# **AOP with Spring Framework**

One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java, and others.

Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

AOP Terminologies

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

|  |  |
| --- | --- |
| **Sr.No** | **Terms & Description** |
| 1 | **Aspect**  This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. |
| 2 | **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. |
| 3 | **Advice**  This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework. |
| 4 | **Pointcut**  This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples. |
| 5 | **Introduction**  An introduction allows you to add new methods or attributes to the existing classes. |
| 6 | **Target object**  The object being advised by one or more aspects. This object will always be a proxied object, also referred to as the advised object. |
| 7 | **Weaving**  Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. |

Types of Advice

Spring aspects can work with five kinds of advice mentioned as follows −

|  |  |
| --- | --- |
| **Sr.No** | **Advice & Description** |
| 1 | **before**  Run advice before the a method execution. |
| 2 | **after**  Run advice after the method execution, regardless of its outcome. |
| 3 | **after-returning**  Run advice after the a method execution only if method completes successfully. |
| 4 | **after-throwing**  Run advice after the a method execution only if method exits by throwing an exception. |
| 5 | **around**  Run advice before and after the advised method is invoked. |

Custom Aspects Implementation

Spring supports the **@AspectJ annotation style** approach and the **schema-based** approach to implement custom aspects. These two approaches have been explained in detail in the following sections.

|  |  |
| --- | --- |
| **Sr.No** | **Approach & Description** |
| 1 | [XML Schema based](https://www.tutorialspoint.com/spring/schema_based_aop_appoach.htm)  Aspects are implemented using the regular classes along with XML based configuration. |
| 2 | [@AspectJ based](https://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm)  @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Additional:

AOP addresses the problem of *cross-cutting concerns*, which would be any kind of code that is repeated in different methods and can't normally be completely refactored into its own module, like with logging or verification. So, with AOP you can leave that stuff out of the main code and define it vertically like so:

function mainProgram()

{

var x = foo();

doSomethingWith(x);

return x;

}

aspect logging

{

before (mainProgram is called):

{

log.Write("entering mainProgram");

}

after (mainProgram is called):

{

log.Write( "exiting mainProgram with return value of "

+ mainProgram.returnValue);

}

}

aspect verification

{

before (doSomethingWith is called):

{

if (doSomethingWith.arguments[0] == null)

{

throw NullArgumentException();

}

if (!doSomethingWith.caller.isAuthenticated)

{

throw Securityexception();

}

}

}

And then an *aspect-weaver* is used to compile the code into this:

function mainProgram()

{

log.Write("entering mainProgram");

var x = foo();

if (x == null) throw NullArgumentException();

if (!mainProgramIsAuthenticated()) throw Securityexception();

doSomethingWith(x);

log.Write("exiting mainProgram with return value of "+ x);

return x;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **@AspectJ Based AOP with Spring**

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ support is enabled by including the following element inside your XML Schema-based configuration file.

<aop:aspectj-autoproxy/>

You will also need the following AspectJ libraries on the classpath of your application. These libraries are available in the 'lib' directory of an AspectJ installation, otherwise you can download them from the internet.

* aspectjrt.jar
* aspectjweaver.jar
* aspectj.jar
* aopalliance.jar

## Declaring an aspect

Aspects classes are like any other normal bean and may have methods and fields just like any other class, except that they will be annotated with @Aspect as follows −

package org.xyz;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class AspectModule {

}

They will be configured in XML like any other bean as follows −

<bean id = "myAspect" class = "org.xyz.AspectModule">

<!-- configure properties of aspect here as normal -->

</bean>

## Declaring a pointcut

A **pointcut** helps in determining the join points (ie methods) of interest to be executed with different advices. While working with @AspectJ-based configuration, pointcut declaration has two parts −

* A pointcut expression that determines exactly which method executions we are interested in.
* A pointcut signature comprising a name and any number of parameters. The actual body of the method is irrelevant and in fact should be empty.

The following example defines a pointcut named 'businessService' that will match the execution of every method available in the classes under the package com.xyz.myapp.service −

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.xyz.myapp.service.\*.\*(..))") // expression

private void businessService() {} // signature

The following example defines a pointcut named 'getname' that will match the execution of getName() method available in the Student class under the package com.demo −

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.demo.Student.getName(..))")

private void getname() {}

## Declaring advices

You can declare any of the five advices using @{ADVICE-NAME} annotations as given in the code snippet. This assumes that you already have defined a pointcut signature method businessService() −

@Before("businessService()")

public void doBeforeTask(){

...

}

@After("businessService()")

public void doAfterTask(){

...

}

@AfterReturning(pointcut = "businessService()", returning = "retVal")

public void doAfterReturnningTask(Object retVal) {

// you can intercept retVal here.

...

}

@AfterThrowing(pointcut = "businessService()", throwing = "ex")

public void doAfterThrowingTask(Exception ex) {

// you can intercept thrown exception here.

...

}

@Around("businessService()")

public void doAroundTask(){

...

}

You can define a pointcut inline for any of the advices. Following is an example to define inline pointcut for before advice −

@Before("execution(\* com.xyz.myapp.service.\*.\*(..))")

public doBeforeTask(){

...

}

## @AspectJ Based AOP Example

To understand the above-mentioned concepts related to @AspectJ based AOP, let us write an example which will implement few of the advices. To write our example with few advices, let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.demo* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Add Spring AOP specific libraries **aspectjrt.jar, aspectjweaver.jar** and **aspectj.jar** in the project. |
| 4 | Create Java classes **Logging**, *Student* and *MainApp* under the *com.demo* package. |
| 5 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 6 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **Logging.java** file. This is actually a sample of aspect module which defines methods to be called at various points.

package com.demo;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Around;

@Aspect

public class Logging {

/\*\* Following is the definition for a pointcut to select

\* all the methods available. So advice will be called

\* for all the methods.

\*/

@Pointcut("execution(\* com.demo.\*.\*(..))")

private void selectAll(){}

/\*\*

\* This is the method which I would like to execute

\* before a selected method execution.

\*/

@Before("selectAll()")

public void beforeAdvice(){

System.out.println("Going to setup student profile.");

}

/\*\*

\* This is the method which I would like to execute

\* after a selected method execution.

\*/

@After("selectAll()")

public void afterAdvice(){

System.out.println("Student profile has been setup.");

}

/\*\*

\* This is the method which I would like to execute

\* when any method returns.

\*/

@AfterReturning(pointcut = "selectAll()", returning = "retVal")

public void afterReturningAdvice(Object retVal){

System.out.println("Returning:" + retVal.toString() );

}

/\*\*

\* This is the method which I would like to execute

\* if there is an exception raised by any method.

\*/

@AfterThrowing(pointcut = "selectAll()", throwing = "ex")

public void AfterThrowingAdvice(IllegalArgumentException ex){

System.out.println("There has been an exception: " + ex.toString());

}

}

Following is the content of the **Student.java** file

package com.demo;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

System.out.println("Age : " + age );

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

System.out.println("Name : " + name );

return name;

}

public void printThrowException(){

System.out.println("Exception raised");

throw new IllegalArgumentException();

}

}

Following is the content of the **MainApp.java** file

package com.demo;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

Student student = (Student) context.getBean("student");

student.getName();

student.getAge();

student.printThrowException();

}

}

Following is the configuration file **Beans.xml**

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop = "http://www.springframework.org/schema/aop"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

<aop:aspectj-autoproxy/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.demo.Student">

<property name = "name" value = "Zara" />

<property name = "age" value = "11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id = "logging" class = "com.demo.Logging"/>

</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Going to setup student profile.

Name : Zara

Student profile has been setup.

Returning:Zara

Going to setup student profile.

Age : 11

Student profile has been setup.

Returning:11

Going to setup student profile.

Exception raised

Student profile has been setup.

There has been an exception: java.lang.IllegalArgumentException

.....

other exception content