Spring aop annotation example

11.8 Using AspectJ with spring applications Cutting that we have covered so far in this chapter is pure Spring AOP. In this section, we'll look at how you can use the AspectJ compactor/weaver instead of, or except, Spring AOP if your needs go beyond the facilities offered only by Spring AOP.11.8.1 Using AspectJ for addiction inject domain objects with Spring The Spring container instantiates and configure beans defined in your application context. It is also possible to ask the bean factory to configure an existing facility with respect to the name of the bean definition containing the configuration being applied. Spring aspects.jar a marker-based aspect that harnesses this ability to enable the injection of dependence of any object. The aid is intended to be used for items created outside the control of any container. Domain objects often fall into this category because they are often created programmically using a new operator or ORM tool as a result of a database query. The @Configurable indicates the class that is eligible for a spring-led configuration. In the simplest case, it can only be used as a label: the package com.xyz.myapp.domain; import org.springframework.beans.factory.annotation.Configurable; @Configurable public class account { } When used as a marker interface in this way, Spring will configure new instances of type with a label (Account in this case) using the bean definition (usually prototype-scope) with the same name as the fully qualified type name (com.xyz.myapp.domain.Account). Since the default name for beans is a fully qualified name of its species, an appropriate way to declaring the definition of a prototype is simply to omit the id attribute: &lt;bean class=com.xyz.myapp.domain.Account scope=prototype&gt; &lt;property name=fundsTransferService ref=fundsTransferService&gt;&lt;/property&gt; &lt;/bean&gt; If you want to explicitly specify the name of the definition of a bean prototype to use, you can do so directly in the note: package com.xyz.myapp.domain; import org.springframework.beans.factory.annotation.Configurable; @Configurable(account) Public Class Account { } Spring will now search for a bean definition called an account and use it as a definition to configure new instances of the account. You can also use automatic wiring to avoid having to define a special definition of beans at all. For Spring to apply automatic wiring, use the autowire property @Configurable notation: specify either @Configurable(autowire=Autowire.BY_TYPE) or @Configurable(autowire=Autowire.BY_NAME for automatic wiring by type or name. As an alternative, from spring 2.5 it is desirable to list an explicit annotation-based addiction injection for your @Configurable beans using @Autowired or @Inject at field level or method (see section 7.9, Annotation-based container configuration for details). details). You can enable spring dependencies to be checked for object references in a newly created and configured object by using the dependencyCheck attribute (for example: @Configurable(autowire=Autowire.BY_NAME,dependencyCheck=true)). If this attribute is set to trueness, spring will confirm after configuration that all properties (not primitive or collection) are set. Independent use of annotation does nothing, of course. It is AnnotationBeanConfigurerAspect in spring aspects.jar that acts on the presence of annotation. In fact, the aspect says after returning from initializing a new type object with @Configurable, configure the newly created object using Spring according to the properties of the stamp. In this context, initialisation refers to newly refurbished objects (e.g. objects currently with a new operator) as well as serial objects undergoing deserialisation (e.g. through readResolve()). Keep at the beginning one of the key phrases in the paragraph above is inline. For most cases, the exact semantics 'after returning from initialization of the new object' will be fine... in this context, after initialization means that addictions will be injected after the construction of the facility - this means that addictions will not be available for use in class constructors. If you want addictions to be injected before the constructor is executed and thus available for use in the body of the constructor, then you must define this on the @Configurable declaration in this way: @Configurable(preConstruction=true) You can find out more information about the language semantics of different pointcut types in aspectJ in this addition of the AspectJ Program Guide. For this to work, species snotated must be woven with the AspectJ weaver - for this you can use an ant or Maven task (see aspectJ Development Environment Guide for example) or weaving at the time of loading (see section 11.8.4,Load-time weaving with AspectJ in the spring box). AnnotationBeanConfigurerAspect itself should be configured until spring (to obtain a reference to the bean factory that will be used to configure new facilities). If you're using a Java-based configuration, simply add @EnableSpringConfigured to any @Configuration class. @Configuration @EnableSpringConfigured AppConfig Class { } If you prefer an XML-based configuration, the spring context space defines the appropriate context: an element configured for spring: Cases of @Configurable objects created before &lt;context:spring-configured&gt;&lt;/context:spring-configured&gt; aspect configurations will result in the release of a message to a debug log and no object configuration. An example may be beans in a spring configuration that creates domain objects when until spring. In this case, you can use the attribute depends on the beans to manually specify that the beans depend on the configuration aspect. &lt;bean id=myService class=com.xzy.myapp.service.MyService depends-on=org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect&gt;&lt;/bean&gt; Note Do not activate @Configurable processing through the bean configurer aspect unless you are really thinking of relying on its semantics at running time. In particular, make sure that you do not use @Configurable on bean classes that are registered as ordinary spring beans with a container: You would get double initialization otherwise, once through the tank and once through the aspect. Testing units @Configurable facilities One of the goals @Configurable support is to enable independent testing of domain objects without the difficulty associated with hard-coded viewlies. If @Configurable types are not woven by AspectJ, then the tag has no impact during unit testing, and you can simply set false or stub asset references in the object being tested and proceed as usual. If @Configurable type weave AspectJ, then you can still use the test outside the container as usual, but you will see a warning message each time you construct an @Configurable object indicating that it is not configured until spring. Working with multiple AnnotationBeanConfigurerAspect application contexts used to implement @Configurable is a AspectJ singleton aspect. The scope of the singleton aspect is the same as the scope of static members, i.e. there is one aspect of the instance per class load that defines the type. This means that if you define multiple application contexts within the same class load hierarchy, you need to consider where to define @EnableSpringConfigured beans and where to set spring aspects.jar on a class course. Consider a typical spring web application configuration with the context of a shared parenting application that defines shared business services and whatever it takes to support them and the context of a one-child-per-service application that contains definitions specified for that servlet. All these contexts will coexist within the same classloader hierarchy, so AnnotationBeanConfigurerAspect can only have a reference to one of them. In this case, we recommend defining @EnableSpringConfigured bean in the common (home) context of the application: this defines the services you're likely to want to inject into your domain properties. As a result, you can't configure domain objects with bean references defined in your child's (servlet-specific) contexts using the @Configurable mechanism (you probably don't want to do something anyway!). When deploying multiple Web applications within the same container, ensure that each web application loads types in the spring aspects.jar using its own charging tool (for example, by setting upweb-inf/lib). If the spring aspects.jar are added only to a wide class of containers (and therefore loaded by a common parent class), all web applications will share the same instance of the aspect that is probably not what you want. 11.8.2 Other spring aspects for AspectJ In addition to the @Configurable aspects, spring aspects.jar contains the AspectJ aspect that can be used to encourage Spring Transaction Management for the types and methods listed with @Transactional. This is primarily intended for users who want to use support for Spring Box transactions outside the spring container. The aspect interpreted by @Transactional is AnnotationTransactionAspect. When using this aspect, you need to highlight the deployment class (and/or methods within that class), not the interface (if any) that the class implements. AspectJ follows the Java rule that notations on interfaces are not inherited. The @Transactional on the class specifies the default transactional semantics for performing any public operation in the class. The @Transactional method within the class overrides the default transactional semantics provided by the class label (if present). Methods of any visibility can be visible, including private methods. Directly noticing non-disclosure is the only way to obtain transaction delimitation for the implementation of such methods. Advice Since spring Framework 4.2, spring-aspects provide a similar aspect that offers exactly the same features for standard javax.transaction.Transactional annotation. Check JtaAnnotationTransactionAspect for more details. For AspectJ developers who want to use support for spring configuration and transaction management but don't want (or can't) use annotations, spring aspects.jar it also contains abstract aspects that you can expand to provide your own pointcut definitions. See sources for AbstractBeanConfigurerAspect and AbstractTransactionAspect aspects for more information. As an example, the following excerpt shows how you might write an aspect to configure all cases of objects defined in a domain model using a prototype of bean definitions that match fully qualified class names: domainobjectConfiguration public aspect extends AbstractBeanConfigurerAspect { public DomainObjectConfiguration() { setBeanWiringInfoResolver (new ClassNameBeanWiringInfoResolver()); } protected pointcut beanCreation(Object beanInstance) : initialization(new(..)) &amp;&amp; SystemArchitecture.inDomainModel() &amp;; this(beanInstance); } 11.8.3 Configuring AspectJ aspects using Spring IoC When using AspectJ aspects with spring applications, it is natural to want and expect to be able to configure such aspects using spring. AspectJ runtime itself is responsible for creating aspects, and the means to configure aspects created through spring depend on AspectJ instantiation model (per-xxx clause) that uses the aspect. Most AspectJ aspects are singleton aspects. The configuration of these aspects is very simple: simply create a definition of beans that refers to the type of aspect as normal and include the bean attribute factory mode = aspectOf. This ensures that Spring gets an instance of the aspect by asking aspectj for it instead of trying to create the instance itself. For example: &lt;bean id=profiler class=com.xyz.profiler.Profiler factory-method=aspectOf&gt; &lt;property name=profilingStrategy ref=jamonProfilingStrategy&gt;&lt;/property&gt; &lt;/bean&gt; non-singleton aspects are harder to configure: however, this can be done by creating a prototype of bean definitions and using @Configurable support from spring aspects.jar to configure aspects after beans have created AspectJ runtime. If you have @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring AOP, and these aspects are configured using spring, and you will need to say spring AOP @AspectJ support for automaticproxia that @AspectJ @AspectJ the exact subset of @AspectJ aspects defined in the configuration should be used for automatic protection. You can do this by using one or more &lt;include&gt;&lt;/include&gt; elements within &lt;aop:aspectj-autoproxy&gt;&lt;/aop:aspectj-autoproxy&gt; declarations. Each element specifies a pattern of names, and only beans with names aligned with at least one of the samples will be used &lt;include&gt;&lt;/include&gt; for spring AOP auto-toxic configuration: &lt;aop:aspectj-autoproxy&gt; &lt;aop:include name=thisBean&gt;&lt;/aop:include&gt; &lt;aop:include name=thatBean&gt;&lt;/aop:include&gt; &lt;/aop:aspectj-autoproxy&gt; Note Don't be fooled by the name of the element: using &lt;aop:aspectj-autoproxy&gt;&lt;/aop:aspectj-autoproxy&gt; will result in the creation of a Spring AOP proxy. The @AspectJ aspect of the declaration is used here right now, but AspectJ work time is not included. 11.8.4 Weaving load times with AspectJ in the Spring Load Time Weaving (LTW) box refers to the process of weaving AspectJ aspects into the application's class files as they load into a Java virtual machine (JVM). The focus of this section is on the configuration and use of LTW in the specific context of the Spring Box: this section is not an introduction to the LTW after all. For full details on LTW specifics and LTW configuration with AspectJ only (with spring not included at all), see the LTW section of the AspectJ Development Environment Guide. The added value that the Spring Frame brings to AspectJ LTW is in enabling much finer grainy control over the weaving process. 'Vanilla' AspectJ LTW is influenced by java (5+) means, which is turned on by specifying the VM argument when running JVM. This is therefore a JVM-level setting, which in situations can be fine, but it is often a little too rough. Spring-supported LTW lets you tune in to turn on by classLoader, which is obviously more fine-grained and that may make more sense in a single-JVM-multiple-application environment (as it is found in a typical application server environment). Furthermore, in certain environments, this support allows you to weave load times without making any changes to the application server startup script that will be required to add -

javaagent:path/to/aspectjweaver.jar or (as we describe later in this section) -javaagent:path/to/org.springframework.instrument-{version}.jar (previously named spring-agent.jar). Developers simply modify one or more files that form the application context to allow weaving at load time instead of relying on administrators who are typically in charge of configuring deployments such as startup scripts. Now that the sales pitch is complete, let's first take a quick walk through aspectJ LTW using Spring, followed by detailed details about the elements introduced in the following example. For a complete example, check out petklinic's sample app. Suppose you're an app developer who's been tasked with diagnosing the cause of some performance issues in your system. Instead of introducing a profiling tool, what we will do is include a simple aspect of profiling that will allow us to get some performance metrics very quickly, so that immediately afterwards we can apply a more precise profiling tool to that particular area. Note the example here uses xml style configuration, it is also possible to configure and use @AspectJ with Java configuration. Specifically, @EnableLoadTimeWeaving can be used as an alternative to &lt;context:load-time-weaver&gt;&lt;/context:load-time-weaver&gt; (see below for details). Here's the profiling aspect. Nothing too classy, just a fast-dirty weather-based profiler, using @AspectJ-style statements. foo packages; 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 a META-INF/aop.xml file, to inform AspectJ weaver that we want to weave our ProfilingAspect into our classes. This convention file, that is, the presence of a file (or file) on a Java classpath called aop.xml je AspectJ. &lt;! DOCTYPE aspectj PUBLIC -//AspectJ//DTD//EN -//AspectJ//DTD//EN Now to the spring part of the configuration. &lt;aspectj&gt; &lt;weaver&gt; &lt;include within=foo.*&gt;&lt;/include&gt; &lt;/weaver&gt; &lt;aspects&gt; &lt;aspect name=foo. ProfilingAspect&gt;&lt;/aspect&gt; &lt;/aspects&gt; &lt;/aspectj&gt; We need to configure LoadTimeWeaver (all explained later, just take it to trust for now). This load time weaver is an essential component responsible for weaving aspect configurations in one or more META-INF/aop.xml files into 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 other options that you can specify, but these are detailed later).&lt;?xml version=1.0 encoding=UTF-8?&gt;Now that all the necessary artifacts are in place - aspect, META-INF/aop.xml file, and spring configuration -, let's create a simple class of drivers with the main (..) method for demonstrating LTW in action. foo package; import org.springframework.context.support.ClassPathXmlApplicationContext; public end class Main { public static gap main(String[] args) { ApplicationContext ctx = new ClassPathXmlApplicationContext(beans.xml, Main.class); &lt;beans xmlns= xmlns:xsi= xmlns:context= schemalocation= amp;gt; &lt;bean id=entitlementCalculationService class=foo. StubEntitlementCalculationService&gt;&lt;/bean&gt; &lt;context:load-time-weaver&gt;&lt;/context:load-time-weaver&gt; &lt;/beans&gt; Right AccountService rightCalculationService = (NameCalculationService) ctx.getBean(realCalculationService); rightsCalculationService.calculateEntitlement(); } There's one more thing to do. The introduction to this section is said to be able to include LTW selectively on a per-classLoader basis with spring, and this is true. However, just for this example, we will use a Java agent (supplied with Spring) to switch LTW. This is the command line that we will use to run the above main class: java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo. The main -javaagent is a flag for determining and enabling agents to instrument programs running on JVM. Spring Framework is supplied with such an asset, InstrumentationSavingAgent, which is packed in a spring instrument.jar which is delivered as the value of the -javaagent argument in the example above. The way out of the execution of the main program will look like something like that below. (I introduced Thread.sleep(..) statement in the implementation of calculateEntitlement() so that the profiler actually records something other than 0 milliseconds - 01234 milliseconds are not overhead that introduces :)) Calculating The Real Stopwatch ProfilingAspect: Running Time (Milis) = 1234 ------ ----- ---------------------------- ms % Task Name ------ ----- ---------------------------- 01234 100% calculateEntitlement Since this LTW is influenced by the full aspect, we are not limited to advising spring beans; the next small variations on main program will give the same result. foo packages; import org.springframework.context.support.ClassPathXmlApplicationContext; public final class Main { public static gap main (String[] args) { new ClassPathXmlApplicationContext(beans.xml, Main.class); RightService LawService = new StubEntitlementCalculationService(); rightsCalculationService.calculateEntitlement(); } Notice that in the above program we simply lift the spring tank, and then create a new instance of StubEntitlementCalculationService completely outside the context of spring... profiling tips continue to weave. The example is, however, simple... however, the basics of LTW support in the spring are all introduced in the example above, and the rest of this section will explain in detail why behind each part of the configuration and use. For example, the ProfilingAspect used in this example can be basic, but it is very useful. This is a nice example of the development aspect that developers can use during development (of course), and then very easily exclude from the construction of an application that is implemented in UAT or production. The aspects you use in LTW must be aspects of AspectJ. They can be written in the aspectj language itself or you can write your aspects @AspectJ style. This means that your aspects are then both valid AspectJ and Spring AOP aspects. Furthermore, the assembled aspect of the class must be available on classpath. AspectJ LTW infrastructure is configured using one or more META-INF/aop.xml files, which are located on Java classpath (directly or more typically in jar files). The structure and contents of this file are detailed in aspectJ's main reference documentation, and an interested reader is referred to that resource. (I appreciate that this section is short, but the aop.xml file is 100% AspectJ - there is no spring-specific information or semantics relating to it, so there is no added value that I can contribute either as a result), so instead of rehoming the quite satisfying section that AspectJ developers wrote, I'm just directing you there.) Necessary Libraries (JARS) At the very least you will need the following libraries to use the support of the Spring Framework for AspectJ LTW: spring-aop.jar (version 2.5 or later, plus all mandatory dependencies) aspectjweaver.jar (version 1.6.8 or later) If you use a spring-provided agent to enable instrumentation, you will also need: Key In Spring's LTW support is the LoadTimeWeaver interface (bundled with org.springframework.instrument.classloading), and numerous deployments of that spring distribution ship. LoadTimeWeaver is responsible for adding one or more java.lang.instrument.ClassFileTransformers to ClassLoader at the time of operation, which opens the door to all kinds of interesting applications, one of which happens to be LTW aspects. Tip If you are not familiar with the idea of transforming runtime class files, we encourage you to read the javadoc API documentation for the java.lang.instrument package before proceeding. It's not a big job because there's - rather embarrassingly - precious little documentation there... key interfaces and classes will at least be placed in front of you for reference as you read through this section. Configuring LoadTimeWeaver for a specific ApplicationContext can be as simple as adding a single line. (Note, for example, that you'll almost certainly need to use ApplicationContext as your spring container - usually BeanFactory won't be enough because LTW support uses BeanFactoryPostProcessors.) To enable Spring Framework LTW support, you must configure LoadTimeWeaver, which is usually done using @EnableLoadTimeWeaving annotation. @Configuration @EnableLoadTimeWeaving appconfig class { } Alternatively, if you prefer an XML-based configuration, use the &lt;context:load-time-weaver&gt;&lt;/context:load-time-weaver&gt; element. Note that the element is defined in the context of the name. &lt;?xml version=1.0 encoding=UTF-8?&gt;&lt;beans xmlns= xmlns:xsi= xmlns:context= xsi:schemalocation= amp;gt; &lt;context:load-time-weaver&gt;&lt;/context:load-time-weaver&gt; &lt;/beans&gt; The above configuration will automatically define and register a certain number of LTW-specific infrastructure grains, such as LoadTimeWeaver and AspectJWeavingEnabler. The default LoadTimeWeaver is the DefaultContextLoadTimeWeaver class, which tries to decorate the auto-detected LoadTimeWeaver: the exact type of LoadTimeWeaver that will be automatically detected depends on your environment during operation (summarized in the following table). Table 11.1. DefaultContextLoadTimeWeaver LoadTimeWeavers Runtime EnvironmentLoadTimeWeaver deploymentRunning in Oracle webLogicWebLogicLoadTimeWeaverRunning in Oracle's GlassFishGlassFishLoadTimeWeaverRunning in Apache TomcatTomcatLoadTimeWeaverRunning at JBoss AS or WildFlyJBossLoadTimeWeaverRunning in IBM's WebSphereWebSphereLoadTimeWeaverJVM began spring(java -javaagent:path/to/spring-instrument.jar) InstrumentationLoadTimeWeaverFallback, expecting the core ClassLoader to follow the common conventions (e.g. applicable to TomcatInstrumentableClassLoader and Resin)ReflectiveLoadTimeWeaver Note that these are only LoadTimeWeavers that are automatically detected when using DefaultContextLoadTimeWeaver: of course it is possible to determine exactly which loadTimeWeaver deployment you want to use. To specify a specific LoadTimeWeaver with Java configuration, implement the LoadTimeWeavingConfigurer interface and override the getLoadTimeWeaver() method: @Configuration @EnableLoadTimeWeaving public class AppConfig implements LoadTimeWeavingConfigurer { @Override public LoadTime LoadWeaver getLoadTimeWeaver() { return new ReflectiveLoadTimeWeaver(); } If you are using an XML-based configuration, you can specify a fully qualified name as the value of the weaver class attribute on &lt;context:load-time-weaver&gt;&lt;/context:load-time-weaver&gt; element: &lt;?xml version=1.0 encoding=UTF-8?&gt;&lt;beans xmlns= xmlns:xs xmlns:context= xsi:schemalocation= amp;gt; &lt;context:load-time-weaver-class=org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver&gt;&lt;/context:load-time-weaver&gt; The &lt;/beans&gt; LoadTimeWeaver defined and registered by configuration can be retrieved later from the Spring container using the famous loadTimeWeaver name. Remember that LoadTimeWeaver exists only as a mechanism for Spring's LTW infrastructure to add one or more ClassFileTransformers. The actual ClassFileTransformer that runs LTW is the PreProcessorAgentAdapter class (from the class org.aspectj.weaver.loadtime package). See javadoke classPreProcessorAgentAdapter for additional details, because the specifics of how weaving is actually effective are outside the scope of this section. One final configuration trait remains: the aspectjWeaving (or aspectj-weaving) attribute if you are using XML). This is a simple attribute that controls whether LTW is enabled or not; It's as simple as that. Accepts one of three possible values, summarized below, according to a given value that is automatic if the attribute is not present. Table 11.2. AspectJ weaving value attribute Annotation ValueXML ValueExplanationENABLEDonAspectJ weaving is on, and aspects will be woven at load time as needed. DISABLEDoffLTW is off... no aspect will be woven at load time. AUTODETECTautodetectAko spring LTW infrastructure can find at least one file, then AspectJ AspectJ is on, otherwise it's off. This is the default value. Environment-specific configuration This last section contains all the additional settings and configuration you will need when using Spring's LTW support in environments such as application servers and web containers. Historically, Apache Tomcat's default class loader has not supported class transformation, which is why Spring provides improved implementation that addresses this need. Called TomcatInstrumentableClassLoader, the loader runs on Tomcat 6.0 and up. Tip Do not define TomcatInstrumentableClassLoader more on Tomcat 8.0 and up. Instead, let Spring automatically use Tomcat's new original InstrumentableClassLoader object through the TomcatLoadTimeWeaver strategy. If you still need to use TomcatInstrumentableClassLoader, it can be registered individually for each web application as follows: Copy org.springframework.instrument.tomcat.jar at $CATALINA_HOME/lib, where $CATALINA_HOME represents the root of the Tomcat installation) Refer Tomcat &lt;Context path=/myWebApp docbase=/my/webApp/location&gt; &lt;Loader loaderclass=org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader&gt;&lt;/Loader&gt; &lt;/Context&gt; to use custom class loader (instead of default) by editing web application context file : Apache Tomcat (6.0+) supports several contextual locations: server configuration file - $CATALINA_HOME/conf/server.xml default context configuration - $CATALINA_HOME/conf/context.xml - this affects all web application implemented by web application configuration that can be deployed or server-side on $CATALINA_HOME/conf/[enginename]/[hostname]/[webapp]-context.xml or embed within a Web application archive in META-INF/context.xml For efficiency , a built-in configuration style per Web application is recommended because it will only affect applications that use a custom class loader and do not require any changes to the server configuration. See Tomcat 6.0.x documentation for more details on available context locations. Alternatively, consider using a generic VM agent that provides spring, which will be listed in Tomcat's launch script (see above). This will make instrumentation available to all implemented web applications, regardless of what ClassLoader it is working on. WebLogic, WebSphere, Resseal, GlassFish, JBoss Recent versions of WebLogic Server (version 10 and up), IBM WebSphere Application Server (version 7 and up), Resseal (3.1 and up) and JBoss (6.x or more) provide ClassLoader that is capable of local instrumentation. Spring's original LTW uses such ClassLoaders to enable AspectJ weaving. You can enable LTW by simply activating the weaving at the time of load as described earlier. Specifically, you don't need to change the launch script to add Please note that ClassLoader is capable of GlassFish GlassFish instruments only in the EAR environment. For GlassFish web applications, follow the instructions to set up Tomcat as listed above. Keep on the note that on JBoss 6.x scanning server applications should be disabled to prevent classes from loading before the app really starts. The quick fix is to add to your artifact a file called WEB-INF/jboss-scanning.xml with the following content: Generic Java Applications &lt;scanning xmlns=urn:jboss:scanning:1.0&gt;&lt;/scanning&gt; When class instrumentation is required in environments that are not supported or supported by existing LoadTimeWeaver deployments, a JDK agent can be the only solution. For such cases, Spring provides InstrumentationLoadTimeWeaver, which requires a spring-specific (but very general) VM agent, org.springframework.instrument-{version}.jar (previously appointed spring-agent.jar). To use it, you must run a virtual machine with a Spring agent, supplying the following JVM options: -javaagent:/path/to/org.springframework.instrument-{version}.jar Keep a hint that this requires modifying the VM startup script that can prevent you from using this in application server environments (depending on your work policies). In addition, a JDK agent will instrument an entire VM that may prove costly. For performance reasons, it is recommended to use this configuration only if your target environment (such as Jetty) does not have (or does not support) a dedicated LTW. LTW, LTW.