

# How to integrate Aspects in J2EE apps with aop.xml

Authors: [Alexandre Vasseur](#)

## Introduction

This tutorial tries to walk you through the idea of Aspects deployment we have introduced in **AspectWerkz**.

This tutorial is using **AspectWerkz 2.0**.

You will learn how to integrate aspects and aspects libraries as regular components in your J2EE environment through the help of **META-INF/aop.xml** and **WEB-INF/aop.xml** files.

It won't give a lot of detail on what is AOP, how the aspects are implemented using AspectWerkz, etc...

If you want to read more about that first, you can read and practice the

- [HelloWorld tutorial](#)
- [HelloWorldHijacked tutorial](#)

In this tutorial, we will use **Tomcat 5** samples applications since everyone can download it easily. Some more articles are on their way to explain integration on **BEA WebLogic Server** and **Apache Geronimo**, where the same concepts apply.

## Installation

I assume you have a Java 1.4 correctly installed.

*Download* the **2.0 AspectWerkz release** (or latest RC) and *unzip* it into a relevant location. I will use C:\aw\aspectwerkz\

This tutorial is based on the 2.0 version of AspectWerkz.

The latest distribution can be found [here](#).

After installation you need to set the `ASPECTWERKZ_HOME` *environment variable* to point to the installation directory. This is because quite a few of the scripts use this to find the required libraries and those scripts aims at making this tutorial

simpler to understand by focussing on the topic to cover.

Below is some samples (for Windows - adapt for Unix/Linux and do an `export ASPECTWERKZ_HOME`)

You should check that your `JAVA_HOME` is properly configured as well.

```
set ASPECTWERKZ_HOME=C:\aw\aspectwerkz
echo %ASPECTWERKZ_HOME%
echo %JAVA_HOME%
```

Then you need to *install Tomcat 5*, since we will deploy our aspects in Tomcat samples applications.

The distribution 5.0.25 can be found [here](#) (choose *jakarta-tomcat-5.0.25.zip*). *Download* and *unzip* it to somewhere relevant.

I will use C:\aw\jakarta-tomcat-5.0.25\

After installation, it is wise to *check the Tomcat installation* by launching

```
C:\aw\jakarta-tomcat-5.0.25\bin\startup.bat
(or use .sh on Unix/Linux)
```

and open your browser to (this link should work) <http://localhost:8080/> to access the main Tomcat page.

In the next parts of the tutorial, we will go thru some Tomcat servlet samples <http://localhost:8080/servlets-examples/> and jsp examples <http://localhost:8080/jsp-examples/>.

*Shutdown* Tomcat by closing the MSDos window or doing some CTRL-C twice.

You can check the [attachments](#) if you don't want to code or build the single class as explained in the following sections. I advise you to practice some instead of downloading them.

## The reusable Aspect

Since Tomcat provides several sample applications, we have to provide a test Aspect that we can use and reuse to understand the different aspect deployment schemes.

To make things easy to understand, we will not go into the details.

Just notice that it is yet another tracing aspect, and that the aspect method named "trace" will be our around advice.

This aspect will print some information about himself, like the hashcode of the ClassLoader that created it (I do not use toString() since Tomcat is too verbose, but you can give it a try).

We will pack this Aspect in different ways in the next parts of the tutorial, but we can already compile it as a regular Java class.

I will write this Aspect in C:\aw\tomcat, in the package demoAOP

```
mkdir C:\aw\tomcat\demoAOP
cd C:\aw\tomcat\demoAOP
```

```
package demoAOP;
import
org.codehaus.aspectwerkz.joinpoint.JoinPoint
;

/** Our Aspect is just a Java class */
public class DemoAspect {
```

```

private int indent = -1;

/** a method for pretty formatting */
private void log(String message) {
    for (int i = 0; i < indent; i++) {
        System.out.print(" ");
    }
    System.out.print("DemoAspect[" +
this.getClass().getClassLoader().hashCode()
+ "]" );
    System.out.println(message);
}

/** An around advice is just a method
with this JoinPoint parameter */
public Object trace(JoinPoint jp) throws
Throwable {
    indent++;
    log("--> " + jp.toString());
    Object result = jp.proceed(); //
will call the next advice or target method,
field access, constructor etc
    log("<--");
    indent--;
}

```

```
        return result;
    }
}
```

This is a standard Java class, and can be *compiled* with `javac ... demoAOP\HelloWorld.java` to produce the regular ".class" file in the "target" directory. We need to link to the aspectwerkz jar due to this JoinPoint class that is used in the source code of the DemoAspect, so we should write

```
javac -classpath %ASPECTWERKZ_HOME%\lib\aspectwerkz-2.0.jar demoAOP\DemoAspect.java,
```

but usage of the environment variable makes sense here to set the CLASSPATH:

```
cd C:\aw\tomcat
%ASPECTWERKZ_HOME%\bin\setEnv
javac demoAOP\DemoAspect.java
```

From there we have several options if we are using online mode (class load time weaving), which is standardized in Java 1.5 with [JSR-163](#) but that AspectWerkz support even for Java 1.3:

- copy the aspect .class file in the WEB-INF/classes directory and write a WEB-INF/aop.xml AOP XML descriptor to define the aspect pointcut within the web application scope
- package the aspect .class file in a jar file, and add a META-INF/aop.xml in this jar file. Copy this jar in the WEB-INF/lib directory.
- instead of deploying our aspect within a specific web application, we could copy the jar at the system classpath level. The interesting thing to understand is that such a system wide META-INF/aop.xml AOP XML descriptor will affect all web applications due to java ClassLoader hierachies.
- and of course we could have both: some aspects at the application level with a WEB-INF/aop.xml and some aspects at the system wide level with a META-INF/aop.xml

When using offline mode (post compilation of your classes before deployment), the same results can be achieved, though it can be sometimes a bit tricky to remember to weave the application both for its own aspects and for the system level aspects.

We will not go through offline mode in this tutorial (refer to HelloWorld tutorial).

To keep things simple, we will always use the following pointcut that will allow to pick out all public methods execution:

```
execution(public * *.*.*(..))
```

We carrfully exclude the package name of the aspect thus:

```
execution(public * *.*.*(..)) &&
!within(demoAOP..*)
```

As a consequence, the following AOP XML descriptor will be used.  
Write this file in C:\aw\tomcat\META-INF\ao.xml for now.

```
<!DOCTYPE aspectwerkz PUBLIC
    "-//AspectWerkz//DTD//EN"

"http://aspectwerkz.codehaus.org/dtd/aspectw
erkz.dtd">

<aspectwerkz>
    <system id="webapp">
        <aspect class="demoAOP.DemoAspect">
            <pointcut name="allPublic"
expression="execution(public * *.*.*(..))
AND !within(demoAOP..*)" />
                <advice name="trace"
type="around" bind-to="allPublic" />
            </aspect>
        </system>
    </aspectwerkz>
```

Note that we could have simplified this XML by defining our `DemoAspect` with annotations (see HelloWorld tutorial for this).

Some of you will have already understand that when the aspect is **deployed within the web application**, its weaving scope will not be the same: the pointcut will pick out public method from classes *really loaded* by the web application (e.g. not `javax.servlet.http.HttpServlet` class).

## Enabling online mode in Tomcat

Since we will use online mode, we need to alter some of the Tomcat startup scripts. If we were using BEA JRockit or Java 1.5, it would be a matter of adding a single JVM option.

Since I will not make any assumption on your JVM environment, I will skip the interesting details in this part and we will use a non-optimized but out-of-the-box solution that ships with the AspectWerkz distribution: the command line tool in ASPECTWERKZ\_HOME/bin/aspectwerkz ([read doc](#)).

**Note: for Tomcat 5.0.27 and after, please refer to the comments at the end of the article**

Open the Tomcat startup script `bin/catalina.bat` (C:\aw\jakarta-tomcat-5.0.25\bin\catalina.bat).

The main idea is to set ASPECTWERKZ\_HOME in this script and replace `JAVA_HOME/bin/java` with `ASPECTWERKZ_HOME/bin/aspectwerkz`.

Locate at the end of the script:

```
...
set _EXECJAVA=%JAVA_HOME%\bin\java

rem Execute Java with the applicable
properties
if not "%JPDA%" == "" goto doJpda
...
```

and *alter* it as follows:

```
...
@REM ** Begin of AspectWerkz configuration
**
@REM ** Adapt ASPECTWERKZ_HOME **
set ASPECTWERKZ_HOME=C:\aw\aspectwerkz
set
_EXECJAVA=%ASPECTWERKZ_HOME%\bin\aspectwerkz
.bat
@REM ** we can add some more AspectWerkz
option this way:
set JAVA_OPTS=%JAVA_OPTS%
-Daspectwerkz.transform.details=false
...
@REM ** End of AspectWerkz configuration **

rem Execute Java with the applicable
properties
if not "%JPDA%" == "" goto doJpda
...
```

For a complete understanding of the different options and other modes, refer to the [AspectWerkz weaving schemes documentation](#).

Note that as a consequence of using the command line tool for simplicity here, we will run the Tomcat with -Xdebug JVM mode (with Java 1.4).

***Do not startup Tomcat for now.***

## **Aspect deployed in the application with a WEB-INF/aop.xml**

This section explains the main step required and consequences of packaging an aop.xml file in your deployed application (WEB-INF/aop.xml).

The big picture is as follow:

```
Application server / container
```

```
|  
|
```

```
Web application
```

```
|
```

```
WEB-INF/
```

```
|
```

```
|___ aop.xml (web application  
AOP deployment descriptor)
```

```
|
```

```
|___ classes/ (regular  
packaging for our Aspect class)
```

In this section we decide to package our aspect as a regular class of the web application, and we define the pointcut and activate the aspect through the help of the WEB-INF/aop.xml file.

As a consequence, only the classes of the web application will be eventually weaved (if the pointcut match), even if the pointcut is supposed to match everything (all public method execution as explained in the previous section). In fact, the deployment mechanism tied to the web application deployment add an unbreakable isolation, and only public method execution **of classes loaded belonging to the web application** will be targetted by our `DemoAspect`.

Lets apply our Aspect to the Tomcat jsp-examples web application:

Open the  
C:\aw\jakarta-tomcat-5.0.25\webapps\jsp-examples\WEB-INF folder of the jsp-examples web application.  
Copy the C:\aw\tomcat\demoAOP folder (compiled Aspect) in the WEB-INF\classes subfolder just as regular web app classes. You should end up in having a  
C:\aw\jakarta-tomcat-5.0.25\webapps\jsp-examples\WEB-INF\classes\demoAOP\DemoAspect.class file.

Lets add the AOP XML descriptor to the web app:

Copy the C:\aw\tomcat\META-INF\aop.xml file to the WEB-INF directory of the web app

To sum up we have only

- added the aspect class to the web app as regular web app packaging dictates to do
- added a WEB-INF/aop.xml file to the web app

*Lets start Tomcat* with the bin/startup.bat script (which will delegate to our modified catalina.bat script). On the *stdout* window some AspectWerkz message should appear, with some verbose blocks of logs about aop.xml deployment.

```
...
*****
*****
* ClassLoader =
org.apache.catalina.loader.WebappClassLoader
@7088278
* SystemID = webapp
* Aspect total count = 1
*
file:/C:/aw/tomcat/java/jakarta-tomcat-5.0.2
5/webapps/jsp-examples/WEB-INF/aop.xml
*****
*****
...

```

And immediately we should see our DemoAspect at work, with some information about the web app initialization.

```
DemoAspect[7088278] -->
METHOD_EXECUTION:org.apache.jsp.jsp2.el.basi
c_002darithmetic_jsp._jspService
    DemoAspect[7088278] -->
METHOD_EXECUTION:listeners.SessionListener.s
essionCreated
    DemoAspect[7088278] <--
DemoAspect[7088278] <--
DemoAspect[7088278] -->
METHOD_EXECUTION:org.apache.jsp.jsp2.simplet
ag.hello_jsp._jspService
DemoAspect[7088278] -->
METHOD_EXECUTION:jsp2.examples.simpletag.Hel
loWorldSimpleTag.doTag
```

You will notice this `7088278` hashcode here, which indeed match the web app classloader hashcode as logged at deployment time.

By browsing the [jsp-samples](#) web application, we learn more about public method execution **of the web application**

Off course, no information about Tomcat itself since the aspects was deployed **within** the web application.

This section is interesting, and I hope you have already some new ideas to deploy your aspects in your J2EE application with AspectWerkz.

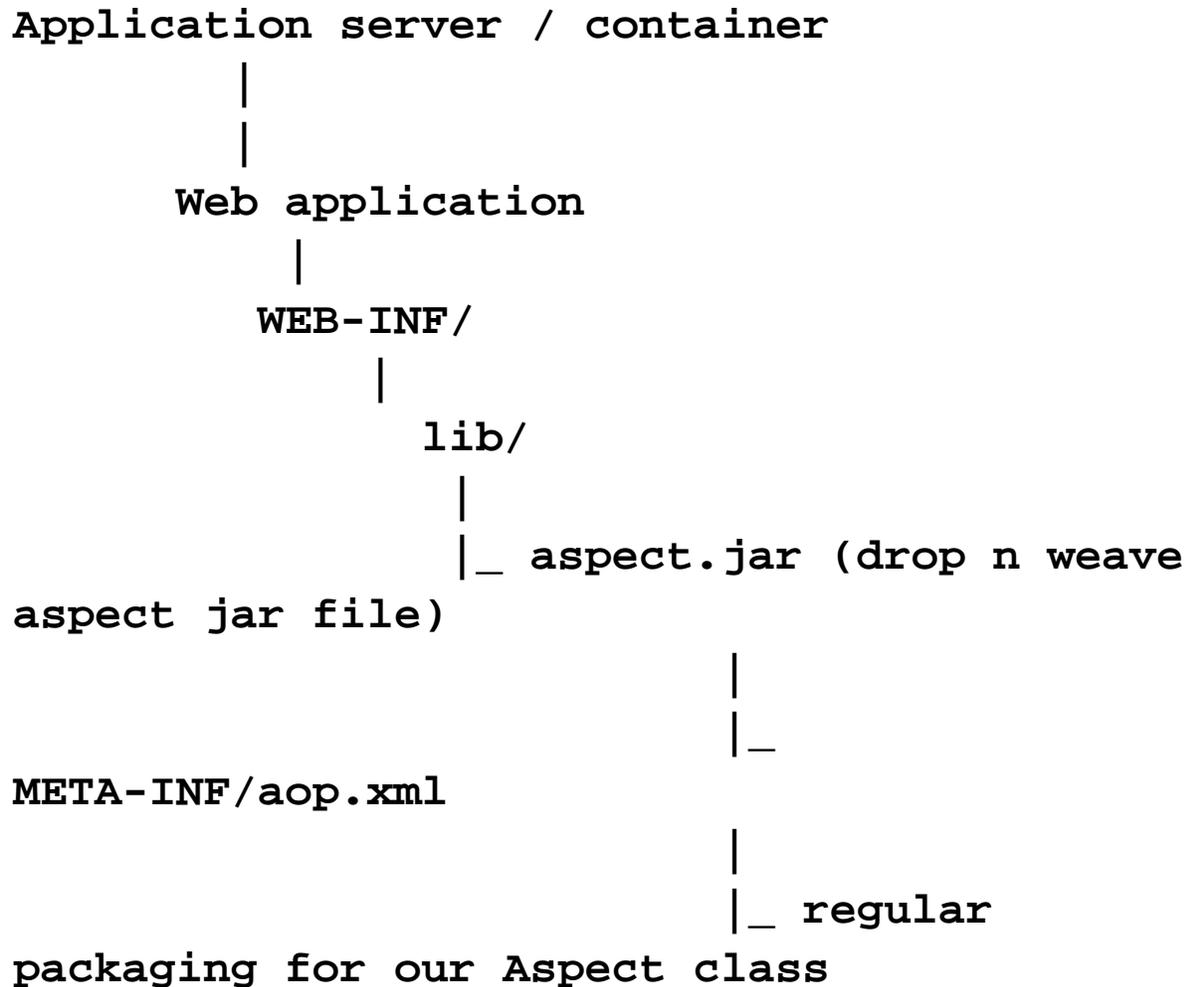
But what happens if you would like to provide some ready to use out of the box aspects, completely obliouvness to the user ?

*Shutdown* Tomcat for the next section.

## Aspect deployed in the application with an Aspect jar file

This section explains the main step required to package a "*drop n weave*" aspect jar file that contains a META-INF/aop.xml file, and consequence of adding this jar file in your deployed application (WEB-INF/lib).

The big picture is as follow:



In this section we will achieve the **same result** as in previous section, but in a slightly different way. We will package our Aspect in a jar file, and we will add the `aop.xml` file in the jar file itself in the special `META-INF` folder, so that our jar file is completely self defined. The jar file is then added to the web application as a regular jar file. As a consequence of web application deployment, the `aop.xml` file from the aspect jar file will still be deployed by the web application class loader, and the same behavior will happen: the aspect will target classes loaded by the web application.

Lets prepare the jar file (regular java / jar packaging rules applies)

```

cd C:\aw\tomcat
jar -cf ..\aspect.jar *

```

You will end up in having a `C:\aw\aspect.jar` file ready to use.

Lets add this *drop n weave* jar file to the Tomcat *servlet samples* web application.

```
Open the
C:\aw\jakarta-tomcat-5.0.25\webapps\servlet-
examples\WEB-INF folder
Create the lib subdirectory (to have a
WEB-INF/lib)
Copy the C:\aw\tomcat\aspect.jar file to it
```

*Startup Tomcat.*

Note: if you still have the **jsp** sample application with its aspect, you will see some messages as well as detailed in previous section.

The following should appear on stdout: some messages about the deployment of our aspect.jar file and its META-INF/aop.xml AOP XML deployment descriptor and right after that, our DemoAspect at work.

```

*****
*****
* ClassLoader =
org.apache.catalina.loader.WebappClassLoader
@22041176
* SystemID = webapp
* Aspect total count = 1
*
jar:file:/D:/java/jakarta-tomcat-5.0.25/webapp
s/servlets-examples/WEB-INF/lib/aspect.jar
!/META-INF/aop.xml
*****
*****
DemoAspect[22041176] -->
METHOD_EXECUTION:listeners.ContextListener.c
ontextInitialized
DemoAspect[22041176] <--
DemoAspect[22041176] -->
METHOD_EXECUTION:listeners.SessionListener.c
ontextInitialized
DemoAspect[22041176] <--
DemoAspect[22041176] -->
METHOD_EXECUTION:filters.ExampleFilter.init
DemoAspect[22041176] <--
DemoAspect[22041176] -->
METHOD_EXECUTION:compressionFilters.Compress
ionFilter.init
DemoAspect[22041176] <--

```

Something interesting: if you kept the jsp sample application aspect, you realize that you deployed the same aspect

class and the same aop.xml file twice. This is made possible due to class loader hierarchies. They don't see each other and the *naming space* is guaranteed by the class loader and by the AspectWerkz AOP container.

Browse some in the [servlet sample application](#) and see the DemoAspect at work.

We will learn something more about that in the next part...

Have you heard that "Aspects are everywhere" (c) The AspectWerkz team ?

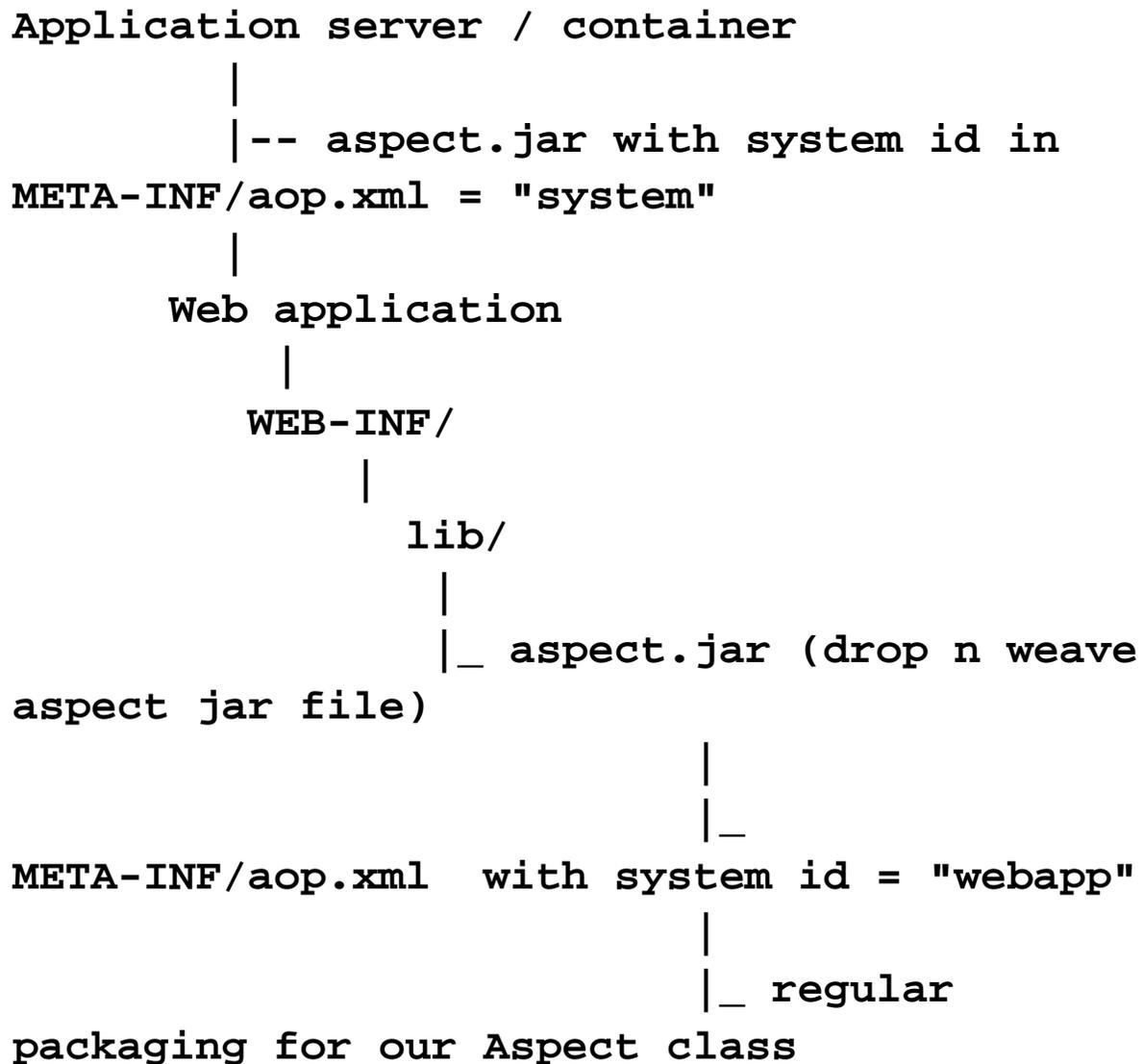
Go on reading !

*Shutdown Tomcat*

## Aspect jar file at the system classpath

This section explains the main step required to package **several META-INF/aop.xml** files and understand what happens then.

The big picture is as follow:



In this section we keep the packaging of the previous section, but we add a new aspect.jar file at the Tomcat wide level (system classpath) to check if we can have there a more wider pointcut. Remember the pointcut is supposed to match all public method, but that AOP container behavior enforce isolation based on the regular class loader hierarchy.

There is an important things to do.

AspectWerkz AOP container requires system id as defined in the *aop.xml* file to be unique **within a class loader hierarchy**. That means:

1. we can not just copy the previous aspect.jar at the system classpath level without changing its id
2. we could just copy the previous aspect.jar to some other webapplication, since then it will be in a parallel classloader hierarchy due to regular J2EE classloading schemes

Lets change the `META-INF/aop.xml` then and may be lets reduce some the scope of the pointcut so that we don't have too much data:

```
Open the C:\aw\tomcat\META-INF\aop.xml file
Change the <system id="..."> value to
"system" instead of "webapp" (use the name
you want)
Change the pointcut so that it is a bit
narrowed to match all public method in
whatever direct
subclass of whatever class/interface of
javax.servlet.http.*
```

The `C:\aw\tomcat\META-INF\aop.xml` file will look like

```
<!DOCTYPE aspectwerkz PUBLIC
    "-//AspectWerkz//DTD//EN"

"http://aspectwerkz.codehaus.org/dtd/aspectw
erkz.dtd">

<aspectwerkz>
    <system id="system">
        <aspect class="demoAOP.DemoAspect">
            <pointcut name="nonPrivate"
expression="execution(!private *
org.apache.catalina.servlets.*.*(..)"/>
                <advice name="trace"
type="around" bind-to="nonPrivate"/>
            </aspect>
        </system>
    </aspectwerkz>
```

Lets package a new *weave n drop* aspect jar file:

```
cd C:\aw\tomcat
jar -cf ..\aspect-system.jar *
```

You will end up in having a C:\aw\aspect-system.jar file ready to use.

Copy this file to C:\aw\jakarta-tomcat-5.0.25\common\lib which contains some regular jar for Tomcat.

*Startup Tomcat.*

Now we have both DemoAspect of each system at work:

```
DemoAspect[13859719] -->
METHOD_EXECUTION:filters.ExampleFilter.doFilter
  DemoAspect[13859719] -->
METHOD_EXECUTION>HelloWorldExample.doGet
  DemoAspect[13859719] <--
  DemoAspect[13859719] -->
METHOD_EXECUTION:filters.ExampleFilter.toString
  DemoAspect[13859719] <--
DemoAspect[13859719] <--
DemoAspect[16094127] -->
METHOD_EXECUTION:org.apache.catalina.servlets.DefaultServlet.doGet
  DemoAspect[16094127] -->
METHOD_EXECUTION:org.apache.catalina.servlets.DefaultServlet.serveResource
  DemoAspect[16094127] -->
METHOD_EXECUTION:org.apache.catalina.servlets.DefaultServlet.getRelativePath
  DemoAspect[16094127] -->
METHOD_EXECUTION:org.apache.catalina.servlets.DefaultServlet.doGet
```

## Conclusion

We have:

1. learned how to package and deploy our aspect along our web application
2. understood the consequence of doing so
3. understood a way to do it with a self contained jar file dropped in the web application libs
4. understood the difference with an Aspect jar that would belong to the system classpath
5. practice class load time integration using the command line tool

To go on in practicing, you can check that the AOP container will refuse deploying the same system id within the same class loader hierarchy, or you can find some interesting pointcut tied to Tomcat and Servlet 2.3 API, or try to change the Aspect to have it a bit more usefull.