Tx begin from annotation");

}

The below line indicates the selecting method can be any access specifier, any return type with any parameters.

@Pointcut (value = "execution (\* com.demo.spring.controller.EmpController.getAllEmployees(..))")

public void myPointcut() {

}

We are implementing the Around Advice in the below code. Object mapper is used to convert the java objects to JSON. Get the methodname and classname and store it in methodName and className variables. The Object array () will get the arguments of the method but the there are no arguments we are passing in the business method.

The first part will print the logs containing className, methodName,arguments and again the second part print the logs containing className, methodName along with the object as String.(List of employees as we are fetching all the employees details in it)

@Around ("myPointcut ()")

public Object applicationLogger(ProceedingJoinPoint pjp) throws Throwable {

ObjectMapper mapper = new ObjectMapper(); //convert objects to JSON

String methodName = pjp.getSignature().getName();

String className = pjp.getTarget().getClass().toString();

Object [] array = pjp.getArgs();

log.info ("method invoked " + className + “: " + methodName + "()" + "arguments : "

+ mapper.writeValueAsString(array));

Object object = pjp.proceed();

log.info (className + “: " + methodName + "()" + "Response : " + mapper.writeValueAsString(object));

return object.

}