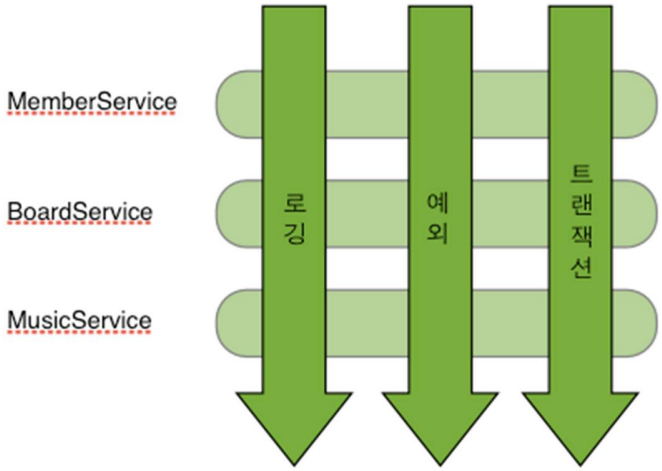
st05.스프링은 AOP 를 지원한다.docx

목차

1. AOP(Aspect Oriented Programming) 란? 2
   - 공통관심사항 (cross-cutting concern) 2
2. AOP 용어 4
   - JoinPoint 4
   - Pointcut 4
   - Advice 5
     Advice 타입 5
   - Aspect 6
   - Target 6
   - proxy 6
3. AOP Weaving 7
   - 컴파일시 Weaving → 컴파일시 코드 삽입 7
   - 클래스 로딩시 Weaving → 로딩시 코드 삽입 7
   - 런타임시 Weaving → 런타임 Proxy 생성 7
4. Spring AOP 냐? AspectJ 냐? 8
5. PointCut 정의 예제 9
6. 스프링에서의 AOP 11
   - 스프링의 AOP 구현 방식 11
   - XML 스키마를 이용한 AOP 설정 11
   - @Aspect 어노테이션 기반의 AOP 구현 15
7. Intercept static methods using AOP 15
8. Reference 16
9. Load-time weaving with AspectJ in the Spring Framework 17

1. AOP(Aspect Oriented Programming) 란?

- 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법

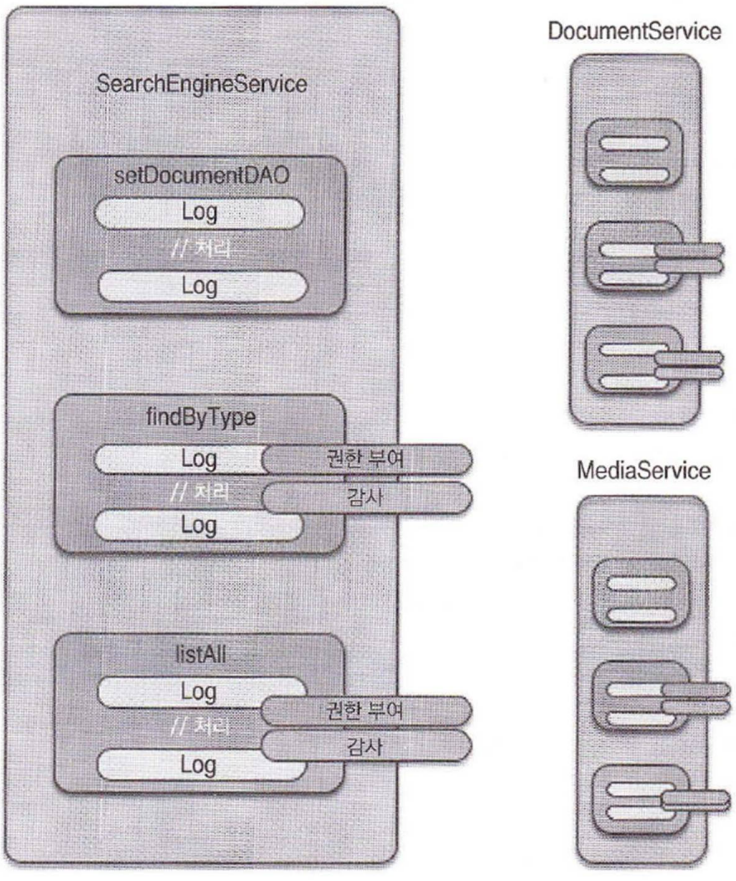


- 문제를 해결하기 위한 **핵심 관심 사항(비즈니스 로직)**과 전체에 적용되는 **공통 관심 사항**을 기준으로 프로그래밍 함으로써 공통 모듈을 여러 코드에 쉽게 적용

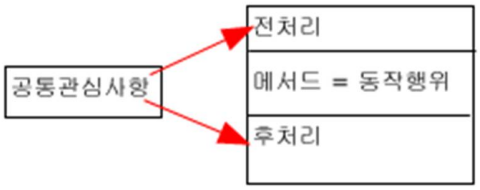
- core concern(핵심 관심 사항) - 비즈니스 로직
- cross-cutting concern(공통관심사항) - 로깅, 보안, 감사, 트랜잭션, 예외, 에러처리, 동기화, 분산

- **공통관심사항 (cross-cutting concern)**

어플리케이션에서 로깅과 같은 기본적인 기능에서부터 트랙잭션이나 보안과 같은 기능에 이르기까지 어플리케이션 전반에 걸쳐 적용되는 공통 기능



- 로깅(Logging)
- 보안체크
- 감사체크
- 트랜잭션 관리

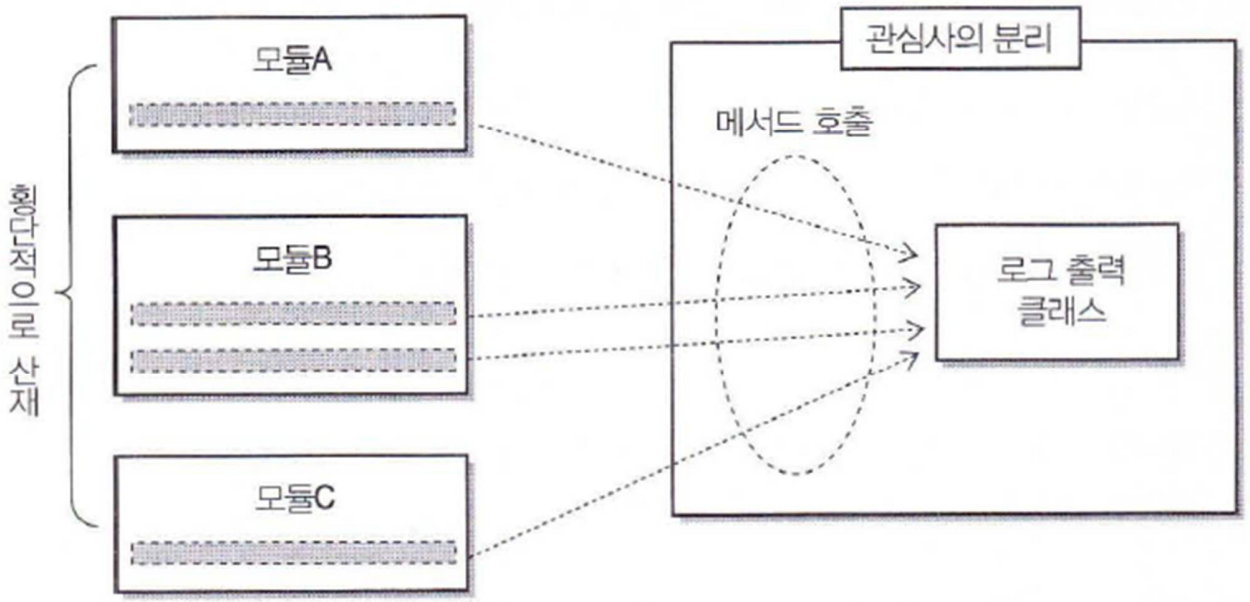


스프링의 AOP (공통관심사항)

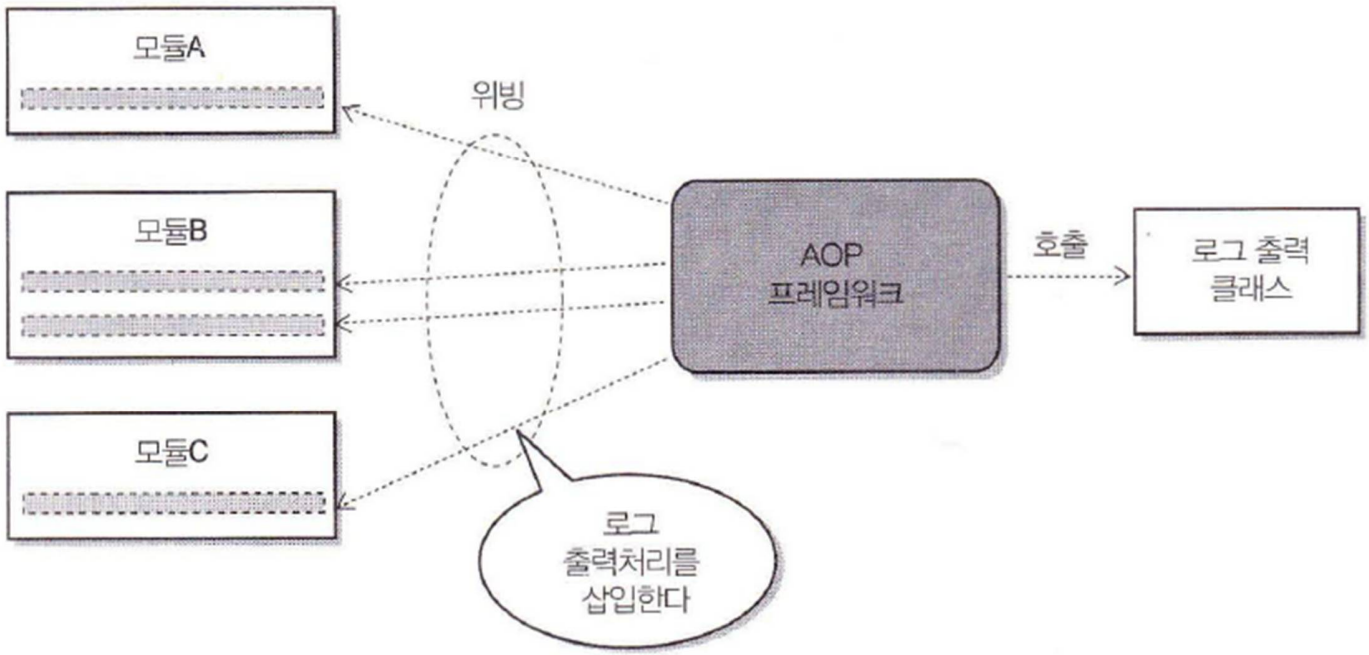
스트럿츠에서의 인터셉터와 비슷한기능

하지만 더욱 확장된 기능을 가짐
(스프링에서도 인터셉터를 만들수는있음)

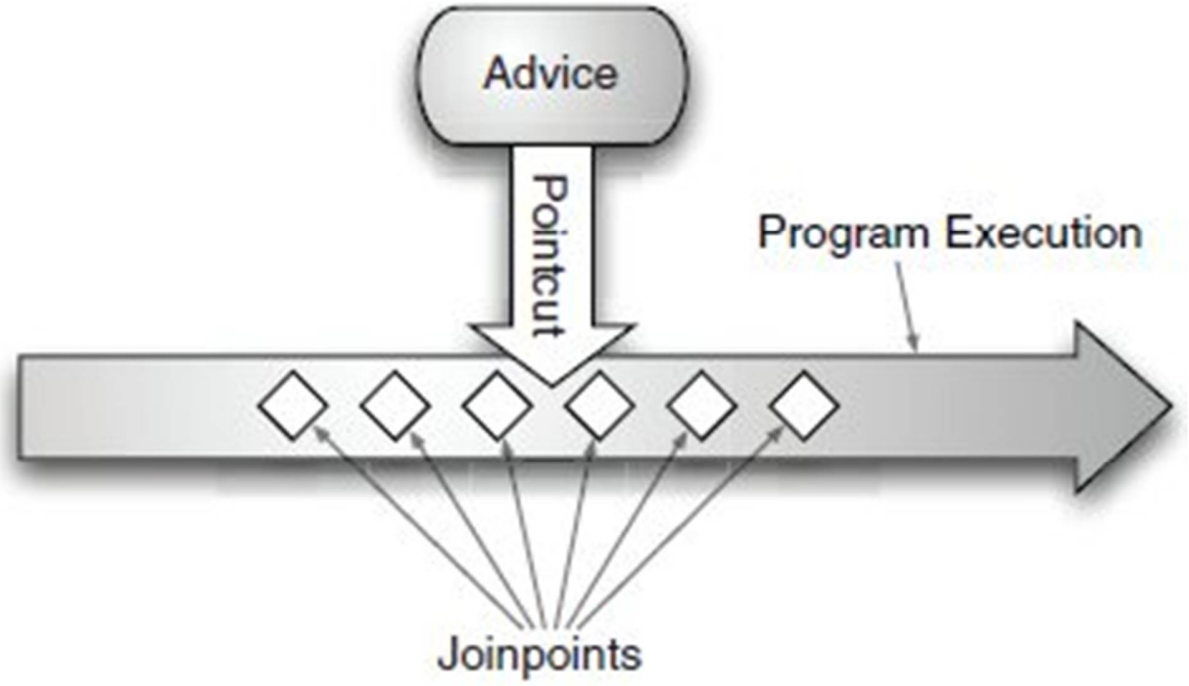
■ 횡단적으로 산재하는 기능 호출



■ AOP에서 횡단적 관심사의 분리와 위빙



2. AOP 용어



JointPoint 의 PointCut 의 Advice 를 Aspect 해~~

JoinPoint	동작 장소. 어디에. 실행 지점
Advice(Advisor)	동작 코드. 무엇을. 실행 액션
PointCut	동작 조건. 어떻게. 결합 패턴(표현식 언어)
Aspect	Pointcut 과 Advice 의 집합

- **JoinPoint**

Advice 를 적용 가능한 지점을 의미한다. 인스턴스 생성 시점', '메소드 호출 시점', '예외 발생 시점'과 같이 어플리케이션을 실행할 때 특정 작업이 시작되는 시점을 '조인포인트'라고 한다. 구체적으로는 메서드 호출이나 예외발생이라는 포인트를 Joinpoint 라고 한다.

- **Pointcut**

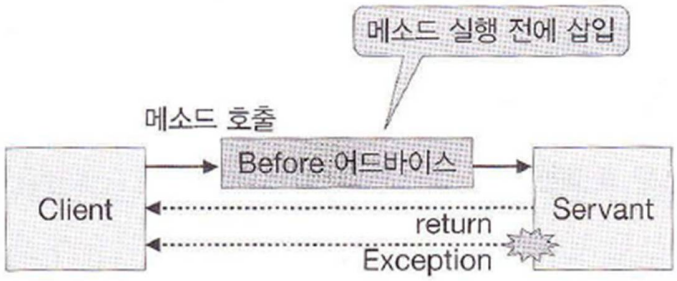
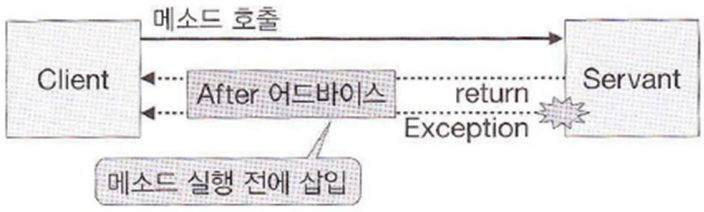
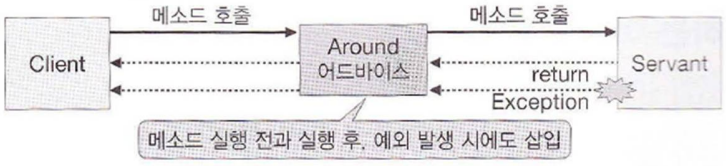
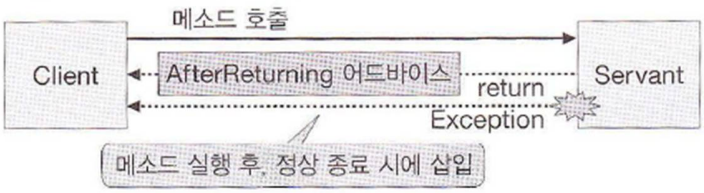
Joinpoint 의 부분 집합으로서 실제로 Advice 가 적용되는 Jointpoint 를 나타낸다. 스프링에서는 정규 표현식이나 AspectJ 문법을 이용하여 Pointcut 을 정의할 수 있다.
하나 또는 복수의 Jointpoint 를 하나로 묶은 것을 Pointcut 이라고 한다. Advice 의 위빙 정의는 Pointcut 을 대상으로 설정한다. 하나의 Pointcut 에는 복수 Advice 를 연결할 수 있다. 반대로 하나의 Advice 를 복수 Pointcut 에 연결하는 것도 가능하다.

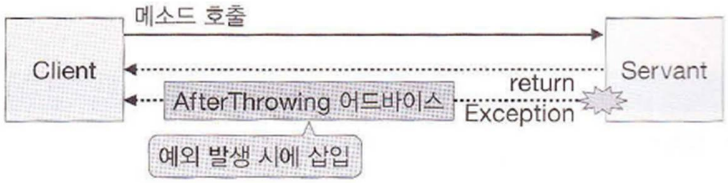
Pointcut(교차점)은 JoinPoint(결합점)들을 선택하고 결합점의 환경정보를 수집하는 프로그램의 구조물이다.

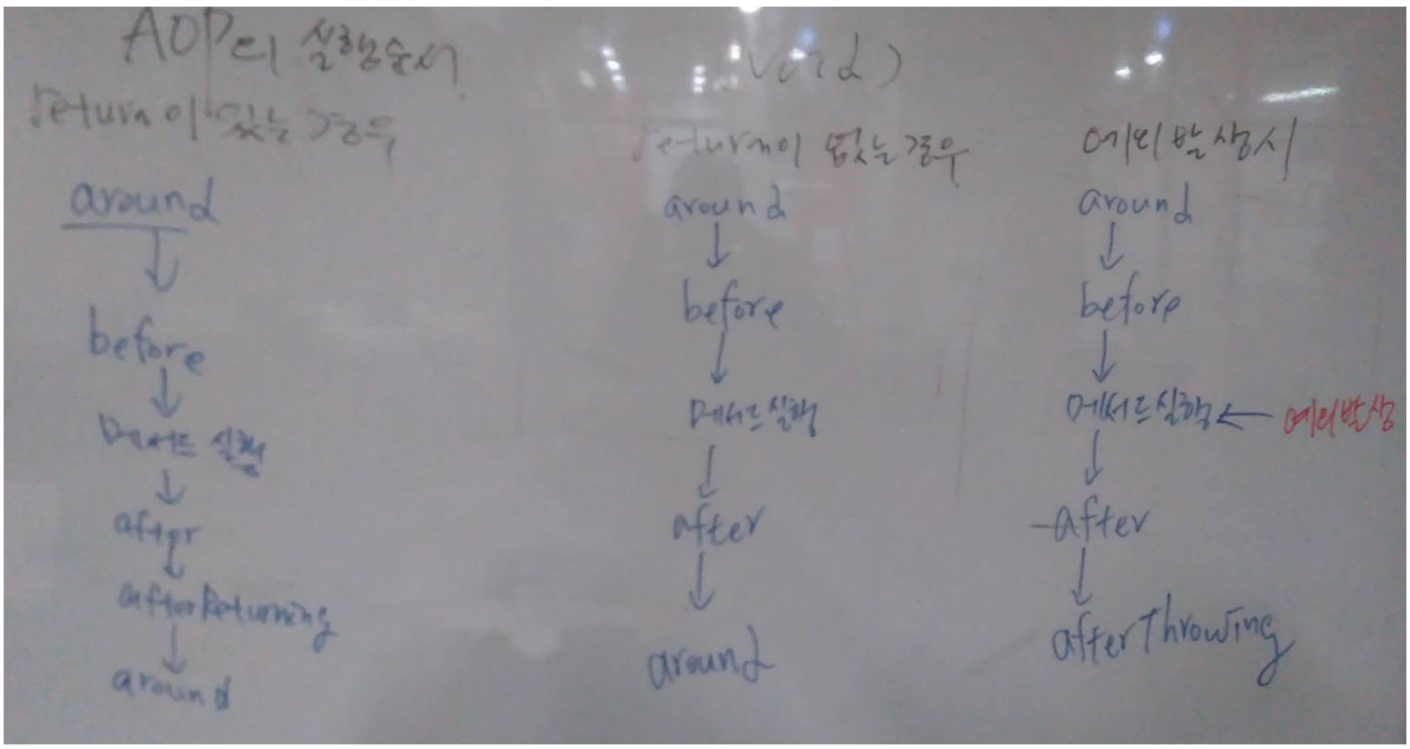
- **Advice**

언제 공통 관심 기능을 핵심 로직에 적용할 지를 정의하고 있다.
Advice 를 어디에서 위빙하는 지는 PointCut 이라는 단위로 정의한다.

Advice 타입

Advice 타입	설명	Annotation
Before	Joinpoint 앞에서 실행할 Advice org.springframework.aop.MethodBeforeAdvice 	@Before
After	Joinpoint 뒤에서 실행할 Advice org.springframework.aop.AfterAdvice 	@After
Around	Joinpoint 앞과 뒤에서 실행되는 Adcvice org.springframework.aop.MethodInterceptor 	@Around
After Returning	Jointpoint 가 정상 종료한 다음에 실행되는 Advice org.springframework.aop.AfterReturningAdvice 	@AfterReturning
After Throwing	Jointpoint 에서 예외가 발생했을 때 실행되는 Advice	@AfterThrowing

	org.springframework.aop.ThrowsAdvice  	
Introduction	클래스에 인터페이스와 구현을 추가하는 특수한 Advice org.springframework.aop.IntroductionInterceptor	



- **Aspect**

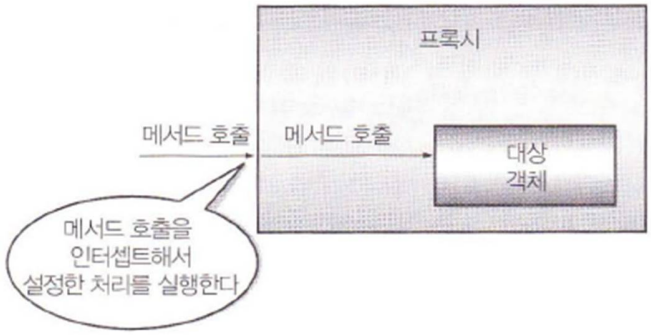
여러 객체에 공통으로 적용되는 공통 관심 사항을 Aspect 라고 한다. 로깅, 트랜잭션이나 보안 등이 Aspect 의 좋은 예이다.
Aspect 는 AOP 의 중심 단위로 Advice 와 PointCut 을 합친 것이다. Aspcet 를 Advisor 라고도 한다.

- **Target**

핵심 로직을 구현하는 클래스를 말한다. 충고를 받는 클래스를 대상(target)이라고 한다. 대상은 여러분이 작성한 클래스는 물론, 별도의 기능을 추가하고자 하는 써드 파티 클래스가 될 수 있다.

- **proxy**

대상 객체에 Advice 가 적용된 후 생성된 객체



3. AOP Weaving

Weaving 이란 코드가 삽입되는 행위를 Weaving 이라고 한다.

Weaving 방식에는

- 컴파일시 코드 Weaving
- 로딩시 코드 Weaving
- 런타임시 코드 Weaving

- **컴파일시 Weaving → 컴파일시 코드 삽입**

대표적인 AOP 지원툴인 **AspectJ** 가 사용하는 방식이다. 컴파일시 핵심로직을 구현한 소스에 알맞은 위치에 공통 기능 코드를 삽입하게 된다.

- **클래스 로딩시 Weaving → 로딩시 코드 삽입**

JVM 이 클래스를 로딩할 때 클래스 정보를 변경할수 있는 에이전트를 제공함으로써 클래스의 바이너리 정보를 변경하여 알맞은 위치에 공통 코드를 삽입한 새로운 클래스 바이너리 코드를 사용하도록 한다. 즉 원본 클래스 파일은 변경하지 않고 클래스를 로딩할 때 JVM 이 변경된 바이트 코드를 사용한다. **AspectWerkz** 가 이에 해당한다.

- **런타임시 Weaving → 런타임 Proxy 생성**

런타임시 Weaving 은 프록시를 이용해 AOP 를 적용한다. 즉 핵심 로직을 구현한 객체에 직접 접근하지 않고 중간에 공통코드가 적용된 프록시를 생성하여 프록시를 통해 핵심 로직에 접근하게 된다. **Spring AOP 가 이에 해당한다**.

4. Spring AOP 냐? AspectJ 냐?

작업 할 수 있는 가장 간단한 것을 사용해라. 스프링 AOP 는 개발이나 빌드 프로세스에 AspectJ 컴파일러/위버(weaver)를 도입해야 하는 요구사항이 없으므로 완전한 AspectJ 를 사용하는 것보다 간단하다. 스프링 빈에서 작업의 실행을 어드바이즈 하는 것이 필요한 것의 전부라면 스프링 AOP 가 좋은 선택이다. 스프링 컨테이너가 관리하지 않는 객체(보통 도메인 객체 같은)를 어드바이즈 해야 한다면 AspectJ 를 사용해야 할 것이다. 간단한 메서들 실행외에 조인포인트를 어드바이즈해야 한다면 마찬가지로 AspectJ 를 사용해야 할 것이다.(예를 들면 필드를 가져오거나 조인포인트를 설정하는 등)

AspectJ 를 사용하는 경우 AspectJ 언어 문법("code style"이라고도 알려진)과 @AspectJ 어노테이션 방식의 선택권이 있다. 알기 쉽게 자바 5 이상을 사용하지 않는다면 code style 을 사용해야 한다. 설계상 관점이 커다란 역할을 한고 AspectJ Development Tools (AJDT) 이클립스 플러그인을 사용할 수 있다면 AspectJ 언어의 문법을 사용하는 것이 더 바람직하다. AspectJ 언어가 관점을 작성하기 위한 목적으로 설계된 언어이기 때문에 더 깔끔하고 간단하다. 이클립스를 사용하지 않거나 어플리케이션에서 주요 역할을 하지 않는 약간의 관점만 가지고 있다면 IDE 에서 일반적인 자바 컴파일과 @AspectJ 방식을 사용하고 빌드 스크립트에 관점을 위빙하는 단계를 추가하는 것을 고려해라.

5. PointCut 정의 예제

Spring AOP 에서 자주 사용되는 PointCut 표현식의 예를 살펴본다.

PointCut	선택된 JoinPoints
execution(public * *(..))	public 메소드 실행
execution(* set*(..))	이름이 set 으로 시작하는 모든 메소드명 실행
execution(* com.xyz.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.xyz.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.xyz.service..*.*(..))	service 패키지와 하위 패키지의 모든 메소드 실행
within(com.xyz.service.*)	service 패키지 내의 모든 결합점
within(com.xyz.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
target(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable 인 모든 결합점

Pointcut	선택된 Joinpoints
bean(accountRepository)	"accountRepository" 빈
!bean(accountRepository)	"accountRepository" 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 "Repository"로 끝나는 모든 빈
bean(accounting/*)	이름이 "accounting/"로 시작하는 모든 빈
bean(*dataSource) || bean(*DataSource)	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈

Pointcut	선택된 Joinpoints
@target(org.springframework.transaction.annotation.Transactional)	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
@within(org.springframework.transaction.annotation.Transactional)	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
@annotation(org.springframework.transaction.annotation.Transactional)	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
@args(com.xyz.security.Classified)	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점

6. 스프링에서의 AOP

스프링은 자체적으로 프록시 기반의 AOP 를 지원한다. 따라서 스프링 AOP 는 메서드 호출 Joinpoint 만을 지원하고 필드값 변경 같은 Joinpoint 를 사용하기 위해서는 AspectJ 와 같이 풍부한 기능을 지원하는 AOP 도구를 사용해야 한다.

- **스프링의 AOP 구현 방식**

- XML 스키마 기반의 AOP 구현
- @Aspect 어노테이션 기반의 AOP 구현
- 스프링 API 를 이용한 AOP 구현 (많이 사용하지는 않음)

- **XML 스키마를 이용한 AOP 설정**

<aop:config />	aop 설정을 포함.
<aop:aspect />	Aspect 를 설정.
<aop:pointcut />	Pointcut 을 설정.
<aop:before />	메서드 실행전에 적용되는 Advice 를 정의
<aop:after-returning />	메서드가 정상적으로 실행된 후에 적용되는 Advice 를 정의
<aop:after-throwing />	예외를 발생시킬 때 적용되는 Advice 를 정의 (catch 블록와 유사)
<aop:after />	예외 여부와 상관없는 Advice 를 정의한다 (finally 블록과 유사)
<aop:around />	메서드 호출 이전, 이후, 예외 발생 등 모든 시점에서 적용 가능한 Advice 를 정의

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context=http://www.springframework.org/schema/context

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

    xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd

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


  <!--핵심기능 -->
    <beans:bean name="myBean" class="com.lecture.spring.bean.MyBean" />

    <!-- 공통관심사항 -->
    <beans:bean id="aspectBean" class="com.lecture.spring.aspect.MyAspect" />

    <aop:config>
        <aop:aspect id="myAspect" ref="aspectBean">
            <aop:before pointcut="execution(* print*(..))" method="beforeMessage" />
            <aop:after-returning pointcut="execution(* print*(..))"
method="afterMessage" />
        </aop:aspect>
    </aop:config>

</ beans: beans>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context=http://www.springframework.org/schema/context

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

    xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd

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

    <beans:bean id="myFirstAOP" class="aop01.aop.FirstAspect" />

    <aop:config>
        <aop:aspect id="myAspect" ref="myFirstAOP">

            <!--  get 으로 시작하는 모든 메서드에 적용하겠다.  -->
            <aop:pointcut id="pc" expression="execution( * get*(*))" />

            <aop:after           method="after"           pointcut-ref="pc" />
            <aop:after-returning method="afterReturning"  pointcut-ref="pc"
returning="product" />
            <aop:after-throwing  method="afterThrowing"   pointcut-ref="pc"
throwing="e"        />
            <aop:around          method="around"          pointcut-ref="pc" />
            <aop:before          method="before"          pointcut-ref="pc" />
        </aop:aspect>
    </aop:config>

</beans:beans>
```

```
public class ProfilingAdvice {
    public Object trace(ProceedingJoinPoint joinPoint) throws Throwable{

        // 전처리
        String signatureString = joinPoint.getSignature().toShortString();
        System.out.println(signatureString + "시작");
        long start = System.currentTimeMillis();



        try{
            Object result = joinPoint.proceed();
            return result;
        }finally{

                // 후처리
            long finish = System.currentTimeMillis();
            System.out.println(signatureString + "종료");
            System.out.println(signatureString + "실행 시간 : " + (finish - start) + "ms");

        }
    }
}
```

- **@Aspect 어노테이션 기반의 AOP 구현**

@Aspect 어노테이션을 이용해서 Aspect 클래스를 구현한다 .이때 Aspect 클래스는 Advice 를 구현한 메서드와 Pointcut 을 포함한다.

Pointcut ： Joinpoint 의 부분집합(그룹화) 실제로 Advice 가 Joinpoint 적용된 스프링 정규표현식이나 AspectJ 의 문법을 이용하여 Pointcut 을 정의 할 수있다.

Joinpoint : Advice 를 적용 가능한 지점을 의미 (메서드호출, 필드값 변경 등)
Weaving ：Advice 를 핵심 로직 코드에 적용하는것

7. Intercept static methods using AOP

http://stackoverflow.com/questions/13744677/set-an-aspectj-advise-for-static-method
http://docs.spring.io/spring/docs/3.0.0.RC2/reference/html/ch07s08.html
http://java-demos.blogspot.kr/2014/04/static-pointcuts-example-in-spring-aop.html

8. Reference

http://devbox.tistory.com/entry/spring-AOP-용어-설명

http://sararing.tistory.com/81

http://blog.naver.com/minsoub/60114310691

http://www.javajigi.net/pages/viewpage.action?pageId=1080

9. Load-time weaving with AspectJ in the Spring Framework

Load-time weaving (LTW) refers to the process of weaving AspectJ aspects into an application's class files as they are being loaded into the Java virtual machine (JVM). The focus of this section is on configuring and using LTW in the specific context of the Spring Framework: this section is not an introduction to LTW though. For full details on the specifics of LTW and configuring LTW with just AspectJ (with Spring not being involved at all), see the LTW section of the AspectJ Development Environment Guide.

The value-add that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected using a Java (5+) agent, which is switched on by specifying a VM argument when starting up a JVM. It is thus a JVM-wide setting, which may be fine in some situations, but often is a little too coarse. Spring-enabled LTW enables you to switch on LTW on a *per-ClassLoader* basis, which obviously is more fine-grained and which can make more sense in a 'single-JVM-multiple-application' environment (such as is found in a typical application server environment).

Further, in certain environments, this support enables load-time weaving *without making any modifications to the application server's launch script* that will be needed to add -javaagent:path/to/aspectjweaver.jar or (as we describe later in this section) -javaagent:path/to/spring-agent.jar. Developers simply modify one or more files that form the application context to enable load-time weaving instead of relying on administrators who typically are in charge of the deployment configuration such as the launch script.

Now that the sales pitch is over, let us first walk through a quick example of AspectJ LTW using Spring, followed by detailed specifics about elements introduced in the following example. For a complete example, please see the Petclinic sample application.

7.8.4.1 A first example

Let us assume that you are an application developer who has been tasked with diagnosing the cause of some performance problems in a system. Rather than break out a profiling tool, what we are going to do is switch on a simple profiling aspect

that will enable us to very quickly get some performance metrics, so that we can then apply a finer-grained profiling tool to that specific area immediately afterwards.

Here is the profiling aspect. Nothing too fancy, just a quick-and-dirty time-based profiler, using the @AspectJ-style of aspect declaration.

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

   @Around("methodsToBeProfiled()")
   public Object profile(ProceedingJoinPoint pjp) throws Throwable {
      StopWatch sw = new StopWatch(getClass().getSimpleName());
      try {
         sw.start(pjp.getSignature().getName());
         return pjp.proceed();
      } finally {
         sw.stop();
         System.out.println(sw.prettyPrint());
      }
   }

   @Pointcut("execution(public * foo..*.*(..))")
   public void methodsToBeProfiled(){}
}
```

We will also need to create an 'META-INF/aop.xml' file, to inform the AspectJ weaver that we want to weave our ProfilingAspect into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called ' META-INF/aop.xml' is standard AspectJ.

```
<!DOCTYPE aspectj PUBLIC
      "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
```

```
    <weaver>

        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>

    </weaver>

    <aspects>

        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>

    </aspects>

</aspectj>
```

Now to the Spring-specific portion of the configuration. We need to configure a LoadTimeWeaver (all explained later, just take it on trust for now). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more 'META-INF/aop.xml' files into the classes in your application. The good thing is that it does not require a lot of configuration, as can be seen below (there are some more options that you can specify, but these are detailed later).

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <!-- a service object; we will be profiling its methods -->
    <bean id="entitlementCalculationService"
        class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>

</beans>
```

Now that all the required artifacts are in place - the aspect, the 'META-INF/aop.xml' file, and the Spring configuration -, let us create a simple driver class with a main(..) method to demonstrate the LTW in action.

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

   public static void main(String[] args) {

      ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml", Main.class);

      EntitlementCalculationService entitlementCalculationService
         = (EntitlementCalculationService) ctx.getBean("entitlementCalculationService");

      // the profiling aspect is 'woven' around this method execution
      entitlementCalculationService.calculateEntitlement();
   }
}
```

There is one last thing to do. The introduction to this section did say that one could switch on LTW selectively on a per-ClassLoader basis with Spring, and this is true. However, just for this example, we are going to use a Java agent (supplied with Spring) to switch on the LTW. This is the command line we will use to run the above Main class:

```
java -javaagent:C:/projects/foo/lib/global/spring-agent.jar foo.Main
```

The '-javaagent' is a Java 5+ flag for specifying and enabling agents to instrument programs running on the JVM. The Spring Framework ships with such an agent, the InstrumentationSavingAgent, which is packaged in the spring-agent.jar that was supplied as the value of the -javaagent argument in the above example.

The output from the execution of the Main program will look something like that below. (I have introduced a Thread.sleep(..) statement into the calculateEntitlement() implementation so that the profiler actually captures something other than 0 milliseconds - the 01234 milliseconds is *not* an overhead introduced by the AOP :))

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
------ ----- ----------------------------
ms     %     Task name
------ ----- ----------------------------
01234  100%  calculateEntitlement
```

Since this LTW is effected using full-blown AspectJ, we are not just limited to advising Spring beans; the following slight variation on the Main program will yield the same result.

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {

        new ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService =
            new StubEntitlementCalculationService();

        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

Notice how in the above program we are simply bootstrapping the Spring container, and then creating a new instance of the StubEntitlementCalculationService totally outside the context of Spring... the profiling advice still gets woven in.

The example admittedly is simplistic... however the basics of the LTW support in Spring have all been introduced in the above example, and the rest of this section will explain the 'why' behind each bit of configuration and usage in detail.

Note

The ProfilingAspect used in this example may be basic, but it is quite useful. It is a nice example of a development-time aspect that developers can use during development (of course), and then quite easily exclude from builds of the application being deployed into UAT or production.

7.8.4.2 Aspects

The aspects that you use in LTW have to be AspectJ aspects. They can be written in either the AspectJ language itself or you can write your aspects in the @AspectJ-style. The latter option is of course only an option if you are using Java 5+, but it does mean that your aspects are then both valid AspectJ *and* Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the classpath.

7.8.4.3 'META-INF/aop.xml'

The AspectJ LTW infrastructure is configured using one or more 'META-INF/aop.xml' files, that are on the Java classpath (either directly, or more typically in jar files).

The structure and contents of this file is detailed in the main AspectJ reference documentation, and the interested reader is referred to that resource. (I appreciate that this section is brief, but the 'aop.xml' file is 100% AspectJ - there is no Spring-specific information or semantics that apply to it, and so there is no extra value that I can contribute either as a result), so rather than rehash the quite satisfactory section that the AspectJ developers wrote, I am just directing you there.)

7.8.4.4 Required libraries (JARS)

At a minimum you will need the following libraries to use the Spring Framework's support for AspectJ LTW:

1. spring.jar (version 2.5 or later)
2. aspectjrt.jar (version 1.5 or later)
3. aspectjweaver.jar (version 1.5 or later)

If you are using the Spring-provided agent to enable instrumentation, you will also need:

1. spring-agent.jar

7.8.4.5 Spring configuration

The key component in Spring's LTW support is the LoadTimeWeaver interface (in the org.springframework.instrument.classloading package), and the numerous implementations of it that ship with the Spring distribution. A LoadTimeWeaver is

responsible for adding one or more java.lang.instrument.ClassFileTransformers to a ClassLoader at runtime, which opens the door to all manner of interesting applications, one of which happens to be the LTW of aspects.

Tip

If you are unfamiliar with the idea of runtime class file transformation, you are encouraged to read the Javadoc API documentation for the java.lang.instrument package before continuing. This is not a huge chore because there is - rather annoyingly - precious little documentation there... the key interfaces and classes will at least be laid out in front of you for reference as you read through this section.

Configuring a LoadTimeWeaver using XML for a particular ApplicationContext can be as easy as adding one line. (Please note that you almost certainly will need to be using an ApplicationContext as your Spring container - typically a BeanFactory will not be enough because the LTW support makes use of BeanFactoryPostProcessors.)

To enable the Spring Framework's LTW support, you need to configure a LoadTimeWeaver, which typically is done using the <context:load-time-weaver/> element. Find below a valid <context:load-time-weaver/> definition that uses default settings.

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:load-time-weaver/>

</beans>
```

The above <context:load-time-weaver/> bean definition will define and register a number of LTW-specific infrastructure beans for you automatically, such as a LoadTimeWeaver and an AspectJWeavingEnabler. Notice how the <context:load-time-weaver/> is defined in the 'context' namespace; note also that the referenced XML Schema file is only available in versions of Spring 2.5 and later.

What the above configuration does is define and register a default LoadTimeWeaver bean for you. The default LoadTimeWeaver is the DefaultContextLoadTimeWeaver class, which attempts to decorate an automatically detected LoadTimeWeaver: the exact type of LoadTimeWeaver that will be 'automatically detected' is dependent upon your runtime environment (summarised in the following table).

Table 7.1. DefaultContextLoadTimeWeaver LoadTimeWeavers

Runtime Environment	LoadTimeWeaver implementation
Running in BEA's Weblogic 10	WebLogicLoadTimeWeaver
Running in Oracle's OC4J	OC4JLoadTimeWeaver
Running in GlassFish	GlassFishLoadTimeWeaver
JVM started with Spring InstrumentationSavingAgent *(java -javaagent:path/to/spring-agent.jar)*	InstrumentationLoadTimeWeaver
Fallback, expecting the underlying ClassLoader to follow common conventions (e.g. applicable to TomcatInstrumentableClassLoader and to Resin)	ReflectiveLoadTimeWeaver

Note that these are just the LoadTimeWeavers that are autodetected when using the DefaultContextLoadTimeWeaver: it is of course possible to specify exactly which LoadTimeWeaver implementation that you wish to use by specifying the fully-qualified classname as the value of the 'weaver-class' attribute of the <context:load-time-weaver/> element. Find below an example of doing just that:

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:load-time-weaver
      weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
```

```
</beans>
```

The LoadTimeWeaver that is defined and registered by the <context:load-time-weaver/> element can be later retrieved from the Spring container using the well-known name 'loadTimeWeaver'. Remember that theLoadTimeWeaver exists just as a mechanism for Spring's LTW infrastructure to add one or more ClassFileTransformers. The actual ClassFileTransformer that does the LTW is the ClassPreProcessorAgentAdapter (from theorg.aspectj.weaver.loadtime package) class. See the class-level Javadoc for the ClassPreProcessorAgentAdapter class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this section.

There is one final attribute of the <context:load-time-weaver/> left to discuss: the 'aspectj-weaving' attribute. This is a simple attribute that controls whether LTW is enabled or not, it is as simple as that. It accepts one of three possible values, summarised below, with the default value if the attribute is not present being ' autodetect'

Table 7.2. 'aspectj-weaving' attribute values

Attribute Value	Explanation
on	AspectJ weaving is on, and aspects will be woven at load-time as appropriate.
off	LTW is off... no aspect will be woven at load-time.
autodetect	If the Spring LTW infrastructure can find at least one 'META-INF/aop.xml' file, then AspectJ weaving is on, else it is off. This is the default value.

7.8.4.6 Environment-specific configuration

This last section contains any additional settings and configuration that you will need when using Spring's LTW support in environments such as application servers and web containers.

Generic Java applications

You may enable Spring's support for LTW in any Java application (standalone as well as application server based) through the use of the Spring-provided instrumentation agent. To do so, start the VM by by specifying the -javaagent:path/to/spring-agent.jar option. Note that this requires modification of the VM launch script which

may prevent you from using this in application server environments (depending on your operation policies).

Tomcat

For web applications deployed onto Apache Tomcat 5.0 and above, Spring provides a TomcatInstrumentableClassLoader to be registered as the web app class loader. The required Tomcat setup looks as follows, to be included either in Tomcat's central server.xml file or in an application-specific META-INF/context.xml file within the WAR root. Spring's spring-tomcat-weaver.jar needs to be included in Tomcat's common lib directory in order to make this setup work.

```
<Context path="/myWebApp" docBase="/my/webApp/location">
  <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"
        useSystemClassLoaderAsParent="false"/>
</Context>
```

Note: We generally recommend Tomcat 5.5.20 or above when enabling load-time weaving. Prior versions have known issues with custom ClassLoader setup.

Alternatively, consider the use of the Spring-provided generic VM agent, to be specified in Tomcat's launch script (see above). This will make instrumentation available to all deployed web applications, no matter which ClassLoader they happen to run on.

For a more detailed discussion of Tomcat-based weaving setup, check out the [the section called "Tomcat load-time weaving setup (5.0+)"] section which discusses specifics of various Tomcat versions. While the primary focus of that section is on JPA persistence provider setup, the Tomcat setup characteristics apply to general load-time weaving as well.

WebLogic, OC4J, Resin, GlassFish

Recent versions of BEA WebLogic (version 10 and above), Oracle Containers for Java EE (OC4J 10.1.3.1 and above) and Resin (3.1 and above) provide a ClassLoader that is capable of local instrumentation. Spring's native LTW leverages such ClassLoaders to enable AspectJ weaving. You can enable LTW by simply activating context:load-time-

weaver as described earlier. Specifically, you do *not* need to modify the launch script to add -javaagent:path/to/spring-agent.jar.

GlassFish provides an instrumentation-capable ClassLoader as well, but only in its EAR environment. For GlassFish web applications, follow the Tomcat setup instructions as outlined above.