Spring - AOP AspectJ Xml Configuration

Last Updated : 23 Jul, 2025

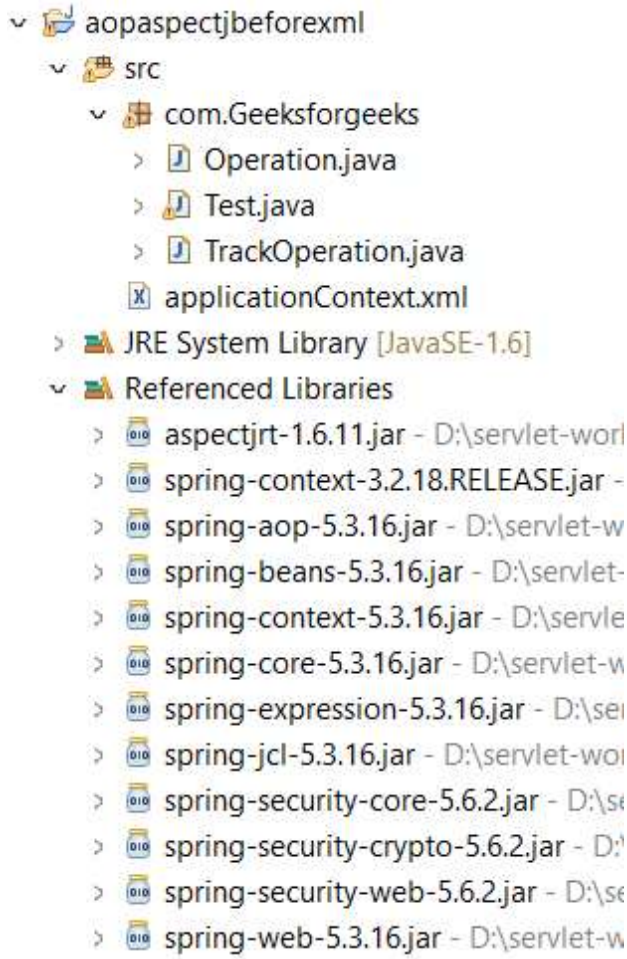
Spring allows you to specify aspects, recommendations, and pointcuts in an XML file. Annotations were used in the aop examples on the preceding page. The XML configuration file will now show us the same samples.

Let's look at the XML components that describe advice.

1. **aop:before:** It's used before the real business logic procedure is called.
2. **aop:after:** It is used after the real business logic method has been called.
3. **aop:after-returning:** It is applied after invoking the actual business logic method used. It may be used to intercept the advisory return value.
4. **aop:around:** It is used both before and after the real business logic method is called.
5. **aop:after-throwing:** If the real business logic procedure throws an exception, it is used.

A. aop:before Example

Before the main business logic procedure, the AspectJ Before Advice is used. Any action, such as conversion or authentication, can be performed here. Make a class with real business logic in it.



File: Operation.java

```
// Java Program to Illustrate Operation Class

package com.Geeksforgeeks;

// Class
public class Operation {

    // Method 1
    public void msg()
    {
        System.out.println("msg() method invoked");
    }
    // Method 2
    public int m()
    {
        System.out.println("m() method invoked");
        return 2;
    }
    // Method 3
    public int k()
    {
```

```
        System.out.println("k() method invoked");
        return 3;
    }
}
```

File: TrackOperation.java

```
package com.Geeksforgeeks;

import org.aspectj.lang.JoinPoint;

public class TrackOperation{

    public void myadvice(JoinPoint jp)//it is advice
    {
        System.out.println("additional concern");
        //System.out.println("Method Signature: "  + jp.getSignature());
    }
}
```

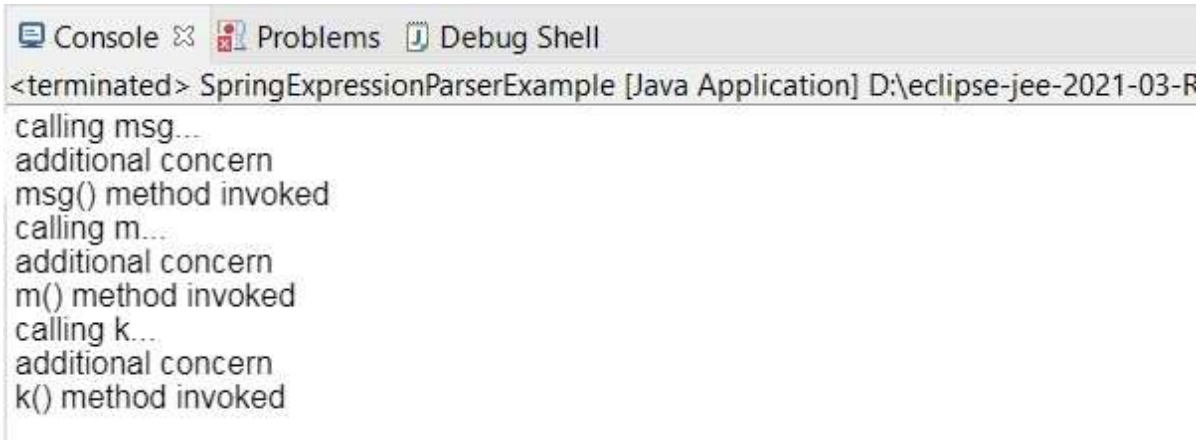
File: applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://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 />

<bean id="opBean" class="com.Geeksforgeeks.Operation">    </bean>

<bean id="trackAspect" class="com.Geeksforgeeks.TrackOperation"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="trackAspect" >
     <!-- @Before -->
     <aop:pointcut id="pointCutBefore"    expression="execution(*
com.Geeksforgeeks.Operation.*(..))" />
     <aop:before method="myadvice" pointcut-ref="pointCutBefore" />
  </aop:aspect>
</aop:config>

</beans>
```

File: Test.java

```java
// Java Program to Illustrate Application Class

package com.Geeksforgeeks;

// Importing required classes
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

// Class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of ApplicationContext
        // and Operation Class
        ApplicationContext context
            = new ClassPathXmlApplicationContext(
                "applicationContext.xml");
        Operation e = (Operation)context.getBean("opBean");

        // Print statements and calling methods
        / as defined in other class above System.out.println
        ("calling msg...");
        e.msg();
        System.out.println("calling m...");
        e.m();
        System.out.println("calling k...");
        e.k();
    }
}
```

Output:



B. aop:after Example

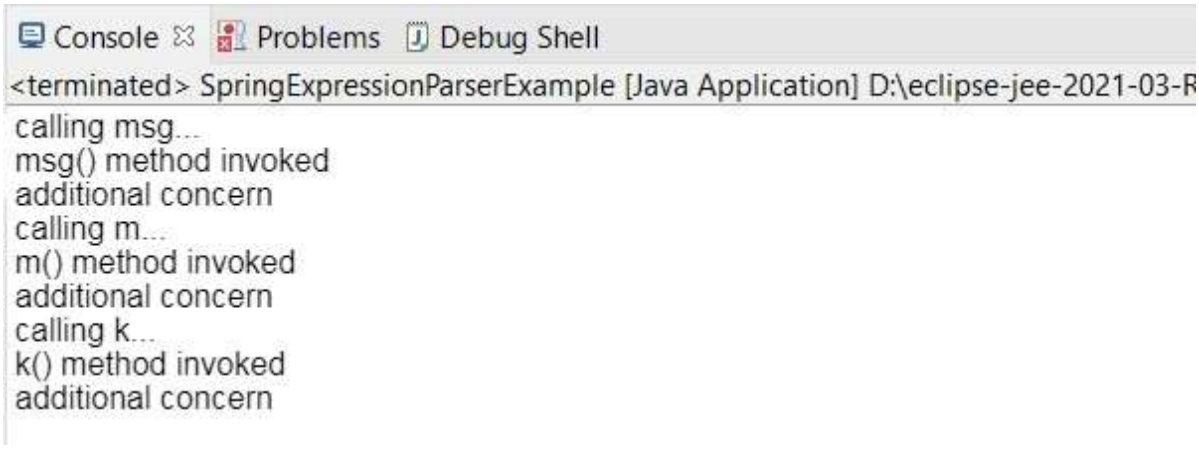
After invoking the real business logic methods, the AspectJ after guidance is implemented. It may be used to keep track of logs, security, and notifications, among other things. We'll assume that the files Operation.java, TrackOperation.java, and Test.java are identical to those in the aop:before example.

Create the applicationContext.xml file, which contains the bean definitions.

File: applicationContext.xml

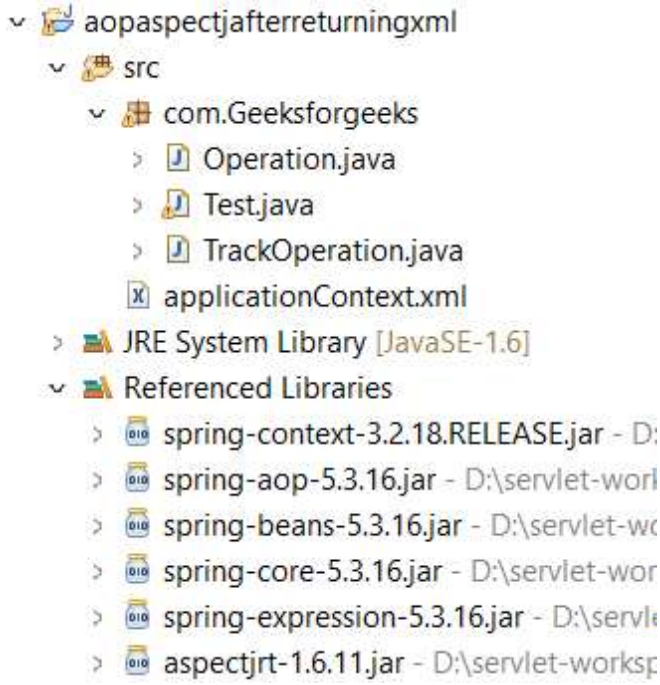
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop/"
    xsi:schemaLocation="http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/beans//spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop/
    http://www.springframework.org/schema/aop//spring-aop-3.0.xsd ">

<aop:aspectj-autoproxy />

<bean id="opBean" class="com.Geeksforgeeks.Operation">   </bean>
<bean id="trackAspect" class="com.Geeksforgeeks.TrackOperation"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="trackAspect" >
     <!-- @After -->
     <aop:pointcut id="pointCutAfter"    expression="execution(*
com.Geeksforgeeks.Operation.*(..))" />
     <aop:after method="myadvice" pointcut-ref="pointCutAfter" />
  </aop:aspect>
</aop:config>

</beans>
```

Output:



You can see that additional concern is printed after calling msg(), m(), and k() methods.

C. aop:after-returning example



We may receive the outcome in the advice by using after returning advice. Make a class to hold the business logic.

File: Operation.java

```
package com.Geeksforgeeks;
public  class Operation{
    public int m(){System.out.println("m() method invoked");return 2;}
```

```
    public int k(){System.out.println("k() method invoked");return 3;}
}
```

File: TrackOperation.java

```
package com.Geeksforgeeks;

import org.aspectj.lang.JoinPoint;

public class TrackOperation{
    public void myadvice(JoinPoint jp,Object result)//it is advice (after
advice)
    {
        System.out.println("additional concern");
        System.out.println("Method Signature: "  + jp.getSignature());
        System.out.println("Result in advice: "+result);
        System.out.println("end of after returning advice...");
    }
}
```

File: applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://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 />

<bean id="opBean" class="com.Geeksforgeeks.Operation">    </bean>

<bean id="trackAspect" class="com.Geeksforgeeks.TrackOperation"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="trackAspect" >
     <!-- @AfterReturning -->
     <aop:pointcut id="pointCutAfterReturning"    expression="execution(*
com.Geeksforgeeks.Operation.*(..))" />
     <aop:after-returning method="myadvice" returning="result" pointcut-
ref="pointCutAfterReturning" />
  </aop:aspect>
</aop:config>

</beans>
```

File: Test.java

```
package com.Geeksforgeeks;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test{
    public static void main(String[] args){
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Operation e = (Operation) context.getBean("opBean");
        System.out.println("calling m...");
        System.out.println(e.m());
        System.out.println("calling k...");
        System.out.println(e.k());
    }
}
```

Output:

D. aop:around Example

Before and after invoking the real business logic methods, the AspectJ surrounding guidance is applied. Make a class with real business logic in it.

File: Operation.java

```
// Java Program to Illustrate Operation Class

package com.geeksforgeeks;

// Class
public class Operation {

    // Method 1
    public void msg()
    {
        System.out.println("msg() is invoked");
    }
    // Method 2
    public void display()
    {
        System.out.println("display() is invoked");
    }
}
```

Create a class that contains advice as an aspect.

You must supply the PreceedingJoinPoint reference to the advise function so that we may execute the proceed() method to continue the request.

File: TrackOperation.java

```
// Java Program to Illustrate TrackOperation Class

package com.geeksforgeeks;

// Importing required classes
import org.aspectj.lang.ProceedingJoinPoint;

// Class
public class TrackOperation {

    // Method
    public Object myadvice(ProceedingJoinPoint pjp)
        throws Throwable
    {
        // Display message
        System.out.println(
            "Additional Concern Before calling actual method");

        Object obj = pjp.proceed();

        // Display message
        System.out.println(
            "Additional Concern After calling actual method");

        return obj;
```

```
    }
}
```

File: applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://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 />

<bean id="opBean" class="com.geeksforgeeks.Operation">   </bean>

<bean id="trackAspect" class="com.geeksforgeeks.TrackOperation"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="trackAspect" >
     <!-- @Around -->
     <aop:pointcut id="pointCutAround"   expression="execution(*
com.geeksforgeeks.Operation.*(..))" />
     <aop:around method="myadvice" pointcut-ref="pointCutAround" />
  </aop:aspect>
</aop:config>

</beans>
```

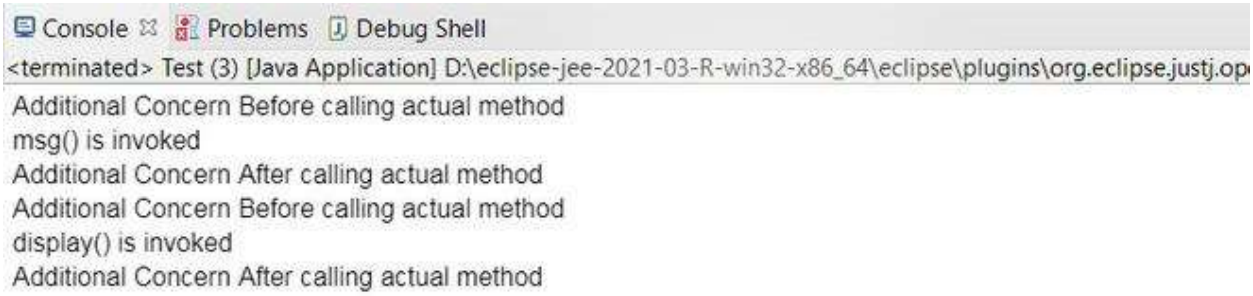
File: Test.java

Now create the Test class that calls the actual methods.

```
// Java Program to Illustrate Application class

package com.geeksforgeeks;

// Importing required classes
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

// Application class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {
```

```
        // Creating an object of ApplicationContext class
        ApplicationContext context
            = new classPathXmlApplicationContext(
                "applicationContext.xml");

        // Creating an object of Operation class
        Operation op = (Operation)context.getBean("opBean");

        op.msg();
        op.display();
    }
}
```

Output:



E. aop:after-throwing example

We may print the exception in the TrackOperation class by utilizing it after throwing advice. Let's take a look at the AspectJ AfterThrowing tip as an example.

Make a class to hold the business logic.

File: Operation.java

```
// Java Program to Illustrate Operation Class

package com.geeksforgeeks;

// Class
public class Operation {

    // Method
    public void validate(int age) throws Exception
    {
```

```
        if (age < 18) {
            throw new ArithmeticException("Not valid age");
        }
        else {
            System.out.println("Thanks for vote");
        }
    }
}
```

Create an aspect class that contains advice after it has been thrown.

We must additionally give the Throwable reference here in order to intercept the exception.

File: TrackOperation.java

```
// Java Program to Illustrate TrackOperation Class

package com.geeksforgeeks;

// Importing required classes
import org.aspectj.lang.JoinPoint;

// Class
public class TrackOperation {

    // Main driver method
    public void myadvice(JoinPoint jp, Throwable error)
    {
        // Print statements
        System.out.println("additional concern");
        System.out.println("Method Signature: "
                           + jp.getSignature());
        System.out.println("Exception is: " + error);
        System.out.println(
            "end of after throwing advice...");
    }
}
```

File: applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://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 />
<bean id="opBean" class="com.geeksforgeeks.Operation">   </bean>
<bean id="trackAspect" class="com.geeksforgeeks.TrackOperation"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="trackAspect" >
     <!-- @AfterThrowing -->
     <aop:pointcut id="pointCutAfterThrowing"    expression="execution(*
com.geeksforgeeks.Operation.*(..))" />
     <aop:after-throwing method="myadvice" throwing="error" pointcut-
ref="pointCutAfterThrowing" />
  </aop:aspect>
</aop:config>

</beans>
```

File: Test.java

Now create the Test class that calls the actual methods.

```
// Java Program to Illustrate Application Class

package com.geeksforgeeks;

// Importing required classes
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

// Main Class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of ApplicationContext
        // and Operation class
        ApplicationContext context
            = new ClassPathXmlApplicationContext(
                "applicationContext.xml");
        Operation op = (Operation)context.getBean("opBean");

        // Display message
        System.out.println("calling validate...");

        // Try block to check for exceptions
        try {
            op.validate(19);
        }

        // Catch block to handle the exceptions
```

```
        catch (Exception e) {
            System.out.println(e);
        }

        // Display message only
        System.out.println("calling validate again...");

        try {
            op.validate(11);
        }

        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Output: