

2

Deployment Options

It's exciting to think that you can deploy and run components in Geronimo after only four months of development. You'd be amazed at all the infrastructure that has to be put into place to get to that point; advanced classloaders, xml-to-object marshallers, directory scanners, archive readers, deployment planners, dependency managers, component validation. The concept of a Container to manage the component's lifecycle at run-time is trivial in comparison to all the work that has to be done to get to the component into the system in the first place.

The Geronimo team has created a plugin for Maven that greatly eases the deployment process, mostly by hiding certain details that you really don't need to be concerned about. Chapters 3 and beyond use this plugin for everything from compiling to deploying to starting the server. While all this is well and good, there always seems to come a time when you just need to do it the hard way. When the need arises, the only way to get a custom solution in place is to be intimately familiar with all these processes.

This chapter goes over in detail all the hocus-pocus and hand-waving required to develop, deploy, and run an app in Geronimo without the plugin for Maven. The plain command-line is covered, as is Ant. Maven itself is also covered; sometimes you may need to wire-up your own plugin and will need to know the details behind the magic Geronimo plugin.

We are going to use the same example for each of the three approaches so you get the clearest idea of what the steps are and are not distracted by sample code—that's goal of chapter's 3 and 4.

The Short Version

Geronimo has a deploy tool in an executable jar that is invoked just like `server.jar` itself. The command goes like this:

```
| C:\geronimo-1.0-M42> java -jar bin\deployer.jar --install --module someWebapp.war
```

Or

```
| C:\geronimo-1.0-M42> java -jar bin\deployer.jar --install --module someEjbs.jar
```

Or even

```
| C:\geronimo-1.0-M42> java -jar bin\deployer.jar --install --module myConnector.rar
```

And eventually

```
| C:\geronimo-1.0-M42> java -jar bin\deployer.jar --install --module someJ2eeApp.ear
```

With these commands you can deploy EJB Jar files, Servlet War files, Connector Rar files, and Enterprise Ear files. The deploy tool is smart enough to figure out what to do. You can also deploy Geronimo-specific components like GBeans, which we won't go into just yet.

There is one fairly major limitation to the deploy tool; you must deploy with the server stopped.

Yes, it is terrible to have to stop the server to deploy and yes, we are working on it. In fact, it may have already changed. The issue is that the config-store, the place where everything goes after deployment, is designed to be used by one process at a time. In other words, the deploy tool and the server cannot both be modifying it. To make hot deployment work, we need to wire up the deploy tool to talk to the running server and send configured applications directly over a socket. This part is easy as pie, but anytime you allow people to deploy things over a network, you really need to secure it. Security is really the part that is standing in the way of hot deploy. We take security very seriously and exposing an insecure deploy tool is not something we are willing to do.

Look Ma' No Tools

This first approach to developing and running apps in Geronimo is the most grueling, but it's also the most revealing. It involves nothing but you, the command prompt, the JSDK 1.4 and Geronimo. This is the definition of roughing it for computer geeks.

How do I do that?

First things first, we must do the obligatory check for your JDK. This is very basic, but I can't tell you the amount of times I've taught a class where half the computers don't have Java installed in their system PATH. Crack open the command prompt and type the following command.

```
| C:\> java -version
```

This should print the version information to the console all nice and pretty as shown in Example 2-1.

Example 2-1. A healthy java setup

```
java version "1.4.2_03"  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_03-b01)  
Java HotSpot(TM) Client VM (build 1.4.2_03-b01, mixed mode)
```

However, if the output looked something like Example 2-2, then Java is not correctly installed in your system PATH variable and must be added before you can continue. If you installed Java in the default location, then add `C:\jdk1.4.2_03\bin` to the path.

The exact directory name will be slightly different for your particular version of Java 1.4, so make sure to check it first.

Example 2-2. A bad PATH variable

```
'java' is not recognized as an internal or external command,  
operable program or batch file.
```

Setting the GERONIMO_HOME variable

We need to do several things that reference the path to where Geronimo is installed. To make this easier and less error prone, we can set up a GERONIMO_HOME variable to point to that path.

Geronimo itself does not need you set a GERONIMO_HOME variable. We, the developers, use executable jars for everything and leverage a clever technique to discover where the jar is installed. This variable is just something to make life on the command line more bearable.

Assuming you installed Geronimo to the C:\geronimo-1.0-M42 directory, then execute the following commands. Adjust the path as necessary.

```
C:\> set GERONIMO_HOME=C:\geronimo-1.0-M42  
  
C:\> dir %GERONIMO_HOME%  
Volume in drive C has no label.  
Volume Serial Number is 54E7-6152  
  
Directory of C:\geronimo-1.0-M42  
  
02/06/2004  03:07 AM    <DIR>      .  
02/06/2004  03:07 AM    <DIR>      ..  
02/06/2004  03:07 AM    <DIR>      bin  
02/06/2004  04:26 AM    <DIR>      config-store  
02/06/2004  03:07 AM    <DIR>      lib  
02/06/2004  03:07 AM    <DIR>      repository  
02/06/2004  03:07 AM    <DIR>      schema  
02/06/2004  03:07 AM    <DIR>      var  
                0 File(s)          0 bytes  
                8 Dir(s)  24,774,176,768 bytes free
```

If you didn't get the output shown, or at least something similar, the path used to set GERONIMO_HOME is not correct. Double check that and try again.

Adding the J2EE jar to the CLASSPATH

We need the J2EE libraries before we can compile anything, so we have a little more setup work to do before we are off and running. Geronimo ships with a copy of these libraries for everyone to use. Execute the following command to add it to your classpath. If you setup your GERONIMO_HOME variable correctly, you shouldn't need to change this command at all.

```
C:\>set CLASSPATH=%GERONIMO_HOME%\repository\geronimo-spec\jars\geronimo-spec-j2ee-  
1.4.jar  
  
C:\>dir %CLASSPATH%  
Volume in drive C has no label.  
Volume Serial Number is 54E7-6152
```

```

Directory of C:\geronimo-1.0-M42\repository\geronimo-spec\jars
02/06/2004  03:07 AM                273,787 geronimo-spec-j2ee-1.4.jar
                1 File(s)                273,787 bytes
                0 Dir(s)  24,773,160,960 bytes free

```

Again, if you didn't get this output, than something is probably wrong with your GERONIMO_HOME variable or you didn't type the example exactly as shown above.

Notice the J2EE jar is in a directory called "repository." The repository is where you put libraries that are needed by your applications. Just adding libraries to this directory does not make them globally available to your applications. Other app servers, like Tomcat, will do this. Geronimo does not.

Geronimo wants to give you the ability to share libraries without forcing all of your applications to have to agree on which *versions* of those libraries to use. Each application states the version of each library it wants to use, which allows for happy-sharing-reuse-love without upgrade-hell.

Creating a Web project

Finally, we have the environment setup out of the way and can get on with the show—you have to love life on the command line. We still need someplace to write our code and the obligatory WEB-INF directory structure. The following command should take care of that nicely.

```
C:\> mkdir C:\exercise-2-1\WEB-INF\classes
```

Ok, we're rolling. Three directories in one command, not bad. Move right into the `exercise-2-1` directory and create the JSP file shown in Example 2-3.

Example 2-3. `viewclass.jsp`

```

<%@ page import="java.lang.reflect.Method" %>
<%
    String className = request.getParameter("class");
    if (className == null) {
        className = "javax.servlet.http.HttpServletRequest";
    }

    Class  clazz      = Class.forName(className);
    Method[] methods  = clazz.getDeclaredMethods();
    Class  superclass = clazz.getSuperclass();
    Class[] interfaces = clazz.getInterfaces();
%>
<html>
<head><title>Class Viewer JSP</title></head>
<body>
    <b><%=clazz.getName()%></b><br>

    <br><b>Methods</b><br>
    <% for (int i=0; i < methods.length; i++){ %>
        <%=methods[i]%><br>
    <% } %>

```

```

        <br><b>Extends:</b><br>
        <%=superclass%><br>

        <br><b>Implements:</b><br>
        <% for (int i=0; i < interfaces.length; i++){ %>
           <%=interfaces[i]%><br>
        <% } %>
    </body>
</html>

```

This is our example; a nice change of pace from Hello World. The JSP accepts a class name as a parameter, and then uses reflection to print all the class information. If no class name is specified, `javax.servlet.http.HttpServletRequest` is used by default.

Let's create a Servlet version of our JSP so we have something to compile and add to the `web.xml` we make later. In your favorite Java editor, create the Servlet shown in Example 2-4.

Example 2-4. ClassViewerServlet.java

```

import java.io.*;
import java.lang.reflect.Method;
import javax.servlet.ServletException;
import javax.servlet.http.*;

public class ClassViewerServlet extends HttpServlet {

    protected void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();

        String className = req.getParameter("class");
        if (className == null) {
            className = "junit.framework.TestCase";
        }

        Class clazz = null;
        try {
            clazz = Class.forName(className);
        } catch (ClassNotFoundException e) {
            throw new ServletException(e);
        }

        Method[] methods = clazz.getDeclaredMethods();
        Class superclass = clazz.getSuperclass();
        Class[] interfaces = clazz.getInterfaces();

        out.print("<html>");
        out.print("<head><title>Class Viewer Servlet</title></head>");
        out.print("<body>");
        out.print("<b>" + clazz.getName() + "</b><br>");

        out.print("<br><b>Methods</b><br>");
        for (int i=0; i < methods.length; i++){
            out.print(methods[i] + "<br>");
        }
    }
}

```

```

        out.print("<br><b>Extends:</b><br>");
        out.print(superclass+"<br>");

        out.print("<br><b>Implements:</b><br>");
        for (int i=0; i < interfaces.length; i++){
            out.print(interfaces[i]+"<br>");
        }
        out.print("</body>");
        out.print("</html>");
    }
}

```

The Servlet version of our JSP functions exactly the same way. The only thing different is that the default class that is displayed is `junit.framework.TestCase`. This gives us a nice excuse to have to deal with third-party libraries in our example as well. In this case, it's JUnit.

The JUnit library is already available in a jar sitting in the Geronimo repository. As we see later, the Geronimo-specific deployment descriptor provides us with a clever mechanism to add that jar to our webapp without actually to package it with the war file we create.

Creating deployment descriptors

I think I can say quite confidently that everyone's favorite part of writing J2EE apps is creating deployment descriptors. Few things are more rewarding than starting with a blank document and typing in all the tags from memory. Vendor-specific deployment descriptors are a rare joy as well.

Okay, so maybe that's laying on the sarcasm a little too thick. But really, this is the section I didn't want to write. Creating deployment descriptors is the job of a tool and not something even the brightest person can do with out a lot of copy-paste code reuse.

Additionally, we (i.e. the Geronimo team) can't guarantee that our own vendor-specific set of deployment descriptors are going to remain constant and unchanged. My gut tells me that by the time you read this, the Geronimo-specific deployment descriptors will have already been reworked a few times over.

Never the less, deployment descriptors are a mandatory part of the J2EE specs, so onward we go!

In your favorite XML editor (if you have one), create the 'web.xml' file as shown in Example 2-5. Keep in mind that this file must be located in the WEB-INF directory of our application.

Example 2-5. WEB-INF/web.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">

  <servlet>
    <servlet-name>ClassViewer</servlet-name>
    <servlet-class>ClassViewerServlet</servlet-class>

```

```

</servlet>

<servlet-mapping>
  <servlet-name>ClassViewer</servlet-name>
  <url-pattern>/ClassViewer</url-pattern>
</servlet-mapping>
</web-app>

```

Make sure there are no blank lines before the `<?xml...?>` declaration, otherwise the file will not parse. This is a standard XML rule, but one frequently broken.

As long as we have our XML editor open, create the `geronimo-jetty.xml` file shown in Example 2-6. This file should sit in the `WEB-INF` directory right next to the `web.xml` file we just created.

Example 2-6. WEB-INF/geronimo-jetty.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="http://geronimo.apache.org/xml/ns/web/jetty"
  configId="my/first/webapp"
  parentId="org/apache/geronimo/Server"
  >

  <dependency>
    <uri>junit/jars/junit-3.8.jar</uri>
  </dependency>

  <context-root>/example</context-root>
  <context-priority-classloader>>false</context-priority-classloader>
</web-app>

```

Make a note of that `configId` attribute. We will need to know that later in order to start the server.

As always, the structure of this file can change. Make sure you check <http://www.oreilly.com/bookurl/errata> for any updates since this book was published.

Packaging our application

The finale to the creation process is to compile and package our application. We already set up our `CLASSPATH`, so this task is straightforward. Using a command shell in which your `CLASSPATH` is set, compile the Servlet, then check the `WEB-INF/classes` directory to make sure everything ended up in the right place.

```

C:\exercise-2-1> javac ClassViewerServlet.java -d WEB-INF\classes

C:\exercise-2-1> dir WEB-INF\classes
Volume in drive C has no label.
Volume Serial Number is 54E7-6152

Directory of C:\exercise-2-1\WEB-INF\classes

```

```

02/06/2004 03:18 AM <DIR>      .
02/06/2004 03:18 AM <DIR>      ..
02/06/2004 03:18 AM                1,953 ClassViewerServlet.class
                1 File(s)          1,953 bytes
                2 Dir(s)  24,767,270,912 bytes free

```

Last but not least, jar this puppy up! Use these two commands from within the exercise-2-1 directory. The first command creates the jar, the second command lists the contents of the new jar. Make sure the output of the second command is the same as shown below.

```

C:\exercise-2-1> jar cf classviewer.war WEB-INF viewclass.jsp

C:\exercise-2-1> jar tf classviewer.war
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/ClassViewerServlet.class
WEB-INF/geronimo-jetty.xml
WEB-INF/web.xml
viewclass.jsp

```

Deploying and running the application

All that remains now is to deploy our war file. If Geronimo is started, you need to kill it before you can deploy. See the sidebar for details on that.

Deployment is quite easy. The deploy tool is an executable jar that has all the information it needs to run. You don't have to set up any special classpath or other nonsense to use it. With the server stopped, execute the following command.

```

C:\exercise-2-1> java -jar %GERONIMO_HOME%\bin\deployer.jar --install --module
classviewer.war

```

The command should be all on one line and will not print any of output. That's really it as far as deployment goes. Pretty easy, right?

The next step is the big one. As Ed McMahon would say to The Great Carnac, I hold in my prompt the last command*. Meditate on your command prompt for a spell then issue the last command.

```

C:\exercise-2-1>java -jar %GERONIMO_HOME%\bin\server.jar my/first/webapp
04:52:12,934 INFO  [Kernel] Starting boot
04:52:13,004 INFO  [MBeanServerFactory] Created MBeanServer with ID:
1dfc547:fc9cacdcc2:-8000:Pepe:1
04:52:13,164 INFO  [Kernel] Booted
04:52:13,194 INFO  [ConfigurationManager] Loaded Configuration
geronimo.config:name="org/apache/geronimo/System"
04:52:13,355 INFO  [Configuration] Started configuration org/apache/geronimo/System
04:52:13,365 INFO  [ReadOnlyRepository] Repository root is file:/C:/geronimo-1.0-
M42/repository/

```

* Ok, so I twisted things a bit. And references to the Johnny Carson Show are maybe a little lost on most people nowadays. I suppose now is not the time to mention I like the Carol Burnett Show too.

```
04:52:13,415 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="my/first/webapp"
04:52:13,425 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="org/apache/geronimo/Server"
04:52:13,935 INFO [Configuration] Started configuration org/apache/geronimo/Server
04:52:13,935 INFO [HttpServer] Starting Jetty/5.0.RC0
04:52:13,945 INFO [HttpServer] Started org.mortbay.jetty.Server@296f76
04:52:14,006 INFO [SocketListener] Started SocketListener on 0.0.0.0:8080
04:52:14,226 INFO [Configuration] Started configuration my/first/webapp
04:52:14,236 INFO [Credential] Checking Resource aliases
04:52:16,048 INFO [HttpContext] Started
WebApplicationContext[/example,file:/C:/geronimo-1.0-M42/config-store/9/war/]
```

Give Geronimo a couple seconds to start, then open your browser to <http://localhost:8080/example/viewclass.jsp>. You should see our little JSP doing its thing as shown in Figure 2-1.

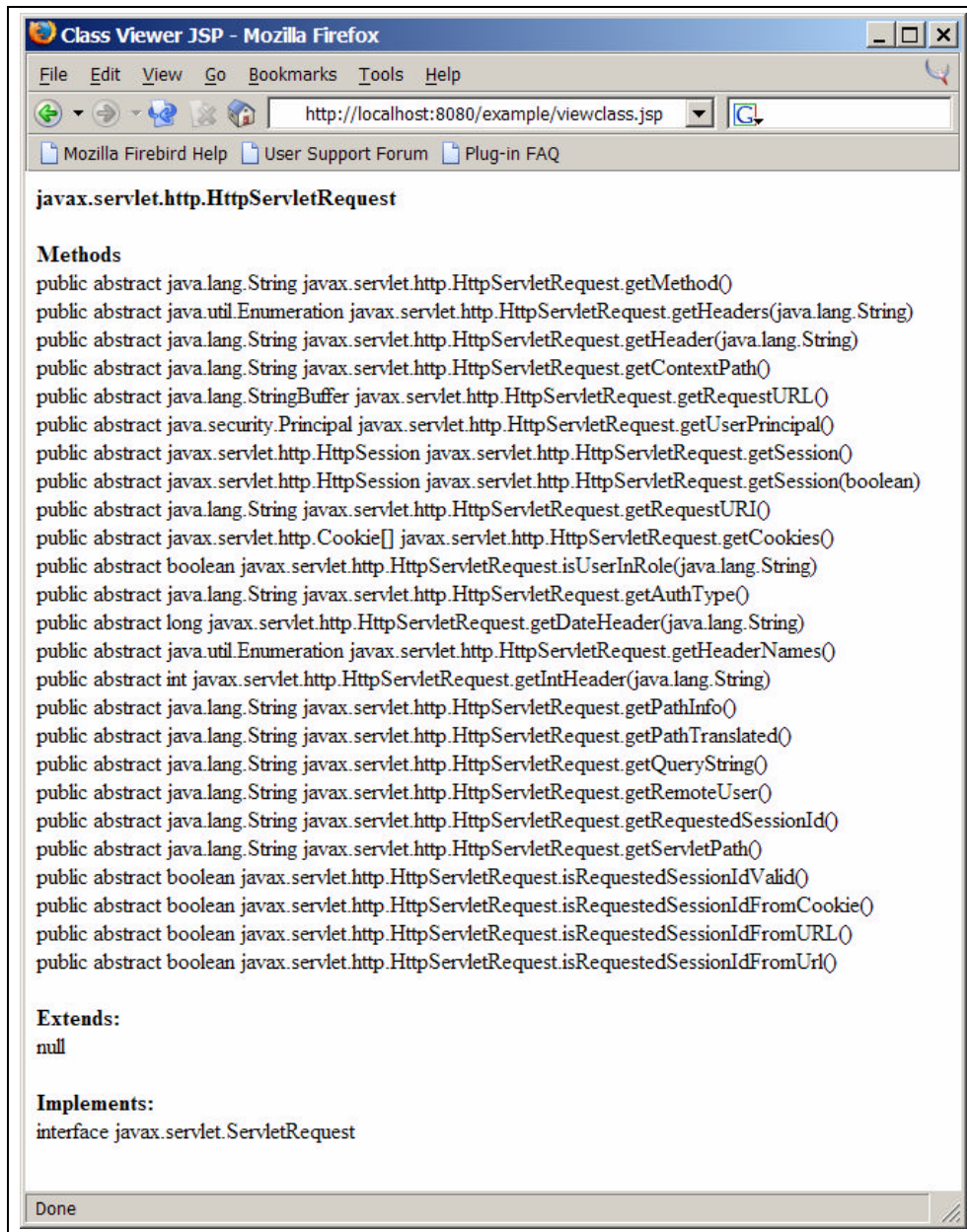


Figure 2-1. A running JSP

Let's also give our Servlet version a whirl and see how it behaves. Point your browser to <http://localhost:8080/example/ClassViewer> and you should see the content displayed in Figure 2-2.

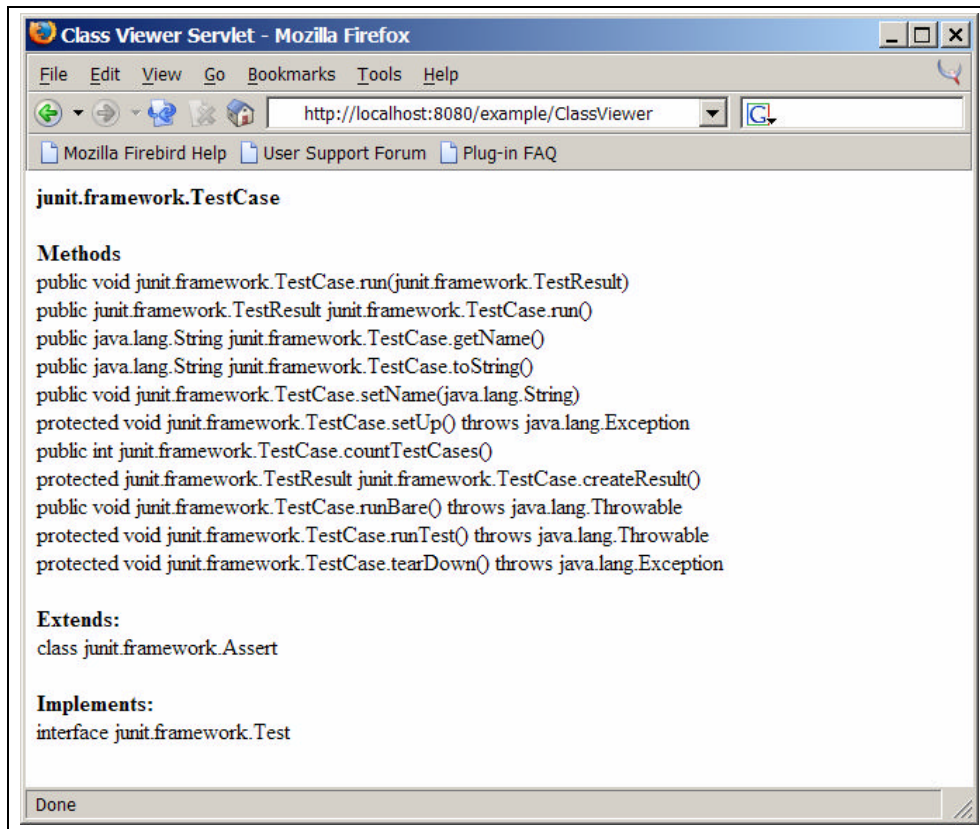


Figure 2-2. A running Servlet

What just happened?

Congratulations! You've just created, packaged, deployed, and ran your first JSP and Servlet in Geronimo. You created a plain war file packaged with a `geronimo-jetty.xml` that had text like the following inside it.

```
<web-app xmlns="http://geronimo.apache.org/xml/ns/web/jetty"
  configId="my/first/webapp"
```

Remember that `configId` that I said would be important? Well, that's the same name we used to start the server.

```
C:\exercise-2-1>java -jar %GERONIMO_HOME%\bin\server.jar my/first/webapp
```

You see how those two match up. As mentioned in chapter 1, that ID is not a file path, but a URI. If it were a file path, it would have to use backslashes to work in a Windows filesystem, but this not the case. You can give your application any kind of URI you want. Just remember what it is as you need it when starting Geronimo.

For fun, you can try passing parameters into the JSP to get it to display other classes. Type the following URLs in your browser and see what happens:

- ? <http://localhost:8080/example/viewclass.jsp?class=java.util.Set>
- ? <http://localhost:8080/example/viewclass.jsp?class=java.lang.reflect.Method>

? <http://localhost:8080/example/viewclass.jsp?class=javax.swing.AbstractButton>

What about...

Most people are used to using what is called an exploded war, rather than a war file itself. For those not familiar with the term, an exploded war is a regular war file unzipped; its contents are “exploded” into the webapp’s directory.

Geronimo does explode wars into the config-store after they are deployed, and it is possible to play with the files in the exploded directory while the server is started. Mucking around in the config-store, however, is strongly discouraged. You are undermining Geronimo’s ability to offer you stability. Now that you’ve been warned, let’s do it anyway.

If you look in the config-store directory, there is a file called index.properties that lists the URIs of all the applications and components deployed in Geronimo.

```
C:\exercise-2-1> more %GERONIMO_HOME%\config-store\index.properties
#Wed May 19 04:16:14 CDT 2004
org/apache/geronimo/DebugConsole=5
org/apache/geronimo/System=3
org/apache/geronimo/DeployerSystem=1
org/apache/geronimo/Server=4
org/apache/geronimo/J2EEDeployer=2
my/first/webapp=9
```

As you see, the webapp we just created (my/first/webapp) has the number 9 after it. This means that the active version of our app is in the 9th directory in the config-store. If we list the contents of that directory with the Windows `tree` command, we see an exploded version of our war file as shown in Example 2-7.

Example 2-7. Our app’s location in the config-store

```
C:\exercise-2-1> tree /F %GERONIMO_HOME%\config-store\9
Folder PATH listing
Volume serial number is 71FAE346 54E7:6152
C:\GERONIMO-1.0-M42\CONFIG-STORE\9
+---META-INF
|       config.ser
|
+---war
|       viewclass.jsp
|
+---WEB-INF
|       geronimo-jetty.xml
|       web.xml
|
+---classes
       ClassViewerServlet.class
```

If you edited the viewclass.jsp you see in this directory and hit Refresh on your browser, you would in fact see the new version of the JSP. So, although it’s discouraged, sometimes you’re willing to break the rules when playing around. Never do this on a live server.

Deploying from the Internet

Before we move on to life with Ant, there is one neat thing I have to show you. When you read the title of this section, you're probably thinking that it teaches you how to be on the internet and deploy to a remote server. Well, reverse that thinking.

Geronimo can actually deploy components straight from the Internet. You can build a library of webapps and publish those on a web server; then others can point the Geronimo deploy tool directly at the applications they want to install and run.

How do I do that?

Geronimo has a JMX DebugConsole that ships with every release. Let's imagine, though, that you really want the most current version and don't want to wait for the next release. We can point the deploy tool to the URL of the application and ask for it to be installed.

Backup your index.properties file

Before we attempt to install the new version though, it's a good idea to backup your index.properties file in the config-store. If something goes wrong in the new version, or you simply don't like it, you can restore the index.properties file and be right back to the configuration you started with. The following command creates you a copy you can use for restoring.

```
| C:\geronimo-1.0-M42\config-store> copy index.properties index.20040206
```

Run the deploy tool

With that, we created a backup with the date as the extension—always good to be organized. Now, let's try and install the new JMX DebugConsole.

Browse over to <http://www.ibiblio.org/maven/geronimo/wars/> with your favorite web browser and find the full URL to the latest greatest version of the geronimo-jmxdebug war file. Let's imagine that there is a *geronimo-jmxdebug-1.0-SNAPSHOT.war* file on Ibiblio we can use[†].

OK, now this is really cool. Break out your command prompt and execute the following command.

```
| C:\geronimo-1.0-M42> java -jar bin\deployer.jar --install --module  
| http://www.ibiblio.org/maven/geronimo/wars/geronimo-jmxdebug-1.0-SNAPSHOT.war
```

The above command should all be on one line.

Start the debug console

Now that we have the latest version of the DebugConsole installed, let's start it up. If you have a instance of the server running already, make sure you kill it before executing the command below.

```
| C:\geronimoo-1.0-M42> java -jar bin\server.jar org/apache/geronimo/DebugConsole
```

[†] The word "snapshot" is a maven-ism for the most recent build of code not yet released

Once the output of that command slows down, pop open your trusty browser and direct it to <http://localhost:8080/debug-tool/> where you should see a page like the one in Figure 2-3.



Figure 2-3. The newly deployed DebugConsole

What just happened?

Geronimo just downloaded and reinstalled the JMX DebugConsole into your config-store. The old version of it is still there, but the newest version of it is now the one that is running. Pretty neat, huh!

The concept for rolling back a configuration is a bit of Geronimo innovation. It has certainly existed in many other types of systems, but hasn't really manifested itself in open source J2EE till now.

We can all thank Maven for inspiring the idea of downloading from the Internet. At some point it will be possible to specify a list of online repositories and to tell Geronimo the relative URI of the one you want installed into the config-store. I can't wait to get that in there. Imagine having a catalog of online components you can just point to and have Geronimo download and install them for you.

For Linux users, that may bring images of RPMs into mind. However, it's really much more like the apt-get tool as Geronimo will also download and install dependencies of the component too. There won't be any of the library-chasing normally associated with RPM-style packages.

Why didn't it work?

If Geronimo failed to download the war file, you are probably behind a firewall that is stopping any access to the web. Try using your web browser to download the war file onto your machine, then run the deploy command on that war file. Not as neat, but you will still be able to play around with the rolling back to a previous configuration.

```
C:\geronimo-1.0-M42> java -jar bin\deployer.jar --install --module geronimo-jmxdebug-1.0-SNAPSHOT.war
```

What about...

To restore your Geronimo installation to its previous state, you only need to restore the previous index file. Kill the running Geronimo server, if there is one, and execute this command.

```
C:\geronimo-1.0-M42\config-store> move index.properties index.rolledback
C:\geronimo-1.0-M42\config-store> move index.20040206 index.properties
```

If you start our server with the same command you used a moment ago, you should get the previous version of the DebugConsole as illustrated in Figure 2-4.

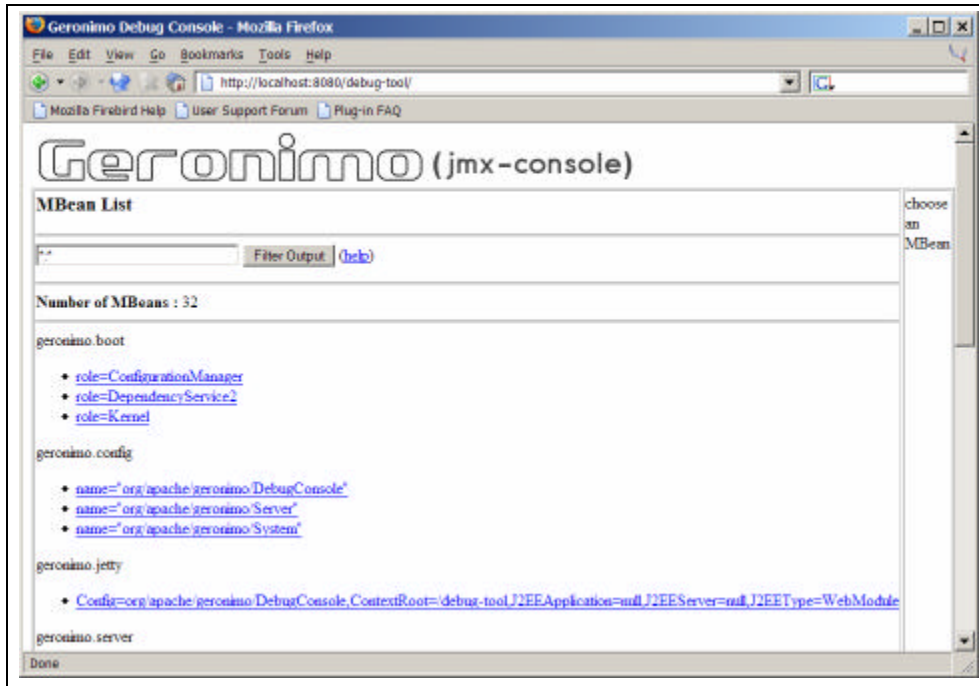


Figure 2-4. Restored DebugConsole

We plan on automating the backup and rollback process at some point, but for now you have to do it by hand.

Deploying with Ant

Running commands from the command line is fine, but most people will be using something like Ant or Maven to do all the deployment. The rest of this book uses Maven for development and deployment of J2EE applications, so let's focus on Ant for a moment before getting into all that.

How do I do that?

First things first, we need to get Ant installed on your machine. Visit <http://ant.apache.org> and download the latest binary release. Once that is downloaded, just unzip it directly to your C: drive. The contents of that directory should look similar to that shown in Example 2-9.

Example 2-9. Your Ant installation

```
C:\apache-ant-1.6.0> dir
Volume in drive C has no label.
Volume Serial Number is 54E7-6152

Directory of C:\apache-ant-1.6.0

01/12/2004  07:04 AM    <DIR>          .
01/12/2004  07:04 AM    <DIR>          ..
```

```

01/12/2004 07:02 AM <DIR> bin
01/12/2004 07:04 AM <DIR> docs
01/12/2004 07:04 AM <DIR> etc
12/18/2003 03:12 PM 17,191 KEYS
01/12/2004 07:04 AM <DIR> lib
12/18/2003 03:12 PM 2,717 LICENSE
12/18/2003 03:12 PM 3,356 LICENSE.dom
12/18/2003 03:12 PM 677 LICENSE.sax
12/18/2003 03:12 PM 2,698 LICENSE.xerces
12/18/2003 03:12 PM 2,657 README
12/18/2003 03:12 PM 18,478 welcome.html
12/18/2003 03:12 PM 102,914 WHATSNEW
      8 File(s) 150,688 bytes
      6 Dir(s) 24,983,121,920 bytes free

```

The last step in our Ant setup is to get the bin directory in our system path. Given the ant directory of C:\apache-ant-1.6.0, the following command does the trick.

```
C:\> set PATH=%PATH%;C:\apache-ant-1.6.0\bin
```

Okay, we are done with our Ant setup. Let's try and invoke the ant command from the same command prompt to verify we did everything correctly.

```
C:\> ant -version
Apache Ant version 1.6.0 compiled on December 18 2003
```

That should yield output similar that showed above. If you got something like the error shown below than you didn't use the right path when setting the PATH variable. Double check the path, remembering that you must include the bin directory in the path.

```
C:\> ant -version
'ant' is not recognized as an internal or external command,
operable program or batch file.
```

Creating a project layout

Ant is a good tool for freeing you from typing long commands, like the ones in the previous section. A few commands are still required to set up a directory structure for development. Ant doesn't have the ability to create a project automatically. To create a project layout in Windows, run these two commands.

```
C:\> mkdir exercise-2-2\src\webapp\WEB-INF
C:\> mkdir exercise-2-2\src\java
```

The mkdir command in OS X, Linux or Cygwin works a little differently. On those systems you must include the -p option:

```
~$ mkdir -p exercise-2-2/src/webapp/WEB-INF
~$ mkdir -p exercise-2-2/src/java
```

From here on out the commands are pretty much OS-neutral, and shouldn't need to be modified. That's a nice reason to use a build tool like ant or maven.

Reusing our source files

We are going to reuse our same example files from the previous section to make things easier. Create or copy those files to the following directories.

? viewclass.jsp (Example 2-3) to directory C:\exercise-2-2\src\webapp

- ? ClassViewerServlet.java (Example 2-4) to directory C:\exercise-2-2\src\java
- ? web.xml (Example 2-5) to directory C:\exercise-2-2\src\webapp\WEB-INF
- ? geronimo-jetty.xml (Example 2-6) to directory C:\exercise-2-2\src\webapp\WEB-INF

When that is done, you should have a file structure that looks like the one in Example 2-8.

Example 2-8. Our project so far

```
C:\EXERCISE-2-2
+---src
|   +---java
|       |   ClassViewerServlet.java
|       |
|       +---webapp
|           |   classviewer.jsp
|           |
|           +---WEB-INF
|               |   geronimo-jetty.xml
|               |   web.xml
```

Making the Ant build files

Ant uses an xml file to know what commands to run to build your project. This xml file, called build.xml, usually gets pretty large, so most sane people put project specific information in a separate file, called build.properties. This successfully gets the variant information out of the build.xml file, but doesn't actually eliminate the need to copy the same build.xml file into every project that wishes to build in a similar manner.

The last step in our project setup is to make the build.xml and build.properties files. At the root of the project in the C:\exercise-2-2 directory, create the build.xml file shown in Example 2-10.

Example 2-10. Ant build.xml file

```
<project name="ClassViewer" default="dist" basedir=".">

  <!-- configurable properties for this build -->
  <property file="build.properties" />

  <!-- set global properties for this build -->
  <property name="source" location="src"/>
  <property name="source.java" location="${source}/java"/>
  <property name="source.web" location="${source}/webapp"/>

  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <path id="project.classpath">
    <pathelement location="${geronimo.home}/repository/geronimo-spec/jars/geronimo-spec-j2ee-1.4.jar"/>
  </path>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${dist}"/>
  </target>
</project>
```

```

</target>

<target name="clean" description="delete build and dist directories" >
  <delete dir="${build}"/>
  <delete dir="${dist}"/>
</target>

<target name="compile" description="compile the source" >
  <javac srcdir="${source.java}" destdir="${build}"
    classpathref="project.classpath"/>
</target>

<target name="jar" depends="init,compile" description="jar the class files" >
  <jar jarfile="${dist}/${project.name}.jar" basedir="${build}"/>
</target>

<target name="war" depends="jar" description="create the war file" >
  <war destfile="${dist}/${project.name}.war"
    update="no" index="true"
    webxml="${source.web}/WEB-INF/web.xml">
    <lib dir="${dist}">
      <include name="${project.name}.jar"/>
    </lib>
    <zipfileset dir="${source.web}"/>
  </war>
</target>

<target name="dist" depends="jar,war" description="generate the distribution" />

<!-- geronimo specific targets -->
<target name="deploy" depends="dist" description="deploy the war file">
  <java jar="${geronimo.home}/bin/deployer.jar"
    fork="true" failonerror="true">
    <arg value="--install"/>
    <arg value="--module"/>
    <arg value="${dist}/${project.name}.war"/>
  </java>
</target>

<target name="start" description="start geronimo">
  <java jar="${geronimo.home}/bin/server.jar"
    fork="true" failonerror="true"
    maxmemory="256m" dir="${geronimo.home}">
    <jvmarg value="-ea"/>
    <arg value="${project.uri}"/>
  </java>
</target>
</project>

```

Assuming you have Geronimo installed in the C:\geronimo-1.0-M42 directory, the build.properties file in Example 2-11 will do the trick. Otherwise, adjust the path to Geronimo as needed. Make sure you create this file right at the root of the project in the C:\exercise-2-2 directory.

Example 2-11. Ant build.properties file

```

# adjust this to point to your geronimo installation
geronimo.home = C:\geronimo-1.0-M42

```

```
# adjust these on a project by project basis
project.name = classviewer
project.uri = my/first/webapp
```

Building, deploying, and running

Alright, setup is over; time for the real fun. If you didn't understand the Ant script, no problem, you can get a full list of the commands available with the `-projecthelp` command[‡]. Let's give it a try.

```
C:\exercise-2-2> ant -projecthelp
Buildfile: build.xml

Main targets:

clean    delete build and dist directories
compile  compile the source
deploy   deploy the war file in gernimo
dist     generate the distribution
jar      jar the class files
start    start gernimo
war      create the war file
Default target: dist
```

Notice that Ant refers to the options that can come after the ant executable as *targets*. Our build script provides us with exactly seven targets, with the default being the dist target. You can use as many of the targets together as you like. Also, some targets depend on others, so running such a target will cause ant to first run all the dependent targets.

Now that we have Ant working, we can compile, package and deploy our project into Geronimo with just the one ant command below. Make sure you are inside the C:\exercise-2-2 directory when you run it.

```
C:\exercise-2-2> ant deploy
Buildfile: build.xml

init:
  [mkdir] Created dir: C:\exercise-2-2\build
  [mkdir] Created dir: C:\exercise-2-2\dist

compile:
  [javac] Compiling 1 source file to C:\exercise-2-2\build

jar:
  [jar] Building jar: C:\exercise-2-2\dist\classviewer.jar

war:
  [war] Building war: C:\exercise-2-2\dist\classviewer.war
  [war] Warning: selected war files include a WEB-INF/web.xml which will be
ignored (please use webxml attribute to war task)

dist:
```

[‡] For Lord of the Rings fans, that's one command to rule them all.

```

deploy:

BUILD SUCCESSFUL
Total time: 5 seconds

```

Believe it or not, that one command did the all the work of the previous section. Pretty slick. The only work we really had to do was in setup, which only happens once per project. With another command, we can start our webapp in Geronimo.

```

C:\exercise-2-2> ant start
Buildfile: build.xml

start:
 [java] 18:27:50,417 INFO [Kernel] Starting boot
 [java] 18:27:50,497 INFO [MBeanServerFactory] Created MBeanServer with ID:
1dfc547:fcecd6facd:-8000:Pepe:1
 [java] 18:27:50,658 INFO [Kernel] Booted
 [java] 18:27:50,688 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="org/apache/geronimo/System"
 [java] 18:27:50,838 INFO [Configuration] Started configuration
org/apache/geronimo/System
 [java] 18:27:50,858 INFO [ReadOnlyRepository] Repository root is
file:/C:/geronimo-1.0-M42/repository/
 [java] 18:27:50,878 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="my/first/webapp"
 [java] 18:27:50,888 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="org/apache/geronimo/Server"
 [java] 18:27:51,369 INFO [Configuration] Started configuration
org/apache/geronimo/Server
 [java] 18:27:51,369 INFO [HttpServer] Starting Jetty/5.0.RC0
 [java] 18:27:51,379 INFO [HttpServer] Started org.mortbay.jetty.Server@5b8827
 [java] 18:27:51,379 INFO [SocketListener] Started SocketListener on
0.0.0.0:8080
 [java] 18:27:51,599 INFO [Configuration] Started configuration
my/first/webapp
 [java] 18:27:51,619 INFO [Credential] Checking Resource aliases
 [java] 18:27:52,200 INFO [HttpContext] Started
WebApplicationContext[/example,file:/C:/geronimo-1.0-M42/config-store/10/war/]

```

What just happened?

There you have it. With the first command we built our code, created a jar file and war file then deployed it directly into Geronimo. With the second command, we used ant to start our app in Geronimo.

If you point your browser to <http://localhost:8080/example/ClassViewer> you should see the content displayed in Figure 2-2 just as in the previous section.

The war file was compiled into the dist directory. If we check it out with the jar command, we see the following contents.

```

C:\exercise-2-2> jar tf dist\classviewer.war
META-INF/
META-INF/MANIFEST.MF

```



```
|_| |_\/_|\_\/_|\_|_| v. 1.0-rc2
```

If you don't get the maven version information output to the screen similar to above, then maven is not completely setup. Run through the setups in the Installing Maven section of chapter one and double check that everything is ok. Most likely, your system PATH variable is not setup.

Creating a project layout

We will be using the exact same project structure and source files as in the previous section. To get set up for this section, copy everything in the C:\exercise-2-2 directory to a new directory called C:\exercise-2-3. Go ahead and delete the dist and build directories as we won't need them.

If you didn't do the previous section, no problem, just run these commands.

```
C:\> mkdir exercise-2-3\src\webapp\WEB-INF
C:\> mkdir exercise-2-3\src\java
```

On OS X, Linux or Cygwin systems you must include the -p option:

```
~$ mkdir -p exercise-2-3/src/webapp/WEB-INF
~$ mkdir -p exercise-2-3/src/java
```

Then create or copy those files to the following directories.

- ? viewclass.jsp (Example 2-3) to directory C:\exercise-2-3\src\webapp
- ? ClassViewerServlet.java (Example 2-4) to directory C:\exercise-2-3\src\java
- ? web.xml (Example 2-5) to directory C:\exercise-2-3\src\webapp\WEB-INF
- ? geronimo-jetty.xml (Example 2-6) to directory C:\exercise-2-3\src\webapp\WEB-INF

When that is done, you should have a file structure that looks like the one shown in Example 2-12.

Example 2-12. Our project so far

```
C:\EXERCISE-2-3
+---src
|   +---java
|   |   ClassViewerServlet.java
|   |
|   +---webapp
|   |   classviewer.jsp
|   |
|   +---WEB-INF
|   |   geronimo-jetty.xml
|   |   web.xml
```

About Maven build files

The maven build files follow a slightly different perspective than the Ant equivalent. The top part of the Ant build.xml contains information on the directory structure of our project, namely where the source files were located. The build.properties file also contained information on the name of our project. The rest of the build.xml contained instructions on how to compile, package and deploy the project. In the eyes of Maven,

these are two completely separate things; project data (the model) and the build instructions (the controller).

In this same vein, Maven uses two files:

1. A file for project data called `project.xml`, with the option of a `project.properties` file to hold miscellaneous data just like the Ant `build.properties` file does.
2. Another (optional) file for special build instructions called `maven.xml`.

Notice the very careful choice of words, “special build instructions.” The only reason you would need a `maven.xml` file is if you wish to do something that isn’t already covered by an existing Maven plugin or if you would like to do something special either before or after a plugin runs. Anything you want to be standard or shared should really be pulled out of your `maven.xml` file and wrapped in a plugin. Maven downloads plugins automatically, so distribution and installation are not issues.

The ability to pull build instructions out of your project build files is the reason why the Geronimo team is so supportive of Maven. The Geronimo deploy instructions and server start instructions are our responsibility, not yours. This chapter doesn’t leverage the Geronimo Maven plugin, but subsequent chapters do.

Our motivation is simple; the last thing we want is to see several thousand users putting Geronimo specific build instructions in their build files then copying that build file into a dozen of their other projects. Every time we improve the deploy process, there would be a few hundred thousand build files that would need to be updated.

Updating the build files is the reality for Ant, but it can be completely avoided with Maven.

Making the Maven project files

First we need to create our project descriptor, or Project Object Model (POM). In the root of our project, the `C:\exercise-2-3` directory, create the `project.xml` file as shown in Example 2-13.

Example 2-13. Maven project.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project>
  <pomVersion>3</pomVersion>
  <id>classviewer</id>
  <currentVersion>1.0</currentVersion>

  <dependencies>
    <dependency>
      <groupId>geronimo-spec</groupId>
      <artifactId>geronimo-spec-j2ee</artifactId>
      <version>1.4</version>
    </dependency>
  </dependencies>

  <build>
    <sourceDirectory>${basedir}/src/java</sourceDirectory>
  </build>
```

```
| </project>
```

This project.xml is pretty minimal; the file can actually get much larger than this. The project data that can be expressed in the project.xml file is actually quite rich allowing for some pretty complicated, yet standard, plugins.

Next we want to create a project.properties file to hold some Geronimo-specific information the location where Geronimo is installed and the URI of our project. Create the project.properties file shown in Example 2-14. The project.properties file should be right next to the project.xml file we just created.

Example 2-14. Maven project.properties file

```
# adjust this to point to your geronimo installation
geronimo.home = C:/geronimo-1.0-M42

# should always match the configId attribute in
# the geronimo-jetty.xml file
project.uri = my/first/webapp
```

Making the Maven build file

The two files we just created don't have any build logic in them. They are the "Model" in a Model-View-Controller (MVC) framework. The actual build logic lies in a combination of maven.xml files and plugins for Maven. Since the basic compile, jar, and war type of plugins already exist, we have very little we need to put in our maven.xml file.

Crack open your favorite XML or text editor and create the maven.xml file shown in Example 2-15. This file should be in the same directory as the project.xml file.

Example 2-15. The maven.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project default="war"
  xmlns:j="jelly:core"
  xmlns:ant="jelly:ant"
  xmlns:maven="jelly:maven">

  <goal name="geronimo:deploy" prereqs="jar,war">
    <java jar="${geronimo.home}/bin/deployer.jar" fork="true"
      failonerror="true">
      <arg value="--install"/>
      <arg value="--module"/>
      <arg value="${basedir}/target/${maven.war.final.name}"/>
    </java>
  </goal>

  <goal name="geronimo:start">
    <java jar="${geronimo.home}/bin/server.jar" fork="true"
      failonerror="true" maxmemory="256m" dir="${geronimo.home}">
      <jvmarg value="-ea"/>
      <arg value="${project.uri}"/>
    </java>
  </goal>

</project>
```

If you compare the maven.xml file against the Ant build.xml file in Example 2-10, you'll see that this file is much smaller. The only real information in maven.xml file is one section for the Geronimo deploy command and another for the Geronimo start command.

Both of these Maven goals in our maven.xml will go away once we start using the Geronimo plugin in the next chapter. The nice thing about seeing them here is it gives you an idea of what to do if you need to add logic to your build that isn't covered by any particular plugin.

Building, deploying, and running

If you recall from the Ant section previously, there are number of targets you can run with your build.xml; seven to be exact. Maven refers to these as *goals* and also provides a command to list the goals that are available to your project. From the command prompt run the following command.

```
C:\exercise-2-3> maven --goals
|  \  |__ _Apache__  |
| | \ | / _ \ \ v / -_) ' \ ~ intelligent projects ~
|_| |_\_|_|_\_|_|_|_| v. 1.0-rc2

Available [Plugins] / Goals
.....
[announcement] : Generate release announcement
  generate ..... Generate release announcement
  text ..... Generate release announcement

[ant] : Generate an Ant build file
  generate-build ..... Generate an Ant build file

[antlr] ( NO DEFAULT GOAL )
  generate ..... Generate source from antlr grammars
  prepare-filesystem ..... Make any necessary directories for antlr
  processing

[appserver] ( NO DEFAULT GOAL )
  clean ..... Safely delete an installed appserver instance
  cycle ..... Forced install and start of a appserver instance
  init ..... Initialize resources needed for the plugin
  install ..... Install a appserver instance
  reinstall ..... Reinstall a appserver instance
  restart ..... Restart a appserver instance
  start ..... Start a appserver instance
  stop ..... Stop a appserver instance

...full output omitted.
```

The first thing you notice is that there are a heck of a lot more than seven goals; on my computer, there are over 102 maven plugins installed and a total of 362 available goals. Granted, not all of these are relevant to what we need to accomplish, but the library of build logic available to us is certainly impressive.


```
[java] 21:09:09,346 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="org/apache/geronimo/System"
[java] 21:09:09,506 INFO [Configuration] Started configuration
org/apache/geronimo/System
[java] 21:09:09,526 INFO [ReadOnlyRepository] Repository root is
file:/C:/geronimo-1.0-M42/repository/
[java] 21:09:09,546 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="my/first/webapp"
[java] 21:09:09,556 INFO [ConfigurationManager] Loaded Configuration
geronimo.config:name="org/apache/geronimo/Server"
[java] 21:09:10,027 INFO [Configuration] Started configuration
org/apache/geronimo/Server
[java] 21:09:10,027 INFO [HttpServer] Starting Jetty/5.0.RC0
[java] 21:09:10,027 INFO [HttpServer] Started org.mortbay.jetty.Server@5b8827
[java] 21:09:10,087 INFO [SocketListener] Started SocketListener on
0.0.0.0:8080
[java] 21:09:10,377 INFO [Configuration] Started configuration my/first/webapp
[java] 21:09:10,397 INFO [Credential] Checking Resource aliases
[java] 21:09:12,260 INFO [HttpContext] Started
WebApplicationContext[/example,file:/C:/geronimo-1.0-M42/config-store/23/war/]
```

What just happened?

Just as with the Ant example, we built, deployed and ran the webapp in Geronimo. Unlike the Ant example, we don't have to write and maintain any logic for compiling code, creating jar files and packaging up our war file.

Similar to Ant, Maven put all our compiled code, jars, and wars into a special directory. Take a peek in the directory called 'target' and see. Running the 'jar tf' command on our war file should print out the following contents.

```
C:\exercise-2-3> jar tf target\classviewer.war
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/lib/
WEB-INF/lib/classviewer-1.0.jar
WEB-INF/geronimo-jetty.xml
WEB-INF/web.xml
classviewer.jsp
META-INF/INDEX.LIST
```

With Geronimo still running, browse to <http://localhost:8080/example/ClassViewer> just as before and you should see the same content as in Figure 2-2.