Spring AOP Example Tutorial – Aspect, Advice, Pointcut, JoinPoint, Annotations, XML Configuration

Spring Framework is developed on two core concepts – Dependency Injection and Aspect Oriented Programming (AOP). We have already see how Spring Dependency Injection works, today we will look into the core concepts of Aspect Oriented Programming and how we can implement it using Spring Framework.

Aspect Oriented Programming Overview

Most of the enterprise applications have some common crosscutting concerns that is applicable for different types of Objects and modules. Some of the common crosscutting concerns are logging, transaction management, data validation etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in Aspect Oriented Programming application modularity is achieved by Aspects and they are configured to cut across different classes.

AOP takes out the direct dependency of crosscutting tasks from classes that we can't achieve through normal object oriented programming model. For example, we can have a separate class for logging but again the functional classes will have to call these methods to achieve logging across the application.

Aspect Oriented Programming Core Concepts

Before we dive into implementation of AOP in Spring Framework, we should understand the core concepts of AOP.

1. **Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring AspectJ integration to define a class as Aspect using `@Aspect` annotation.
2. **Join Point**: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.
3. **Advice**: Advices are actions taken for a particular join point. In terms of programming, they are methods that gets executed when a certain join point with matching pointcut is reached in the application. You can think of Advices as Struts2 interceptors or Servlet Filters.
4. **Pointcut**: Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression

language.

5. **Target Object**: They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.
6. **AOP proxy**: Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes. We can also use CGLIB proxy by adding it as the dependency in the Spring AOP project.
7. **Weaving**: It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

AOP Advice Types

Based on the execution strategy of advices, they are of following types.

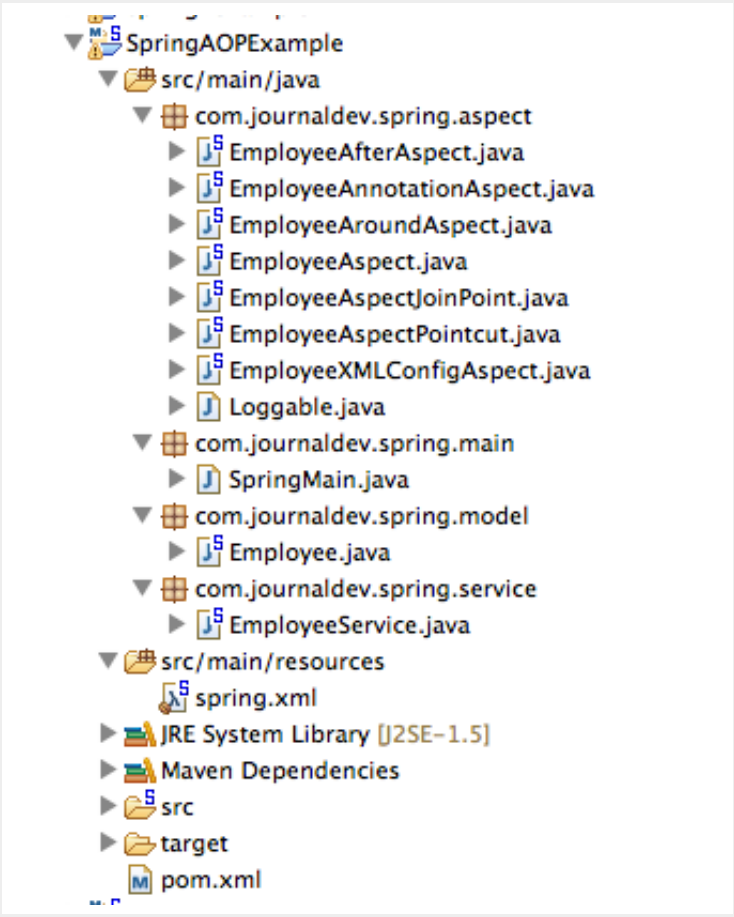
1. **Before Advice**: These advices runs before the execution of join point methods. We can use `@Before` annotation to mark an advice type as Before advice.
2. **After (finally) Advice**: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using `@After` annotation.
3. **After Returning Advice**: Sometimes we want advice methods to execute only if the join point method executes normally. We can use `@AfterReturning` annotation to mark a method as after returning advice.
4. **After Throwing Advice**: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively. We use `@AfterThrowing` annotation for this type of advice.
5. **Around Advice**: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use `@Around` annotation to create around advice methods.

The points mentioned above may sound confusing but when we will look at the implementation of Spring AOP, things will be more clear. Let's start creating a simple Spring project with AOP implementations. Spring provides support for using AspectJ annotations to create aspects and we will be using that for simplicity. All the above AOP annotations are defined in `org.aspectj.lang.annotation` package.

Spring Tool Suite provides useful information about the aspects, so I would suggest you to use it. If you are not familiar with STS, I would recommend you to have a look at Spring MVC Tutorial where

I have explained how to use it.

Create a new Simple Spring Maven project so that all the Spring Core libraries are included in the pom.xml files and we don't need to include them explicitly. Our final project will look like below image, we will look into the Spring core components and Aspect implementations in detail.



Spring AOP AspectJ Dependencies

Spring framework provides AOP support by default but since we are using AspectJ annotations for configuring aspects and advices, we would need to include them in the pom.xml file.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.o
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.springframework.samples</groupId>
    <artifactId>SpringAOPExample</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>

        <!-- Generic properties -->
        <java.version>1.6</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEnco

        <!-- Spring -->
        <spring-framework.version>4.0.2.RELEASE</spring-framework.version>

        <!-- Logging -->
```

```
        <logback.version>1.0.13</logback.version>
        <slf4j.version>1.7.5</slf4j.version>

        <!-- Test -->
        <junit.version>4.11</junit.version>

        <!-- AspectJ -->
        <aspectj.version>1.7.4</aspectj.version>

    </properties>

    <dependencies>
        <!-- Spring and Transactions -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring-framework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>${spring-framework.version}</version>
        </dependency>

        <!-- Logging with SLF4J & LogBack -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>${logback.version}</version>
            <scope>runtime</scope>
        </dependency>

        <!-- AspectJ dependencies -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
    </dependencies>
</project>
```

Notice that I have added aspectjrt and aspectjtools dependencies (version 1.7.4) in the project. Also I have updated the Spring framework version to be the latest one as of date i.e 4.0.2.RELEASE.

Model Class

Let's create a simple java bean that we will use for our example with some additional methods.

Employee.java

Employee.java

```java
package com.journaldev.spring.model;

import com.journaldev.spring.aspect.Loggable;

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

    @Loggable
    public void setName(String nm) {
        this.name=nm;
    }

    public void throwException(){
        throw new RuntimeException("Dummy Exception");
    }

}
```

Did you noticed that *setName()* method is annotated with `Loggable` annotation. It is a custom java annotation defined by us in the project. We will look into it's usage later on.

Service Class

Let's create a service class to work with Employee bean.

EmployeeService.java

```java
package com.journaldev.spring.service;

import com.journaldev.spring.model.Employee;

public class EmployeeService {

    private Employee employee;

    public Employee getEmployee(){
        return this.employee;
    }

    public void setEmployee(Employee e){
        this.employee=e;
    }
}
```

I could have used Spring annotations to configure it as a Spring Component, but we will use XML based configuration in this project. EmployeeService class is very standard and just provides us an access point for Employee beans.

Spring Bean Configuration with AOP

If you are using STS, you have option to create "Spring Bean Configuration File" and chose AOP

schema namespace but if you are using some other IDE, you can simply add it in the spring bean configuration file.

My project bean configuration file looks like below.

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

<!-- Enable AspectJ style of Spring AOP -->
<aop:aspectj-autoproxy />

<!-- Configure Employee Bean and initialize it -->
<bean name="employee" class="com.journaldev.spring.model.Employee">
    <property name="name" value="Dummy Name"></property>
</bean>

<!-- Configure EmployeeService bean -->
<bean name="employeeService" class="com.journaldev.spring.service.EmployeeSer
    <property name="employee" ref="employee"></property>
</bean>

<!-- Configure Aspect Beans, without this Aspects advices wont execute -->
<bean name="employeeAspect" class="com.journaldev.spring.aspect.EmployeeAspect
<bean name="employeeAspectPointcut" class="com.journaldev.spring.aspect.Emplo
<bean name="employeeAspectJoinPoint" class="com.journaldev.spring.aspect.Empl
<bean name="employeeAfterAspect" class="com.journaldev.spring.aspect.Employee
<bean name="employeeAroundAspect" class="com.journaldev.spring.aspect.Employe
<bean name="employeeAnnotationAspect" class="com.journaldev.spring.aspect.Emp

</beans>
```

For using AOP in Spring beans, we need to do following:

1. Declare AOP namespace like xmlns:aop="http://www.springframework.org/schema/aop"
2. Add aop:aspectj-autoproxy element to enable Spring AspectJ support with auto proxy at runtime
3. Configure Aspect classes as other Spring beans

You can see that I have a lot of aspects defined in the spring bean configuration file, it's time to look into those one by one.

Before Aspect Example

EmployeeAspect.java

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
```

```
@Aspect
public class EmployeeAspect {

    @Before("execution(public String getName())")
    public void getNameAdvice(){
        System.out.println("Executing Advice on getName()");
    }

    @Before("execution(* com.journaldev.spring.service.*.get*())")
    public void getAllAdvice(){
        System.out.println("Service method getter called");
    }
}
```

Important points in above aspect class is:

- Aspect classes are required to have `@Aspect` annotation.
- @Before annotation is used to create Before advice
- The string parameter passed in the `@Before` annotation is the Pointcut expression
- *getNameAdvice()* advice will execute for any Spring Bean method with signature `public String getName()`. This is a very important point to remember, if we will create Employee bean using new operator the advices will not be applied. Only when we will use ApplicationContext to get the bean, advices will be applied.
- We can use asterisk (*) as wild card in Pointcut expressions, *getAllAdvice()* will be applied for all the classes in `com.journaldev.spring.service` package whose name starts with `get` and doesn't take any arguments.

We will look these advices in action in a test class after we have looked into all the different types of advices.

Pointcut Methods and Reuse

Sometimes we have to use same Pointcut expression at multiple places, we can create an empty method with `@Pointcut` annotation and then use it as expression in advices.

EmployeeAspectPointcut.java

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class EmployeeAspectPointcut {

    @Before("getNamePointcut()")
    public void loggingAdvice(){
        System.out.println("Executing loggingAdvice on getName()");
    }

```

```
    @Before("getNamePointcut()")
    public void secondAdvice(){
        System.out.println("Executing secondAdvice on getName()");
    }

    @Pointcut("execution(public String getName())")
    public void getNamePointcut(){}

    @Before("allMethodsPointcut()")
    public void allServiceMethodsAdvice(){
        System.out.println("Before executing service method");
    }

    //Pointcut to execute on all the methods of classes in a package
    @Pointcut("within(com.journaldev.spring.service.*)")
    public void allMethodsPointcut(){}

}
```

Above example is very clear, rather than expression we are using method name in the advice annotation argument.

JoinPoint and Advice Arguments

We can use JoinPoint as parameter in the advice methods and using it get the method signature or the target object.

We can use `args()` expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined. We can use Generic objects also in the advice arguments.

EmployeeAspectJoinPoint.java

```
package com.journaldev.spring.aspect;

import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeAspectJoinPoint {


    @Before("execution(public void com.journaldev.spring.model..set*(*))")
    public void loggingAdvice(JoinPoint joinPoint){
        System.out.println("Before running loggingAdvice on method="+joinPoint

        System.out.println("Agruments Passed=" + Arrays.toString(joinPoint.get

    }

    //Advice arguments, will be applied to bean methods with single String arg
    @Before("args(name)")
    public void logStringArguments(String name){
        System.out.println("String argument passed="+name);
    }
```

```
}
```

After Advice Example

Let's look at a simple aspect class with example of After, After Throwing and After Returning advices.

EmployeeAfterAspect.java

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class EmployeeAfterAspect {

    @After("args(name)")
    public void logStringArguments(String name){
        System.out.println("Running After Advice. String argument passed="+nar
    }

    @AfterThrowing("within(com.journaldev.spring.model.Employee)")
    public void logExceptions(JoinPoint joinPoint){
        System.out.println("Exception thrown in Employee Method="+joinPoint.t
    }

    @AfterReturning(pointcut="execution(* getName())", returning="returnStrin
    public void getNameReturningAdvice(String returnString){
        System.out.println("getNameReturningAdvice executed. Returned String=
    }

}
```

We can use `within` in pointcut expression to apply advice to all the methods in the class. We can use @AfterReturning advice to get the object returned by the advised method.
We have *throwException()* method in the Employee bean to showcase the use of After Throwing advice.

Spring Around Aspect Example

As explained earlier, we can use Around aspect to cut the method execution before and after. We can use it to control whether the advised method will execute or not. We can also inspect the returned value and change it. This is the most powerful advice and needs to be applied properly.

EmployeeAroundAspect.java

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
```

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

@Aspect
public class EmployeeAroundAspect {

    @Around("execution(* com.journaldev.spring.model.Employee.getName())")
    public Object employeeAroundAdvice(ProceedingJoinPoint proceedingJoinPoint
        System.out.println("Before invoking getName() method");
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("After invoking getName() method. Return value="+va
        return value;
    }
}
```

Around advice are always required to have ProceedingJoinPoint as argument and we should use it's proceed() method to invoke the target object advised method. If advised method is returning something, it's advice responsibility to return it to the caller program. For void methods, advice method can return null. Since around advice cut around the advised method, we can control the input and output of the method as well as it's execution behavior.

Advice with Custom Annotation Pointcut

If you look at all the above advices pointcut expressions, there are chances that they gets applied to some other beans where it's not intended. For example, someone can define a new spring bean with getName() method and the advices will start getting applied to that even though it was not intended. That's why we should keep the scope of pointcut expression as narrow as possible.

An alternative approach is to create a custom annotation and annotate the methods where we want the advice to be applied. This is the purpose of having Employee *setName()* method annotated with @Loggable annotation.

Spring Framework @Transactional annotation is a great example of this approach for Spring Transaction Management.

Loggable.java

```
package com.journaldev.spring.aspect;

public @interface Loggable {

}
```

EmployeeAnnotationAspect.java

```
package com.journaldev.spring.aspect;

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

```

```
@Aspect
public class EmployeeAnnotationAspect {

    @Before("@annotation(com.journaldev.spring.aspect.Loggable)")
    public void myAdvice(){
        System.out.println("Executing myAdvice!!");
    }
}
```

myAdvice() method will advice only setName() method. This is a very safe approach and whenever we want to apply the advice on any method, all we need is to annotate it with Loggable annotation.

Spring AOP XML Configuration

I always prefer annotation but we also have option to configure aspects in spring configuration file. For example, let's say we have a class as below.

EmployeeXMLConfigAspect.java

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

public class EmployeeXMLConfigAspect {

    public Object employeeAroundAdvice(ProceedingJoinPoint proceedingJoinPoint
        System.out.println("EmployeeXMLConfigAspect:: Before invoking getName
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("EmployeeXMLConfigAspect:: After invoking getName(
        return value;
    }
}
```

We can configure it by including following configuration in the Spring Bean config file.

```
<bean name="employeeXMLConfigAspect" class="com.journaldev.spring.aspect.Emplo

<!-- Spring AOP XML Configuration -->
<aop:config>
    <aop:aspect ref="employeeXMLConfigAspect" id="employeeXMLConfigAspectID" or
        <aop:pointcut expression="execution(* com.journaldev.spring.model.Empl
        <aop:around method="employeeAroundAdvice" pointcut-ref="getNamePointcu
    </aop:aspect>
</aop:config>
```

AOP xml config elements purpose is clear from their name, so I won't go into much detail about it.

Spring AOP in Action

Let's have a simple Spring program and see how all these aspects cut through the bean methods.

SpringMain.java

```java
package com.journaldev.spring.main;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.service.EmployeeService;

public class SpringMain {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContex
        EmployeeService employeeService = ctx.getBean("employeeService", Emplo

        System.out.println(employeeService.getEmployee().getName());

        employeeService.getEmployee().setName("Pankaj");

        employeeService.getEmployee().throwException();

        ctx.close();
    }

}
```

Now when we execute above program, we get following output.

```
Mar 20, 2014 8:50:09 PM org.springframework.context.support.ClassPathXmlApplic
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationC
Mar 20, 2014 8:50:09 PM org.springframework.beans.factory.xml.XmlBeanDefiniti
INFO: Loading XML bean definitions from class path resource [spring.xml]
Service method getter called
Before executing service method
EmployeeXMLConfigAspect:: Before invoking getName() method
Executing Advice on getName()
Executing loggingAdvice on getName()
Executing secondAdvice on getName()
Before invoking getName() method
After invoking getName() method. Return value=Dummy Name
getNameReturningAdvice executed. Returned String=Dummy Name
EmployeeXMLConfigAspect:: After invoking getName() method. Return value=Dummy
Dummy Name
Service method getter called
Before executing service method
String argument passed=Pankaj
Before running loggingAdvice on method=execution(void com.journaldev.spring.mo
Agruments Passed=[Pankaj]
Executing myAdvice!!
Running After Advice. String argument passed=Pankaj
Service method getter called
Before executing service method
Exception thrown in Employee Method=execution(void com.journaldev.spring.model
Exception in thread "main" java.lang.RuntimeException: Dummy Exception
    at com.journaldev.spring.model.Employee.throwException(Employee.java:19)
    at com.journaldev.spring.model.Employee$$FastClassBySpringCGLIB$$da2dc051
    at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.i
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Re
    at org.springframework.aop.aspectj.AspectJAfterThrowingAdvice.invoke(Aspec
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Re
    at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Re
    at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedIntercept
```

```
37      at com.journaldev.spring.model.Employee$$EnhancerBySpringCGLIB$$3f881964.
38      at com.journaldev.spring.main.SpringMain.main(SpringMain.java:17)
```

You can see that advices are getting executed one by one based on their pointcut configurations. You should configure them one by one to avoid confusion.

That's all for Spring AOP Tutorial, I hope you learned the basics of AOP with Spring and can learn more from examples. Download the sample project from below link and play around with it.