

How to Implement AOP in Spring Boot Application?

Last Updated : 23 Jul, 2025

AOP(Aspect Oriented Programming) breaks the full program into different smaller units. In numerous situations, we need to log, and audit the details as well as need to pay importance to declarative transactions, security, caching, etc., Let us see the key terminologies of AOP

1. **Aspect:** It has a set of APIs for cross-cutting requirements. Logging module is an example of the AOP aspect of logging.
2. **Joint Point:** AOP aspect plug in place
3. **Advice:** Via this, the actual implementation of code is taken care for the AOP approach. It can be either before/after/returning/after throwing. In this article let us see the examples related to this.
4. **Pointcut:** Set of one or more join points where an advice need to be executed.
5. **Introduction:** This is the place where we can add new methods or attributes to the existing classes.
6. **Target Object:** One or more aspects will be there to provide advice for the target object.
7. **Weaving:** Linking aspects with other application types or objects. It can be done at compile-time/runtime/loadtime.

In this article, we will briefly see about the different kinds of advice available.

@Around

This is the most effective and desirable advice among all other advice. The first parameter is of type ProceedingJoinPoint. Code must have proceed()

on the ProceedingJoinPoint. When this is given, the code has to be executed before and after when the method is matched with the pointcut.

Sample code snippet:

Priority-wise always @Around will be invoked even if there are @Before annotations,

```
// Displays all the available methods i.e. the advice will
// be called for all the methods. The method declaration is
// called the pointcut signature. It provides a name that can
// be used by advice annotations to refer to that pointcut.
@Pointcut(
    value
    = "execution(* com.gfg.examples.service.ServiceExample.*(..))")
private void
printLogs()
{
}

// Declares the around advice that is applied before and
// after the method matching with a pointcut expression. Even
// there are @Before annotations, @Around will be invoked
// first with the before invocation and then only @Before
// will be called
@Around(value = "printLogs()")
public void
logsAroundAdvice(ProceedingJoinPoint proJoinPoint)
    throws Throwable
{
    System.out.println(
        "The method aroundAdvice() before invocation of the method "
        + proJoinPoint.getSignature().getName()
        + " method");
    try {
        proJoinPoint.proceed();
    }
    finally {
    }
    System.out.println(
        "The method aroundAdvice() after invocation of the method "
        + proJoinPoint.getSignature().getName()
        + " method");
}
```



As always this is the starting point, business logic needs to be processed here and it is ideal.

Possible Use-cases:

- When employee pay roll calculation has been initiated.

- When a student started writing the exam
- When an external application like a printer/scanner is initiated in the mentioned timeframe.

As @Around advice will inform about each and every activity before and after the progress, it will be a benchmark and also errors can be easily diagnosed via this approach.

@Before

Priority wise it may be the first step if there is no @Around advice. If @Around is there, always @Around will have the higher priority and @Before will run after the beginning portion of @Around.

Sample code snippet:

```
// If there is no @Around advice, @Before will be called
// first, otherwise @Around Before Invocation is called
@Before(
    "execution(* com.geeksforgeeks.examples.service.ServiceExample.*(..))")
public void
printLogStatementsBefore()
{
    System.out.println(
        ".....Looking for @Around advice, if none is there, @Before
will be called first. My role is to execute before each and every
method.....");
}
```

Apt scenario to have @Before:

Usually, in the master-detail relationship, the master record should exist and then only corresponding child data can be inserted similarly while deleting, child data should be deleted and then only master data can be deleted. @Before will help to overcome these dependencies scenarios at the beginning itself.

@After

Priority wise always @Around will be executed, if it is available and after that only @After will execute. This will be a good place to conclude the end of an activity.

Sample code snippet:

```
// If there is no @Around advice, @After will be called
// after @Before(if available) first, otherwise @Around After
// Invocation is called
@After(
    "execution(* com.gfg.examples.service.ServiceExample.*(..))")
public void
printLogStatementsAfter()
{
    System.out.println(
        ".....Looking for @Around advice, if none is there, @After
        will be called after @Before(if available). My role is to execute after each
        and every method.....");
}
```

Apt scenario to have @After:

To inform the users about the completion of an operation like completion of the download process of a file or an image or the printing job completed etc.,

@AfterReturning

Thought @After advice is there, @AfterReturning advice will indicate the success of an operation. Hence the success steps will be indicated here.

Sample code snippet:

```
// implementing after returning advice
// This is generally used to indicate the output after
// successful return of the method, will be called at Last
// i.e. after @Around
@AfterReturning(
    value
    = "execution(* com.gfg.examples.service.ServiceExample.*(..))",
    returning = "account")
public void
logsAfterReturningDisplay(JoinPoint joinPoint)
{
    System.out.println("After Returning method:"
                        + joinPoint.getSignature());
}
```

Apt Scenario to have @AfterReturning:

When there are activities like printer jobs, we need to know about the status of completion of printing jobs. Once it is successfully finished, need to initiate other subsequent activities. So success related operations can be provided in @AfterReturning

@AfterThrowing

There are many scenarios of failure of an activity, for example, when there is no power, the printing job cannot be completed. During those times, need to inform about the cause of failure and subsequent recovery measures. @AfterThrowing is the right place for it.

Sample code snippet:

```
// implementing after throwing advice
// This is generally used to indicate the exception in case
// of exception , will be called whenever exception occurs
@AfterThrowing(
    value
    = "execution(* com.gfg.examples.service.ServiceExample.*(..))",
    throwing = "ex")
public void
logsAfterThrowingDisplay(JoinPoint jPoint, Exception ex)
{
    System.out.println("After Throwing exception in method:"
                       + jPoint.getSignature());
    System.out.println("Exception is:" + ex.getMessage());
}
```



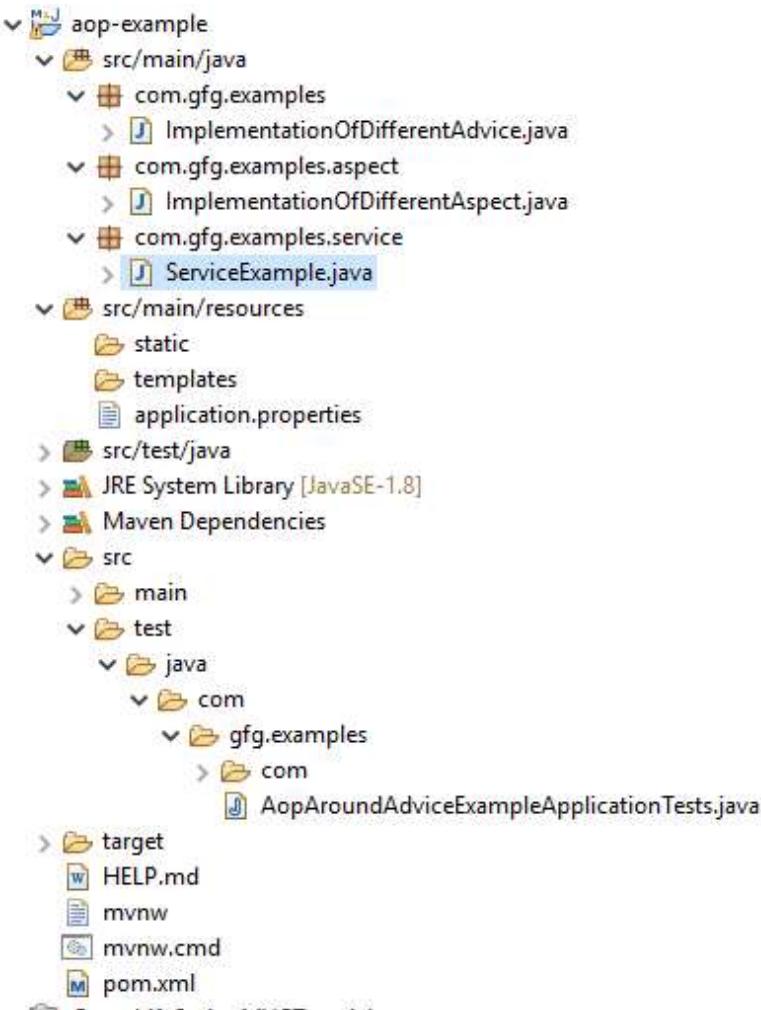
Apt Scenario for @AfterThrowing:

Due to an I/O error, Arithmetic Exceptions, SQLExceptions, etc., we need to provide corrective measures.

Let us combine everything and check as a sample maven project.

Example Project

Project Structure:



pom.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://maven.apache.org/POM/4.0.0"
          xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
                               https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.2.RELEASE</version>
        <relativePath/>
    </parent>
    <groupId>com.gfg.examples</groupId>
    <artifactId>aop-example</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>aop-example</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>

```

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

Key important files

ServiceExample.java

```

import org.springframework.stereotype.Service;

@Service
public class ServiceExample {
    public void getAccountBalance(String employeeAccountNumber) {
        System.out.println("Inside getBalance() method");
        // To mention about for a certain
        // employeeAccountNumber value
        if (employeeAccountNumber.equals("Emp1212")) {
            System.out.println("Total balance: .....");
        }
        else {
            System.out.println(
                "Sorry! wrong account number. Please give correct account
number to verify");
        }
    }
}

```

```

}

public String employeeStatus(String employeeNumber)
{
    System.out.println(
        "Inside checkEmployeeExistence() method");
    String status = null;
    if (employeeNumber.equals("emp12345")) {
        System.out.println(employeeNumber
            + " is currently active");
        status = "active";
    }
    else {
        System.out.println(employeeNumber
            + " is currently inactive");
        status = "Inactive";
    }
    return status;
}

public String
eligibilityForPromotion(int promotionExamMarks)
{
    System.out.println(
        "Inside eligibilityForPromotion() method");
    String status = null;
    if (promotionExamMarks >= 650) {
        System.out.println("Eligible for promotion..");
        status = "eligible";
    }
    else {
        System.out.println(
            "Not eligible for promotion..");
        status = "not eligible";
    }
    return status;
}
}

```

ImplementationOfDifferentAspect.java

```

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.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

// Enables the spring AOP functionality in an application
@Aspect
@Component

```

```

public class ImplementationOfDifferentAspect {
    // Displays all the available methods i.e. the advice
    // will be called for all the methods The method
    // declaration is called the pointcut signature. It
    // provides a name that can be used by advice annotations
    // to refer to that pointcut.
    @Pointcut(
        value
        = "execution(* com.gfg.examples.service.ServiceExample.*(..))")
    private void
    printLogs()
    {
    }
    // If there is no @Around advice, @Before will be called
    // first, otherwise @Around Before Invocation is called
    @Before(
        "execution(* com.gfg.examples.service.ServiceExample.*(..))")
    public void
    printLogStatementsBefore()
    {

        System.out.println(
            ".....Looking for @Around advice, if none is there,
@Before will be called first. My role is to execute before each and every
method.....");
    }
    // If there is no @Around advice, @After will be called
    // after @Before(if available) first, otherwise @Around
    // After Invocation is called
    @After(
        "execution(* com.gfg.examples.service.ServiceExample.*(..))")
    public void
    printLogStatementsAfter()
    {
        System.out.println(
            ".....Looking for @Around advice, if none is there,
@After will be called after @Before(if available). My role is to execute
after each and every method.....");
    }

    // implementing after returning advice
    // This is generally used to indicate the output after
    // successful return of the method, will be called at
    // last i.e. after @Around
    // AOP aspect plug in place is JointPoint
    @AfterReturning(
        value
        = "execution(* com.gfg.examples.service.ServiceExample.*(..)",
        returning = "account")
    public void
    logsAfterReturningDisplay(JoinPoint joinPoint)
    {
        System.out.println("After Returning method:"
                           + joinPoint.getSignature());
        // System.out.println(account);
    }
}

```

```

// implementing after throwing advice
// This is generally used to indicate the exception in
// case of exception , will be called whenever exception
// occurs
@AfterThrowing(
    value
    = "execution(* com.gfg.examples.service.ServiceExample.*(..))",
    throwing = "ex")
public void
logsAfterThrowingDisplay(JoinPoint jPoint, Exception ex)
{
    System.out.println(
        "After Throwing exception in method:"
        + jPoint.getSignature());
    System.out.println("Exception is:"
        + ex.getMessage());
}

// Declares the around advice that is applied before and
// after the method matching with a pointcut expression
// Even there are @Before annotations, @Around will be
// invoked first with the before invocation and then only
// @Before will be called
@Around(value = "printLogs()")
public void
logsAroundAdvice(ProceedingJoinPoint proJoinPoint)
    throws Throwable
{
    System.out.println(
        "The method aroundAdvice() before invocation of the method "
        + proJoinPoint.getSignature().getName()
        + " method");
    try {
        proJoinPoint.proceed();
    }
    finally {
    }
    System.out.println(
        "The method aroundAdvice() after invocation of the method "
        + proJoinPoint.getSignature().getName()
        + " method");
}
}

```

ImplementationOfDifferentAdvice.java

```

import com.gfg.examples.service.ServiceExample;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@SpringBootApplication
// @EnableAspectJAutoProxy annotation enables support for

```

```

// handling the components marked with @Aspect annotation.
@EnableAspectJAutoProxy
public class ImplementationOfDifferentAdvice {
    public static void main(String[] args)
    {
        ConfigurableApplicationContext context
            = SpringApplication.run(
                ImplementationOfDifferentAdvice.class,
                args);
        // Fetching the object from the application
        // context.
        ServiceExample service
            = context.getBean(ServiceExample.class);
        // checking for an employee available in the
        // organization
        String employeeNumber = "emp12345";
        try {
            service.employeeStatus(employeeNumber);
        }
        catch (Exception ex) {
            System.out.println("Exception occurred.."
                + ex.getMessage());
        }
        // Displaying balance in the account.
        String employeeAccountNumber = "Emp1212";
        try {
            service.getAccountBalance(
                employeeAccountNumber);
        }
        catch (Exception ex) {
            System.out.println("Exception occurred.."
                + ex.getMessage());
        }

        // Employee has undergone some exams for promotion.
        // Let us check that
        int promotionExamMarks = 650;
        try {
            service.eligibilityForPromotion(
                promotionExamMarks);
        }
        catch (Exception ex) {
            System.out.println("Exception occurred.."
                + ex.getMessage());
        }
        // Closing the context object.
        context.close();
    }
}

```

Output:

The method aroundAdvice() before invocation of the method employeeStatus method
Looking for @Around advice, if none is there, @Before will be called first. My role is to execute before each and every method.....
 Inside checkEmployeeExistence() method
 emp12345 is currently active
 The method aroundAdvice() after invocation of the method employeeStatus method
Looking for @Around advice, if none is there, @After will be called after @Before(if available). My role is to execute after each and every
 After Returing method:String com.gfg.examples.service.ServiceExample.employeeStatus(String)
 The method aroundAdvice() before invocation of the method getAccountBalance method
Looking for @Around advice, if none is there, @Before will be called first. My role is to execute before each and every method.....
 Inside getBalance() method
 Total balance:
 The method aroundAdvice() after invocation of the method getAccountBalance method
Looking for @Around advice, if none is there, @After will be called after @Before(if available). My role is to execute after each and every
 After Returing method:void com.gfg.examples.service.ServiceExample.getAccountBalance(String)
 The method aroundAdvice() before invocation of the method eligibilityForPromotion method
Looking for @Around advice, if none is there, @Before will be called first. My role is to execute before each and every method.....
 Inside eligibilityForPromotion() method
 Eligible for promotion..
 The method aroundAdvice() after invocation of the method eligibilityForPromotion method
Looking for @Around advice, if none is there, @After will be called after @Before(if available). My role is to execute after each and every

Explanation:

1. employeeStatus method

```
// @around advice begins
```

The method aroundAdvice() before invocation of the method
 employeeStatus method

```
// @before advice
```

.....Looking for @Around advice, if none is there, @Before
 will be called first. My role is to execute before each and every
 method.....

```
// execution steps
```

Inside() method

emp12345 is currently active

```
// @around advice ends
```

The method aroundAdvice() after invocation of the method
 employeeStatus method

```
// @after advice
```

.....Looking for @Around advice, if none is there, @After
 will be called after @Before(if available). My role is to execute
 after each and every method.....

@afterreturning

After Returning method:String

com.gfg.examples.service.ServiceExample.employeeStatus(String)

```
// Similarly it will be done for other methods
```

In case we have run the method with inputs that cause exceptions, maybe in the below example we can try

```
// Displaying balance in the account.
String employeeAccountNumber = null;
try {
    service.getAccountBalance(employeeAccountNumber);
}
catch (Exception ex) {
    System.out.println("Exception occurred.."
        + ex.getMessage());
}
```



Output:

```
Inside getBalance() method
.....Looking for @Around advice, if none is there, @After will be called after @Before(if available). My role is to execute after each and every
After Throwing exception in method: void com.gfg.examples.service.ServiceExample.getAccountBalance(String)
Exception is:Cannot invoke "String.equals(Object)" because "employeeAccountNumber" is null
Exception occurred..Cannot invoke "String.equals(Object)" because "employeeAccountNumber" is null
The method aroundAdvice() before invocation of the method eligibilityForPromotion method
.....Looking for @Around advice, if none is there, @Before will be called first. My role is to execute before each and every method.....
Inside eligibilityForPromotion() method
Eligible for promotion..
The method aroundAdvice() after invocation of the method eligibilityForPromotion method
.....Looking for @Around advice, if none is there. @After will be called after @Before(if available). My role is to execute after each and every
```

In code, accountnumber
is null and hence this
exception

Throughout the samples, if we look, we have used an apt way of providing @Around, @Before, @After, @AfterReturning, and @AfterThrowing advices and in this way, we can implement AOP functionality easily.

[Comment](#)
[More info](#)
[Campus Training Program](#)


Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate
Tower, Sector- 136, Noida, Uttar Pradesh