**AOP**

In Spring Boot, Aspect-Oriented Programming (AOP) is a powerful mechanism for separating cross-cutting concerns, allowing you to modularize concerns such as logging, security, and transaction management.

**Terminologies:**

*controller*

**Withdraw(val)**

**Deposit(val)**

**Bal\_enq()**

*aspect*

**Before-Bal\_check()**

**After-disp\_bal()**

**Aspect**: An aspect is a module that encapsulates a cross-cutting concern. It contains advice and join points. In Spring AOP, aspects are implemented using regular Java classes annotated with @Aspect.

**Advice**: Advice is the action taken by an aspect at a particular join point. Types of advice include @Before, @After, @AfterReturning, @AfterThrowing, and @Around. Advice methods are annotated with these annotations to specify when they should be executed.

**Join Point**: A join point is a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, join points are method executions.

**Pointcut**: A pointcut is a set of one or more join points where advice should be executed. It defines the criteria for selecting join points in the code. Pointcut expressions are used to specify these criteria.

**Target Object**: The object being advised by one or more aspects. In Spring AOP, the target object is the object containing the business logic.

Steps:

1. Create a new AspectConfig.java file
2. Provide the below code to include

package com.example.onetomanyuniauthor.config;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.*After*;

import org.aspectj.lang.annotation.*AfterReturning*;

import org.aspectj.lang.annotation.*Around*;

import org.aspectj.lang.annotation.*Aspect*;

import org.aspectj.lang.annotation.*Before*;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.annotation.*Configuration*;

@*Configuration*  //@Configuration annotation which indicates that the class has @Bean definition methods.

@*Aspect*    //Contains classes like JoinPoint, Point cut, @Before, @After etc..

public class AspectConfig {

    Logger logger=LoggerFactory.getLogger(getClass());

    // Executes before the method

 @*Before*(value="execution(\* com.example.onetomanyuniauthor.controller.\*.\*(..)) ")// pointcut

public *void* beforeAdvice (JoinPoint *joinpoint*)

{   logger.info("Inside Befroe Advice");

}

//Executes after the method

@*After*(value="execution(\* com.example.onetomanyuniauthor.controller.\*.\*(..)) ")

public *void* afterAdvice (JoinPoint *joinpoint*)

{   logger.info("Inside After Advice");

}

//Executes before the method along with the parameter

 @*Before*(value="execution(\* com.example.onetomanyuniauthor.controller.\*.\*(..)) and args(object1,object2) ")

public *void* beforeAdviceWith1Param (JoinPoint *joinpoint*, long *object1,* Object *object2*)

{   logger.info("Inside Befroe Advice with parameter");

}

//Executes afterReturning the value to called the method(which performs prior to After method)

@*AfterReturning*(value="execution(\*com.example.onetomanyuniauthor.controller.AuthorController.getAuthors(..)) ")

public *void* afterReturningAdvice (JoinPoint *joinpoint*)

{   logger.info("Inside AfterReturning Advice");

}

//Executes a method by allowing the advice to control the method execution. It is more powerful than other types of advice because it has the ability to control both the input and output of the target method.

@*Around* (value="execution(\* com.example.onetomanyuniauthor.controller.\*.\*(..)) ")

public Object  newAround(ProceedingJoinPoint *joinpoint*) throws Throwable

{   System.out.println("Before Executing Method: "+*joinpoint*.getSignature().toString());

    Object result=*joinpoint*.proceed();

    System.out.println("After Executing Method: "+*joinpoint*.getSignature().toString());

    return result;

}

}