Spring AOP + AspectJ annotation example

By mkyong | June 21, 2011 | Updated : August 29, 2012 | Viewed : 367,779 times

In this tutorial, we show you how to integrate AspectJ annotation with Spring AOP framework. In simple, Spring AOP + AspectJ allow you to intercept method easily.

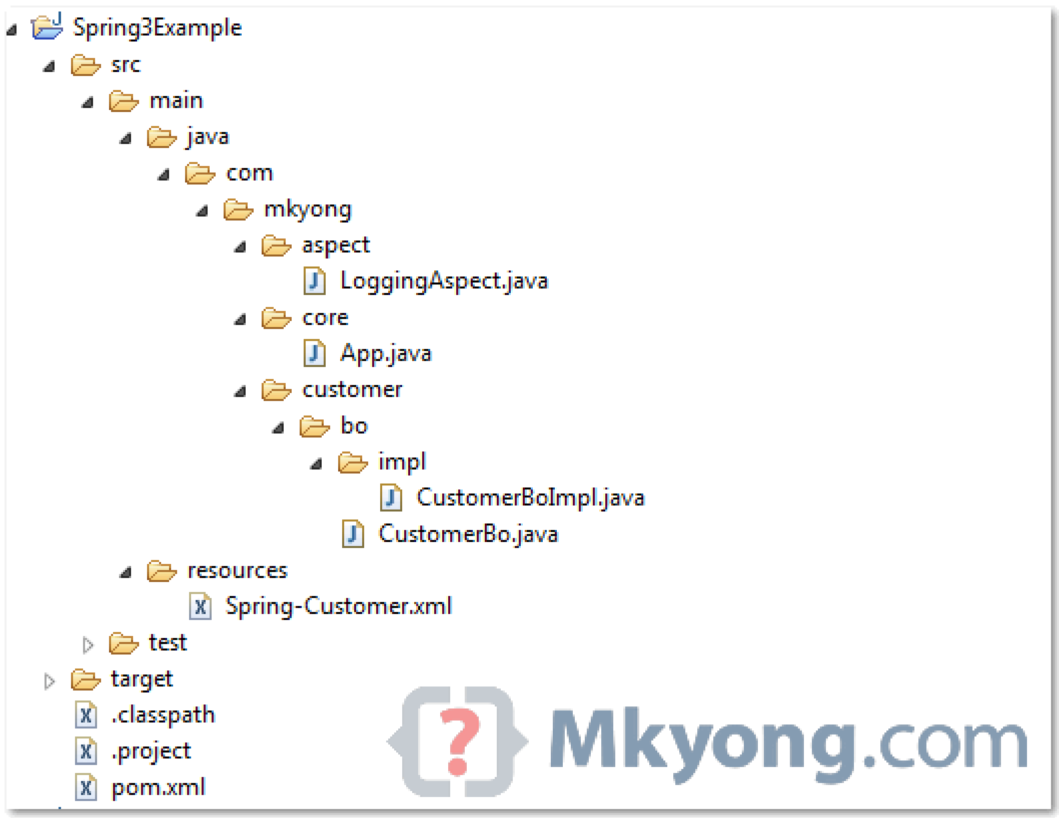
Common AspectJ annotations :

1. **@Before** – Run before the method execution
2. **@After** – Run after the method returned a result
3. **@AfterReturning** – Run after the method returned a result, intercept the returned result as well.
4. **@AfterThrowing** – Run after the method throws an exception
5. **@Around** – Run around the method execution, combine all three advices above.

Note
For Spring AOP without AspectJ support, read this build-in Spring AOP examples.

1. Directory Structure

See directory structure of this example.



```
Spring3Example
  src
    main
      java
        com
          mkyong
            aspect
              LoggingAspect.java
            core
              App.java
            customer
              bo
                impl
                  CustomerBoImpl.java
                CustomerBo.java
      resources
        Spring-Customer.xml
    test
  target
  .classpath
  .project
  pom.xml
```

2. Project Dependencies

To enable AspectJ, you need **aspectjrt.jar**, **aspectjweaver.jar**and **spring-aop.jar**. See following Maven `pom.xml` file.

AspectJ supported since Spring 2.0
This example is using Spring 3, but the AspectJ features are supported since Spring 2.0.

File : pom.xml

Markup

```
<project ...>

    <properties>
        <spring.version>3.0.5.RELEASE</spring.version>
    </properties>

    <dependencies>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <!-- Spring AOP + AspectJ -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.6.11</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.6.11</version>
        </dependency>

    </dependencies>
</project>
```

3. Spring Beans

Normal bean, with few methods, later intercept it via AspectJ annotation.

Java
```
package com.mkyong.customer.bo;

public interface CustomerBo {

    void addCustomer();

    String addCustomerReturnValue();

    void addCustomerThrowException() throws Exception;

    void addCustomerAround(String name);
}
```

Java
```
package com.mkyong.customer.bo.impl;

import com.mkyong.customer.bo.CustomerBo;

public class CustomerBoImpl implements CustomerBo {

    public void addCustomer(){
        System.out.println("addCustomer() is running ");
    }

    public String addCustomerReturnValue(){
        System.out.println("addCustomerReturnValue() is running ");
        return "abc";
    }

    public void addCustomerThrowException() throws Exception {
        System.out.println("addCustomerThrowException() is running ");
        throw new Exception("Generic Error");
    }

    public void addCustomerAround(String name){
        System.out.println("addCustomerAround() is running, args : " + name);
    }
}
```

4. Enable AspectJ

In Spring configuration file, put " `<aop:aspectj-autoproxy />` ", and define your Aspect (interceptor) and normal bean.

File : Spring-Customer.xml

Markup
```
<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 />

    <bean id="customerBo" class="com.mkyong.customer.bo.impl.CustomerBoImpl" />

    <!-- Aspect -->
    <bean id="logAspect" class="com.mkyong.aspect.LoggingAspect" />

</beans>
```

4. AspectJ @Before

In below example, the `logBefore()` method will be executed before the execution of customerBo interface, `addCustomer()` method.

Note
AspectJ "pointcuts" is used to declare which method is going to intercept, and you should refer to this Spring AOP pointcuts guide for full list of supported pointcuts expressions.

File : LoggingAspect.java

Java
```
package com.mkyong.aspect;

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

@Aspect
public class LoggingAspect {

    @Before("execution(* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")
```

```java
    public void logBefore(JoinPoint joinPoint) {

        System.out.println("logBefore() is running!");
        System.out.println("hijacked : " + joinPoint.getSignature().getName());
        System.out.println("******");
    }

}
```

Run it

```java
    CustomerBo customer = (CustomerBo) appContext.getBean("customerBo");
    customer.addCustomer();
```

Output

```
logBefore() is running!
hijacked : addCustomer
******
addCustomer() is running
```

5. AspectJ @After

In below example, the `logAfter()` method will be executed after the execution of customerBo interface, `addCustomer()` method.

File : LoggingAspect.java

```java
package com.mkyong.aspect;

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

@Aspect
public class LoggingAspect {

    @After("execution(* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")
    public void logAfter(JoinPoint joinPoint) {

        System.out.println("logAfter() is running!");
        System.out.println("hijacked : " + joinPoint.getSignature().getName());
        System.out.println("******");

    }

}
```

Run it

Java
```
    CustomerBo customer = (CustomerBo) appContext.getBean("customerBo");
    customer.addCustomer();
```

Output

Java
```
addCustomer() is running
logAfter() is running!
hijacked : addCustomer
******
```

6. AspectJ @AfterReturning

In below example, the `logAfterReturning()` method will be executed after the execution of customerBo interface, `addCustomerReturnValue()` method. In addition, you can intercept the returned value with the "**returning**" attribute.

To intercept returned value, the value of the "returning" attribute (result) need to be same with the method parameter (result).

File : LoggingAspect.java

Java
```
package com.mkyong.aspect;

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

@Aspect
public class LoggingAspect {

   @AfterReturning(
      pointcut = "execution(* com.mkyong.customer.bo.CustomerBo.addCustomerReturnValue(..))",
      returning= "result")
   public void logAfterReturning(JoinPoint joinPoint, Object result) {

    System.out.println("logAfterReturning() is running!");
    System.out.println("hijacked : " + joinPoint.getSignature().getName());
    System.out.println("Method returned value is : " + result);
    System.out.println("******");

   }
}
```

Run it

Java
```java
    CustomerBo customer = (CustomerBo) appContext.getBean("customerBo");
    customer.addCustomerReturnValue();
```

Output

Java
```
addCustomerReturnValue() is running
logAfterReturning() is running!
hijacked : addCustomerReturnValue
Method returned value is : abc
******
```

7. AspectJ @AfterReturning

In below example, the `logAfterThrowing()` method will be executed if the customerBo interface, `addCustomerThrowException()` method is throwing an exception.

File : LoggingAspect.java

Java
```java
package com.mkyong.aspect;

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

@Aspect
public class LoggingAspect {

   @AfterThrowing(
      pointcut = "execution(* com.mkyong.customer.bo.CustomerBo.addCustomerThrowException(..))",
      throwing= "error")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {

    System.out.println("logAfterThrowing() is running!");
    System.out.println("hijacked : " + joinPoint.getSignature().getName());
    System.out.println("Exception : " + error);
    System.out.println("******");

    }
}
```

Run it

Java

```
    CustomerBo customer = (CustomerBo) appContext.getBean("customerBo");
    customer.addCustomerThrowException();
```

Output

Java

```
addCustomerThrowException() is running
logAfterThrowing() is running!
hijacked : addCustomerThrowException
Exception : java.lang.Exception: Generic Error
******
Exception in thread "main" java.lang.Exception: Generic Error
    //...
```

8. AspectJ @Around

In below example, the `logAround()` method will be executed before the customerBo interface, `addCustomerAround()` method, and you have to define the " `joinPoint.proceed();` " to control when should the interceptor return the control to the original `addCustomerAround()` method.

File : LoggingAspect.java

Java

```
package com.mkyong.aspect;

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

@Aspect
public class LoggingAspect {

   @Around("execution(* com.mkyong.customer.bo.CustomerBo.addCustomerAround(..))")
   public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {

    System.out.println("logAround() is running!");
    System.out.println("hijacked method : " + joinPoint.getSignature().getName());
    System.out.println("hijacked arguments : " + Arrays.toString(joinPoint.getArgs()));

    System.out.println("Around before is running!");
    joinPoint.proceed(); //continue on the intercepted method
    System.out.println("Around after is running!");

    System.out.println("******");

   }

}
```

Run it

```java
    CustomerBo customer = (CustomerBo) appContext.getBean("customerBo");
    customer.addCustomerAround("mkyong");
```

Output

```java
logAround() is running!
hijacked method : addCustomerAround
hijacked arguments : [mkyong]
Around before is running!
addCustomerAround() is running, args : mkyong
Around after is running!
******
```

Conclusion

It's always recommended to apply the least power AsjectJ annotation. It's rather long article about AspectJ in Spring. for further explanations and examples, please visit the reference links below.

Anti annotation or using JDK 1.4 ?
No worry, AspectJ supported XML configuration also, read this Spring AOP + AspectJ XML example.