



Building J2EE Applications

This chapter covers:

- Organizing the directory structure
- Building J2EE archives (EJB, WAR, EAR, Web Services)
- Setting up in-place Web development
- Deploying J2EE archives to a container
- Automating container start/stop

Keep your face to the sun and you will
never see the shadows.

- *Helen Keller*

4.1. Introduction

J2EE (or Java EE as it is now called) applications are everywhere. Whether you are using the full J2EE stack with EJBs or only using Web applications with frameworks such as Spring or Hibernate, it's likely that you are using J2EE in some of your projects. As a consequence the Maven community has developed plugins to cover every aspect of building J2EE applications. This chapter will take you through the journey of creating the build for a full-fledged J2EE application called DayTrader. You'll learn not only how to create a J2EE build but also how to create a productive development environment (especially for Web application development) and how to deploy J2EE modules into your container.

4.2. Introducing the DayTrader Application

DayTrader is a real world application developed by IBM and then donated to the Apache Geronimo project. Its goal is to serve as both a functional example of a full-stack J2EE 1.4 application and as a test bed for running performance tests. This chapter demonstrates how to use Maven on a real application to show how to address the complex issues related to automated builds. Through this example, you'll learn how to build EARs, EJBs, Web services, and Web applications. As importantly, you'll learn how to automate configuration and deployment of J2EE application servers.

The functional goal of the DayTrader application is to buy and sell stock, and its architecture is shown in Figure 4-1.

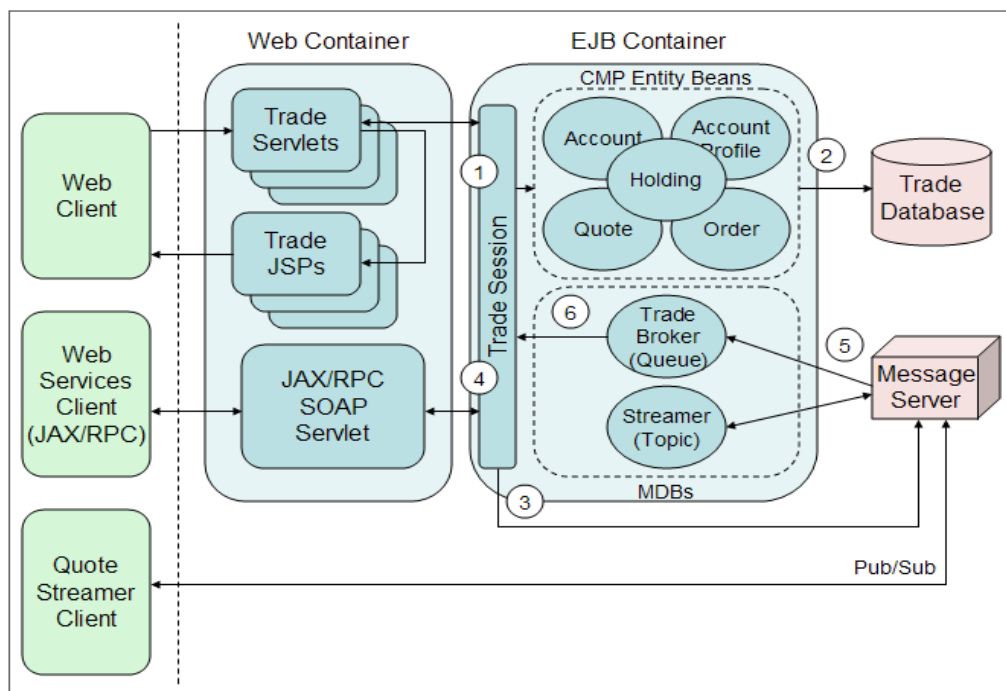


Figure 4-1: Architecture of the DayTrader application

There are 4 layers in the architecture:

- The Client layer offers 3 ways to access the application: using a browser, using Web services, and using the Quote Streamer. The Quote Streamer is a Swing GUI application that monitors quote information about stocks in real-time as the price changes.
- The Web layer offers a view of the application for both the Web client and the Web services client. It uses servlets and JSPs.
- The EJB layer is where the business logic is. The Trade Session is a stateless session bean that offers the business services such as login, logout, get a stock quote, buy or sell a stock, cancel an order, and so on. It uses container-managed persistence (CMP) entity beans for storing the business objects (`Order`, `Account`, `Holding`, `Quote` and `AccountProfile`), and Message-Driven Beans (MDB) to send purchase orders and get quote changes.
- The Data layer consists of a database used for storing the business objects and the status of each purchase, and a JMS Server for interacting with the outside world.

A typical “buy stock” use case consists of the following steps that were shown in Figure 4-1:

1. The user gives a buy order (by using the Web client or the Web services client). This request is handled by the Trade Session bean.
2. A new “open” order is saved in the database using the CMP Entity Beans.
3. The order is then queued for processing in the JMS Message Server.
4. The creation of the “open” order is confirmed for the user.
5. Asynchronously the order that was placed on the queue is processed and the purchase completed. Once this happens the Trade Broker MDB is notified
6. The Trade Broker calls the Trade Session bean which in turn calls the CMP entity beans to mark the order as “completed”. The user is notified of the completed order on a subsequent request.

4.3. Organizing the DayTrader Directory Structure

The first step to organizing the directory structure is deciding what build modules are required. The easy answer is to follow Maven’s artifact guideline: **one module = one main artifact**. Thus you simply need to figure out what artifacts you need. Looking again at Figure 4-1, you can see that the following modules will be needed:

- A module producing an EJB which will contain all of the server-side EJBs.
- A module producing a WAR which will contain the Web application.
- A module producing a JAR that will contain the Quote Streamer client application.
- A module producing another JAR that will contain the Web services client application.

In addition you may need another module producing an EAR which will contain the EJB and WAR produced from the other modules. This EAR will be used to easily deploy the server code into a J2EE container.

Note that this is the minimal number of modules required. It is possible to come up with more. For example, you may want to split the WAR module into 2 WAR modules: one for the browser client and one for the Web services client. Best practices suggest to do this only when the need arises. If there isn't a strong need you may find that managing several modules can be more cumbersome than useful. On the other hand, it is important to split the modules when it is appropriate for flexibility. For example, if you needed to physically locate the WARs in separate servlet containers to distribute the load.

The next step is to give these modules names and map them to a directory structure. As a general rule, it is better to find functional names for modules. However, it is usually easier to choose names that represent a technology instead. For the DayTrader application the following names were chosen:

- `ejb` - the module containing the EJBs
- `web` - the module containing the Web application
- `streamer` - the module containing the client side streamer application
- `wsappclient` - the module containing the Web services client application
- `ear` - the module producing the EAR which packages the EJBs and the Web application

There are two possible layouts that you can use to organize these modules: a flat directory structure and a nested one. Let's discuss the pros and cons of each layout.

Figure 4-2 shows these modules in a **flat directory structure**. It is flat because you're locating all the modules in the same directory.

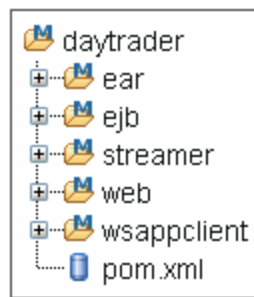


Figure 4-2: Module names and a simple flat directory structure

The top-level `daytrader/pom.xml` file contains the POM elements that are shared between all of the modules. This file also contains the list of modules that Maven will build when executed from this directory (see the Chapter 3, Creating Applications with Maven, for more details):

```
[...]
<modules>
  <module>ejb</module>
  <module>web</module>
  <module>streamer</module>
  <module>wsappclient</module>
  <module>ear</module>
</modules>
[...]
```

This is the easiest and most flexible structure to use, and is the structure used in this chapter. However, if you have many modules in the same directory you may consider finding commonalities between them and create subdirectories to partition them. Note that in this case the modules are still separate, not nested within each other. For example, you might separate the client side modules from the server side modules in the way shown in Figure 4-3.

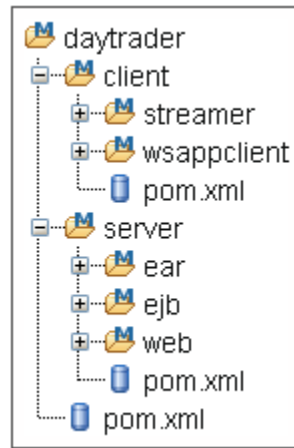


Figure 4-3: Modules split according to a server-side vs client-side directory organization

As before, each directory level containing several modules contains a `pom.xml` file containing the shared POM elements and the list of modules underneath.

The other alternative is to use a **nested directory structure**, as shown in Figure 4-4. In this case, the `ejb` and `web` modules are nested in the `ear` module. This makes sense as the EAR artifact is composed of the EJB and WAR artifacts produced by the `ejb` and `web` modules. Having this nested structure clearly shows how nested modules are linked to their parent.

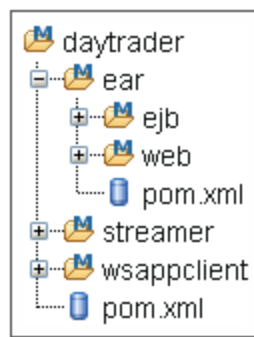


Figure 4-4: Nested directory structure for the EAR, EJB and Web modules

However, even though the nested directory structure seems to work quite well here, it has several drawbacks:

- Eclipse users will have issues with this structure as Eclipse doesn't yet support nested projects. You'd need to consider the three modules as one project, but then you'll be restricted in several ways. For example, the three modules wouldn't be able to have different natures (Web application project, EJB project, EAR project).
- It doesn't allow flexible packaging. For example, the `ejb` or `web` modules might depend on a utility JAR and this JAR may be also required for some other EAR. Or the `ejb` module might be producing a client EJB JAR which is not used by the EAR, but by some client-side application.

These examples show that there are times when there is not a clear parent for a module. In those cases using a nested directory structure should be avoided. In addition, the nested strategy doesn't fit very well with the Assembler role as described in the [J2EE specification](#).

The Assembler has a pool of modules and its role is to package those modules for deployment. Depending on the target deployment environment the Assembler may package things differently: one EAR for one environment or two EARs for another environment where a different set of machines are used, etc. A flat layout is more neutral with regard to assembly and should thus be preferred.

Now that you have decided on the directory structure for the DayTrader application, you're going to create the Maven build for each module, starting with the `wsappclient` module after we take care of one more matter of business. The modules we will work with from here on will each be referring to the parent `pom.xml` of the project, so before we move on to developing these sub-projects we need to install the parent POM into our local repository so it can be further built on.

Now run `mvn -N install` in `daytrader/` in order to install the parent POM in your local repository and make it available to all modules:

```
C:\dev\m2book\code\j2ee\daytrader>mvn -N install
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building DayTrader :: Performance Benchmark Sample
[INFO]    task-segment: [install]
[INFO] -----
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\pom.xml to
C:\[...]\.m2\repository\org\apache\geronimo\samples\daytrader\daytrader\1.0\daytrader-1.0.pom.
```

We are now ready to continue on with developing the sub-projects!

4.4. Building a Web Services Client Project

Web Services are a part of many J2EE applications, and Maven's ability to integrate toolkits can make them easier to add to the build process. For example, the Maven plugin called *Axis Tools plugin* takes WSDL files and generates the Java files needed to interact with the Web services it defines. As the name suggests, the plugin uses the Axis framework (<http://ws.apache.org/axis/java/>), and this will be used from DayTrader's `wsappclient` module. We start our building process off by visiting the Web services portion of the build since it is a dependency of later build stages.

Axis generates the following:

Table 4-1: Axis generated classes

WSDL clause	Java class(es) generated
For each entry in the type section	A Java class A holder if this type is used as an in-out/out parameter
For each port type	A Java interface
For each binding	A stub class
For each service	A service interface A service implementation (the locator)

For more details on the generation process, see <http://ws.apache.org/axis/java/user-guide.html#WSDL2JavaBuildingStubsSkeletonsAndDataTypesFromWSDL>.

Figure 4-5 shows the directory structure of the `wsappclient` module. As you may notice, the WSDL files are in `src/main/wsd`, which is the default used by the Axis Tools plugin:

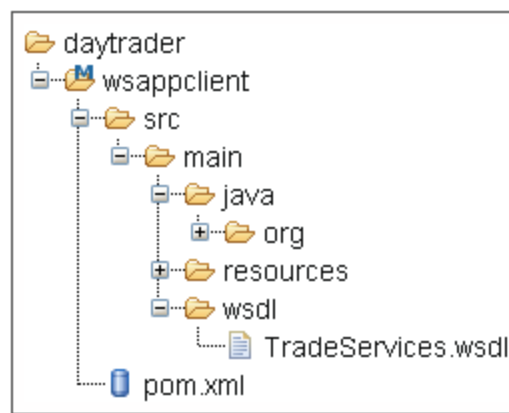


Figure 4-5: Directory structure of the `wsappclient` module

The location of WSDL source can be customized using the `sourceDirectory` property. For example:

```
[...]
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>axistools-maven-plugin</artifactId>
  <configuration>
    <sourceDirectory>
      src/main/resources/META-INF/wsdl
    </sourceDirectory>
  </configuration>
[...]
```

In order to generate the Java source files from the `TradeServices.wsdl` file, the `wsappclient/pom.xml` file must declare and configure the Axis Tools plugin:

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>axistools-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>wsdl2java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Note that there's no need to define a phase in the execution element as the `wsdl2java` goal is bound to the `generate-sources` phase by default.

At this point if you were to execute the build, it would fail. This is because after the sources are generated, you will require a dependency on Axis and Axis JAXRPC in your `pom.xml`. While you might expect the Axis Tools plugin to define this for you, it is required for two reasons: it allows you to control what version of the dependency to use regardless of what the Axis Tools plugin was built against, and more importantly, it allows users of your project to automatically get the dependency transitively. Similarly, any tools that report on the POM will be able to recognize the dependency.

As before, you need to add the J2EE specifications JAR to compile the project's Java sources. Thus add the following three dependencies to your POM:

```
<dependencies>
  <dependency>
    <groupId>axis</groupId>
    <artifactId>axis</artifactId>
    <version>1.2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>axis</groupId>
    <artifactId>axis-jaxrpc</artifactId>
    <version>1.2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-j2ee_1.4_spec</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

The Axis JAR depends on the Mail and Activation Sun JARs which cannot be redistributed by Maven. Thus, they are not present on ibiblio and you'll need to install them manually. Run `mvn install` and Maven will fail and print the installation instructions.

After manually installing Mail and Activation, running the build with `mvn install` leads to:

```
C:\dev\m2book\code\j2ee\daytrader\wsappclient>mvn install
[...]
[INFO] [axistools:wSDL2Java {execution: default}]
[INFO] about to add compile source root
[INFO] processing wsd1:
      C:\dev\m2book\code\j2ee\daytrader\wsappclient\
      src\main\wsdl\TradeServices.wsdl
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 13 source files to
C:\dev\m2book\code\j2ee\daytrader\wsappclient\target\classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [surefire:test]
[INFO] No tests to run.
[INFO] [jar:jar]
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\wsappclient\
target\daytrader-wsappclient-1.0.jar
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\wsappclient\
target\daytrader-wsappclient-1.0.jar to
C:\[...]\m2\repository\org\apache\geronimo\samples\daytrader\
daytrader-wsappclient\1.0\daytrader-wsappclient-1.0.jar
[...]
```

Note that the `daytrader-wsappclient` JAR now includes the class files compiled from the generated source files, in addition to the sources from the standard source directory.

The Axis Tools plugin boasts several other goals including `java2wsdl` that is useful for generating the server-side WSDL file from handcrafted Java classes. The generated WSDL file could then be injected into the Web Services client module to generate client-side Java files. But that's another story... The Axis Tools reference documentation can be found at <http://mojo.codehaus.org/axistools-maven-plugin/>.

Now that we have discussed and built the Web services portion, let's visit EJBs next.

4.5. Building an EJB Project

Let's create a build for the `ejb` module.

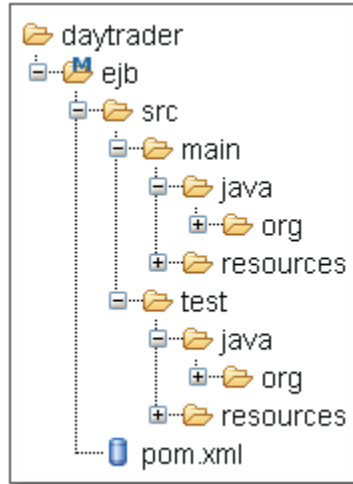


Figure 4-6: Directory structure for the DayTrader `ejb` module

Figure 4-6 shows a canonical directory structure for EJB projects:

- Runtime Java source code in `src/main/java`.
- Runtime classpath resources in `src/main/resources`. More specifically, the standard `ejb-jar.xml` deployment descriptor is in `src/main/resources/META-INF/ejb-jar.xml`. Any container-specific deployment descriptor should also be placed in this directory.
- Unit tests in `src/test/java` and classpath resources for the unit tests in `src/test/resources`. Unit tests are tests that execute in isolation from the container. Tests that require the container to run are called integration tests and are covered at the end of this chapter.

Now, take a look at the content of this project's `pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>daytrader-ejb</artifactId>
  <name>Apache Geronimo DayTrader EJB Module</name>
  <packaging>ejb</packaging>
  <description>DayTrader EJBs</description>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.samples.daytrader</groupId>
      <artifactId>daytrader-wsappclient</artifactId>
      <version>1.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-j2ee_1.4_spec</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.3</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ejb-plugin</artifactId>
        <configuration>
          <generateClient>true</generateClient>
          <clientExcludes>
            <clientExclude>*/ejb/*Bean.class</clientExclude>
          </clientExcludes>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

As you can see, you're extending a parent POM using the `parent` element. This is because the DayTrader build is a multi-module build and you are gathering common POM elements in a parent `daytrader/pom.xml` file. If you look through all the dependencies you should see that we are ready to continue with building and installing this portion of the build.

The `ejb/pom.xml` file is a standard POM file except for three items:

- You need to tell Maven that this project is an EJB project so that it generates an EJB JAR when the package phase is called. This is done by specifying:

```
<packaging>ejb</packaging>
```

- As you're compiling J2EE code you need to have the J2EE specifications JAR in the project's build classpath. This is achieved by specifying a dependency element on the J2EE JAR. You could instead specify a dependency on Sun's J2EE JAR. However, this JAR is not redistributable and as such cannot be found on ibiblio. Fortunately, the Geronimo project has made the J2EE JAR available under an Apache license and this JAR can be found on ibiblio.

You should note that you're using a `provided` scope instead of the default `compile` scope. The reason is that this dependency will already be present in the environment (being the J2EE application server) where your EJB will execute. You make this clear to Maven by using the `provided` scope; this prevents the EAR module from including the J2EE JAR when it is packaged. Even though this dependency is provided at runtime, it still needs to be listed in the POM so that the code can be compiled.

- Lastly, the `pom.xml` contains a configuration to tell the Maven EJB plugin to generate a Client EJB JAR file when `mvn install` is called. The Client will be used in a later examples when building the web module. By default the EJB plugin does not generate the client JAR, so you must explicitly tell it to do so:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-ejb-plugin</artifactId>
  <configuration>
    <generateClient>true</generateClient>
    <clientExcludes>
      <clientExclude>**/ejb/*Bean.class</clientExclude>
    </clientExcludes>
  </configuration>
</plugin>
```

The EJB plugin has a default set of files to exclude from the client EJB JAR: `**/*Bean.class`, `**/*CMP.class`, `**/*Session.class` and `**/package.html`.

In this example, you need to override the defaults using a `clientExclude` element because it happens that there are some required non-EJB files matching the default `**/*Bean.class` pattern and which need to be present in the generated client EJB JAR. Thus you're specifying a pattern that only excludes from the generated client EJB JAR all EJB implementation classes located in the `ejb` package (`**/ejb/*Bean.class`). Note that it's also possible to specify a list of files to include using `clientInclude` elements.

You're now ready to execute the build. Relax and type `mvn install`:

```
C:\dev\m2book\code\j2ee\daytrader\ejb>mvn install
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building DayTrader :: EJBs
[INFO]    task-segment: [install]
[INFO] -----
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 49 source files to C:\dev\m2book\code\j2ee\daytrader\ejb\target\classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
Compiling 1 source file to C:\dev\m2book\code\j2ee\daytrader\ejb\target\test-classes
[INFO] [surefire:test]
[INFO] Setting reports dir: C:\dev\m2book\code\j2ee\daytrader\ejb\target\surefire-reports

-----
T E S T S
-----
[surefire] Running org.apache.geronimo.samples.daytrader.FinancialUtilsTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,02 sec

Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0

[INFO] [ejb:ejb]
[INFO] Building ejb daytrader-ejb-1.0
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\ejb\target\daytrader-ejb-1.0.jar
[INFO] Building ejb client daytrader-ejb-1.0-client
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\ejb\target\daytrader-ejb-1.0-client.jar
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\ejb\target\daytrader-ejb-1.0.jar to
C:\[...]\m2\repository\org\apache\geronimo\samples\
daytrader\daytrader-ejb\1.0\daytrader-ejb-1.0.jar
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\ejb\target\daytrader-ejb-1.0-client.jar to
C:\[...]\m2\repository\org\apache\geronimo\samples\
daytrader\daytrader-ejb\1.0\daytrader-ejb-1.0-client.jar
```

Maven has created both the EJB JAR and the client EJB JAR and installed them in your local Maven repository.

The EJB plugin has several other configuration elements that you can use to suit your exact needs. Please refer to the EJB plugin documentation on <http://maven.apache.org/plugins/maven-ejb-plugin/>.

Early adopters of EJB3 may be interested to know how Maven supports EJB3. At the time of writing, the EJB3 specification is still not final. There is a working prototype of an EJB3 Maven plugin, however in the future it will be added to the main EJB plugin after the specification is finalized. Stay tuned!

4.6. Building an EJB Module With Xdoclet

If you've been developing a lot of EJBs (version 1 and 2) you have probably used [XDoclet](#) to generate all of the EJB interfaces and deployment descriptors for you. Using XDoclet is easy: by adding Javadoc annotations to your classes, you can run the XDoclet processor to generate those files for you. When writing EJBs it means you simply have to write your EJB implementation class and XDoclet will generate the Home interface, the Remote and Local interfaces, the container-specific deployment descriptors, and the `ejb-jar.xml` descriptor.

Note that if you're an EJB3 user, you can safely skip this section – you won't need it!

Here's an extract of the TradeBean session EJB using Xdoclet:

```
/**
 * Trade Session EJB manages all Trading services
 *
 * @ejb.bean
 *     display-name="TradeEJB"
 *     name="TradeEJB"
 *     view-type="remote"
 *     impl-class-name=
 *         "org.apache.geronimo.samples.daytrader.ejb.TradeBean"
 * @ejb.home
 *     generate="remote"
 *     remote-class=
 *         "org.apache.geronimo.samples.daytrader.ejb.TradeHome"
 * @ejb.interface
 *     generate="remote"
 *     remote-class=
 *         "org.apache.geronimo.samples.daytrader.ejb.Trade"
 * [...]
 */
public class TradeBean implements SessionBean
{
    [...]
    /**
     * Queue the Order identified by orderID to be processed in a
     * One Phase commit [...]
     *
     * @ejb.interface-method
     *     view-type="remote"
     * @ejb.transaction
     *     type="RequiresNew"
     * [...]
     */
    public void queueOrderOnePhase(Integer orderID)
        throws javax.jms.JMSEException, Exception
    [...]
}
```

To demonstrate XDoclet, create a copy of the DayTrader `ejb` module called `ejb-xdoclet`. As you can see in Figure 4-7, the project's directory structure is the same as in Figure 4-6, but you don't need the `ejb-jar.xml` file anymore as it's going to be generated by Xdoclet.

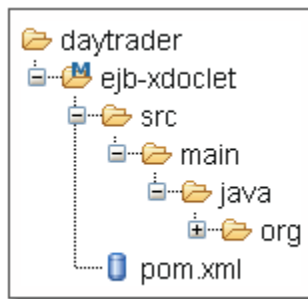


Figure 4-7: Directory structure for the DayTrader `ejb` module when using Xdoclet

The other difference is that you only need to keep the `*Bean.java` classes and remove all of the Home, Local and Remote interfaces as they'll also get generated.

Now you need to tell Maven to run XDoclet on your project. Since XDoclet generates source files, this has to be run before the compilation phase occurs. This is achieved by using the [Maven XDoclet plugin](#) and binding it to the `generate-sources` life cycle phase. Here's the portion of the `pom.xml` that configures the plugin:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>xdoclet-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>xdoclet</goal>
      </goals>
      <configuration>
        <tasks>
          <ejbdoclet verbose="true" force="true" ejbSpec="2.1" destDir=
            "${project.build.directory}/generated-sources/xdoclet">
            <fileset dir="${project.build.sourceDirectory}">
              <include name="**/*Bean.java"></include>
              <include name="**/*MDB.java"></include>
            </fileset>
            <homeinterface/>
            <remoteinterface/>
            <localhomeinterface/>
            <localinterface/>
            <deploymentdescriptor
              destDir="${project.build.outputDirectory}/META-INF"/>
            </ejbdoclet>
          </tasks>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

The XDoclet plugin is configured within an execution element. This is required by Maven to bind the `xdoclet` goal to a phase. The plugin generates sources by default in `${project.build.directory}/generated-sources/xdoclet` (you can configure this using the `generatedSourcesDirectory` configuration element).

It also tells Maven that this directory contains sources that will need to be compiled when the compile phase executes. Finally, in the tasks element you use the `ejbdoclet` Ant task provided by the XDoclet project (for reference documentation see <http://xdoclet.sourceforge.net/xdoclet/ant/xdoclet/modules/ejb/EjbDocletTask.html>).

In practice you can use any XDoclet task (or more generally any Ant task) within the tasks element, but here the need is to use the `ejbdoclet` task to instrument the EJB class files. In addition, the XDoclet plugin will also trigger Maven to download the XDoclet libraries from Maven's remote repository and add them to the execution classpath.

Executing `mvn install` now automatically executes XDoclet and compiles the generated files:

```
C:\dev\m2book\code\j2ee\daytrader\ejb-xdoclet>mvn install
[...]
```

```
[INFO] [xdoclet:xdoclet {execution: default}]
[INFO] Initializing DocletTasks!!!
[INFO] Executing tasks
10 janv. 2006 16:53:50 xdoclet.XDocletMain start
INFO: Running <homeinterface/>
Generating Home interface for
    'org.apache.geronimo.samples.daytrader.ejb.TradeBean'.
[...]
```

```
INFO: Running <remoteinterface/>
Generating Remote interface for
    'org.apache.geronimo.samples.daytrader.ejb.TradeBean'.
[...]
```

```
10 janv. 2006 16:53:50 xdoclet.XDocletMain start
INFO: Running <localhomeinterface/>
Generating Local Home interface for
    'org.apache.geronimo.samples.daytrader.ejb.AccountBean'.
[...]
```

```
10 janv. 2006 16:53:51 xdoclet.XDocletMain start
INFO: Running <localinterface/>
Generating Local interface for
    'org.apache.geronimo.samples.daytrader.ejb.AccountBean'.
[...]
```

```
10 janv. 2006 16:53:51 xdoclet.XDocletMain start
INFO: Running <deploymentdescriptor/>
Generating EJB deployment descriptor (ejb-jar.xml).
[...]
```

```
[INFO] [ejb:ejb]
[INFO] Building ejb daytrader-ejb-1.0
[...]
```

You might also want to try [XDoclet2](http://xdoclet2.codehaus.org/). It's based on a new architecture but the tag syntax is backward-compatible in most cases. There's also a Maven 2 plugin for XDoclet2 at <http://xdoclet.codehaus.org/Maven2+Plugin>. However, it should be noted that XDoclet2 is a work in progress and is not yet fully mature, nor does it boast all the plugins that XDoclet1 has.

4.7. Deploying EJBs

Now that you know how to build an EJB project, you will learn how to deploy it. Later, you will also learn how to test it automatically, in the *Testing J2EE Applications* section of this chapter. Let's discover how you can automatically start a container and deploy your EJBs into it.

First, you will need to have Maven start the container automatically. To do so you're going to use the Maven plugin for Cargo. [Cargo](#) is a framework for manipulating containers. It offers generic APIs (Java, Ant, Maven 1, Maven 2, Netbeans, IntelliJ IDEA, etc.) for performing various actions on containers such as starting, stopping, configuring them and deploying modules to them. In this example, the JBoss container will be used.

Edit the `ejb/pom.xml` file and add the following Cargo plugin configuration:

```
<build>
  <plugins>
    [...]
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <configuration>
        <container>
          <containerId>jboss4x</containerId>
          <zipUrlInstaller>
            <url>http://internap.dl.sourceforge.net/
              sourceforge/jboss/jboss-4.0.2.zip</url>
            <installDir>${installDir}</installDir>
          </zipUrlInstaller>
        </container>
      </configuration>
    </plugin>
  </plugins>
</build>
```

If you want to debug Cargo's execution, you can use the `log` element to specify a file where Cargo logs will go and you can also use the `output` element to specify a file where the container's output will be dumped. For example:

```
<container>
  <containerId>jboss4x</containerId>
  <output>${project.build.directory}/jboss4x.log</output>
  <log>${project.build.directory}/cargo.log</log>
  [...]
</container>
```

See <http://cargo.codehaus.org/Debugging> for full details.

In the `container` element you tell the Cargo plugin that you want to use JBoss 4.x (`containerId` element) and that you want Cargo to download the JBoss 4.0.2 distribution from the specified URL and install it in `${installDir}`. The location where Cargo should install JBoss is a user-dependent choice and this is why the `${installDir}` property was introduced. In order to build this project you need to create a Profile where you define the `${installDir}` property's value.

As explained in Chapter 3, you can define a profile in the POM, in a `profiles.xml` file, or in a `settings.xml` file. In this case, as the content of the Profile is user-dependent you wouldn't want to define it in the POM. Nor should the content be shared with other Maven projects at large, in a `settings.xml` file. Thus the best place is to create a `profiles.xml` file in the `ejb/` directory:

```
<profilesXml>
  <profiles>
    <profile>
      <id>vmassol</id>
      <properties>
        <installDir>c:/apps/cargo-installs</installDir>
      </properties>
    </profile>
  </profiles>
  <activeProfiles>
    <activeProfile>vmassol</activeProfile>
  </activeProfiles>
</profilesXml>
```

This sample `profiles.xml` file defines a profile named `vmassol`, activated by default and in which the `${installDir}` property points to `c:/apps/cargo-installs`.

It's also possible to tell Cargo that you already have JBoss installed locally. In that case replace the `zipURLInstaller` element with a `home` element. For example:

```
<home>c:/apps/jboss-4.0.2</home>
```

That's all you need to have a working build and to deploy the EJB JAR into JBoss. The Cargo plugin does all the work: it provides a default JBoss configuration (using port 8080 for example), it detects that the Maven project is producing an EJB from the packaging element and it automatically deploys it when the container is started.

Of course, the EJB JAR should first be created, so run `mvn package` to generate it, then start JBoss and deploy the EJB JAR by running `mvn cargo:start` (or `mvn package cargo:start` to do it all at once):

```
C:\dev\m2book\code\j2ee\daytrader\ejb>mvn cargo:start
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'cargo'.
[INFO] -----
[INFO] Building DayTrader :: EJBs
[INFO]    task-segment: [cargo:start]
[INFO] -----
[INFO] [cargo:start]
[INFO] [talledLocalContainer] Parsed JBoss version = [4.0.2]
[INFO] [talledLocalContainer] JBoss 4.0.2 starting...
[INFO] [talledLocalContainer] JBoss 4.0.2 started on port [8080]
[INFO] Press Ctrl-C to stop the container...
```

That's it! JBoss is running, and the EJB JAR has been deployed.

As you have told Cargo to download and install JBoss, the first time you execute `cargo:start` it will take some time, especially if you are on a slow connection. Subsequent calls will be fast as Cargo will not download JBoss again.

If the container was already started and you wanted to just deploy the EJB, you would run the `cargo:deploy` goal. Finally, to stop the container call `mvn cargo:stop`.

Cargo has many other configuration options such as the possibility of using an existing container installation, modifying various container parameters, deploying on a remote machine, and more. Check the documentation at <http://cargo.codehaus.org/Maven2+plugin>.

4.8. Building a Web Application Project

Now, let's focus on building the DayTrader web module. The layout is the same as for a JAR module (see the first two chapters of this book), except that there is an additional `src/main/webapp` directory for locating Web application resources such as HTML pages, JSPs, WEB-INF configuration files, etc. (see Figure 4-8).

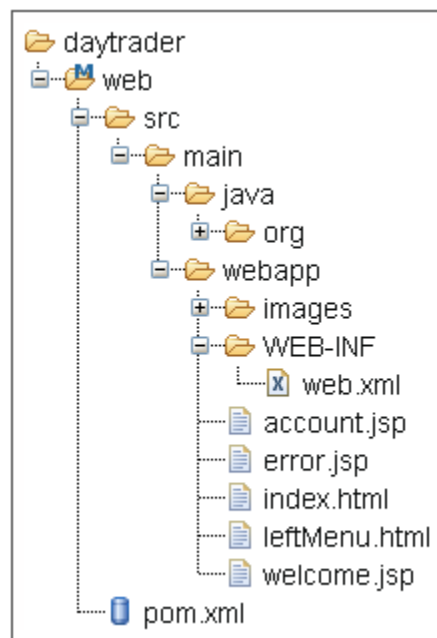


Figure 4-8: Directory structure for the DayTrader web module showing some Web application resources

As usual everything is specified in the `pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>daytrader-web</artifactId>
  <name>DayTrader :: Web Application</name>
  <packaging>war</packaging>
  <description>DayTrader Web</description>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.samples.daytrader</groupId>
      <artifactId>daytrader-ejb</artifactId>
      <version>1.0</version>
      <type>ejb-client</type>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-j2ee_1.4_spec</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

You start by telling Maven that it's building a project generating a WAR:

```
<packaging>war</packaging>
```

Next, you specify the required dependencies. The reason you are building this web module after the `ejb` module is because the web module's servlets call the EJBs. Therefore, you need to add a dependency on the `ejb` module in `web/pom.xml`:

```
<dependency>
  <groupId>org.apache.geronimo.samples.daytrader</groupId>
  <artifactId>daytrader-ejb</artifactId>
  <version>1.0</version>
  <type>ejb-client</type>
</dependency>
```

Note that you're specifying a type of `ejb-client` and not `ejb`. This is because the servlets are a client of the EJBs. Therefore, the servlets only need the EJB client JAR in their classpath to be able to call the EJBs. This is why you told the EJB plugin to generate a client JAR earlier on in `ejb/pom.xml`.

Depending on the main EJB JAR would also work, but it's not necessary and would increase the size of the WAR file. It's always cleaner to depend on the minimum set of required classes, for example to prevent coupling.

If you add a dependency on a WAR, then the WAR you generate will be overlaid with the content of that dependent WAR, allowing the aggregation of multiple WAR files. However, only files not in the existing Web application will be added, and files such as `web.xml` won't be merged. An alternative is to use the `uberwar` goal from the Cargo Maven Plugin (see <http://cargo.codehaus.org/Merging+WAR+files>).

The final dependency listed is the J2EE JAR as your web module uses servlets and calls EJBs. Again, the Geronimo J2EE specifications JAR is used with a provided scope (as seen previously when building the EJB).



As you know, Maven 2 supports transitive dependencies. When it generates your WAR, it recursively adds your module's dependencies, unless their scope is test or provided. This is why we defined the J2EE JAR using a `provided` scope in the web module's `pom.xml`. Otherwise it would have surfaced in the `WEB-INF/lib` directory of the generated WAR.

The configuration is very simple because the defaults from the WAR plugin are being used. As seen in the introduction, it's a good practice to use the default conventions as much as possible, as it reduces the size of the `pom.xml` file and reduces maintenance.

Running `mvn install` generates the WAR and installs it in your local repository:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn install
[...]
```

```
[INFO] [war:war]
[INFO] Exploding webapp...
[INFO] Copy webapp resources to
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Assembling webapp daytrader-web in
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Generating war
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
[INFO] Building war:
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
[INFO] [install:install]
[INFO] Installing
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
       to C:\[...]\.m2\repository\org\apache\geronimo\samples\daytrader\
       daytrader-web\1.0\daytrader-web-1.0.war
```

Table 4-2 lists some other parameters of the WAR plugin that you may wish to configure.

Table 4-2: WAR plugin configuration properties

Configuration property	Default value	Description
warSourceDirectory	<code>\${basedir}/src/main/webapp</code>	Location of Web application resources to include in the WAR.
webXml	The web.xml file found in <code>\${warSourceDirectory}/WEB-INF/web.xml</code>	Specify where to find the web.xml file.
warSourceIncludes/warSourceExcludes	All files are included	Specify the files to include/exclude from the generated WAR.
warName	<code>\${project.build.finalName}</code>	Name of the generated WAR.

For the full list, see the reference documentation for the WAR plugin at <http://maven.apache.org/plugins/maven-war-plugin/>.

4.9. Improving Web Development Productivity

If you're doing Web development you know how painful it is to have to package your code in a WAR and redeploy it every time you want to try out a change you made to your HTML, JSP or servlet code. Fortunately, Maven can help. There are two plugins that can alleviate this problem: the Cargo plugin and the Jetty6 plugin. You'll discover how to use the Jetty6 plugin in this section as you've already seen how to use the Cargo plugin in a previous section.

The Jetty6 plugin creates a custom Jetty6 configuration that is wired to your source tree. The plugin is configured by default to look for resource files in `src/main/webapp`, and it adds the compiled classes in `target/classes` to its execution classpath. The plugin monitors the source tree for changes, including the `pom.xml` file, the `web.xml` file, the `src/main/webapp` tree, the project dependencies and the compiled classes and classpath resources in `target/classes`. If any change is detected, the plugin reloads the Web application in Jetty.

A typical usage for this plugin is to develop the source code in your IDE and have the IDE configured to compile classes in `target/classes` (this is the default when the Maven IDE plugins are used to set up your IDE project). Thus any recompilation in your IDE will trigger a redeploy of your Web application in Jetty, providing an extremely fast turnaround time for development.

Let's try the Jetty6 plugin on the DayTrader web module. Add the following to the `web/pom.xml` file:

```
[...]
<build>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty6-plugin</artifactId>
      <configuration>
        <scanIntervalSeconds>10</scanIntervalSeconds>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>org.apache.geronimo.specs</groupId>
          <artifactId>geronimo-j2ee_1.4_spec</artifactId>
          <version>1.0</version>
          <scope>provided</scope>
        </dependency>
      </dependencies>
    </plugin>
  </build>
[...]
```

The `scanIntervalSeconds` configuration property tells the plugin to monitor for changes every 10 seconds. The reason for the dependency on the J2EE specification JAR is because Jetty is a servlet engine and doesn't provide the EJB specification JAR. Since the Web application earlier declared that the specification must be provided through the provided scope, adding this dependency to the plugin adds it to the classpath for Jetty.

You execute the Jetty6 plugin by typing `mvn jetty6:run`:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn jetty6:run
[...]
```

```
[INFO] [jetty6:run]
[INFO] Configuring Jetty for project:
        Apache Geronimo DayTrader Web Module
[INFO] Webapp source directory is:
C:\dev\m2book\code\j2ee\daytrader\web\src\main\webapp
[INFO] web.xml file located at: C:\dev\m2book\code\j2ee\daytrader\
web\src\main\webapp\WEB-INF\web.xml
[INFO] Classes located at: C:\dev\m2book\code\j2ee\daytrader\
web\target\classes
[INFO] tmp dir for webapp will be
C:\dev\m2book\code\j2ee\daytrader\web\target\jetty-tmp
[INFO] Starting Jetty Server ...
[INFO] No connectors configured, using defaults:
org.mortbay.jetty.nio.SelectChannelConnector listening on 8080
with maxIdleTime 30000
0 [main] INFO org.mortbay.log - Logging to
org.slf4j.impl.SimpleLogger@1242b11 via org.mortbay.log.Slf4jLog
[INFO] Context path = /daytrader-web
[INFO] Webapp directory =
C:\dev\m2book\code\j2ee\daytrader\web\src\main\webapp
[INFO] Setting up classpath ...
[INFO] Finished setting up classpath
[INFO] Started configuring web.xml, resource base=
C:\dev\m2book\code\j2ee\daytrader\web\src\main\webapp
[INFO] Finished configuring web.xml
681 [main] INFO org.mortbay.log - Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Starting scanner at interval of 10 seconds.
```

As you can see, Maven pauses as Jetty is now started and may be stopped at anytime by simply typing **Ctrl-C**, but then the fun examples won't work!. Your Web application has been deployed and the plugin is waiting, listening for changes. Open a browser with the `http://localhost:8080/daytrader-web/register.jsp` URL as shown in Figure 4-9 to see the Web application running.

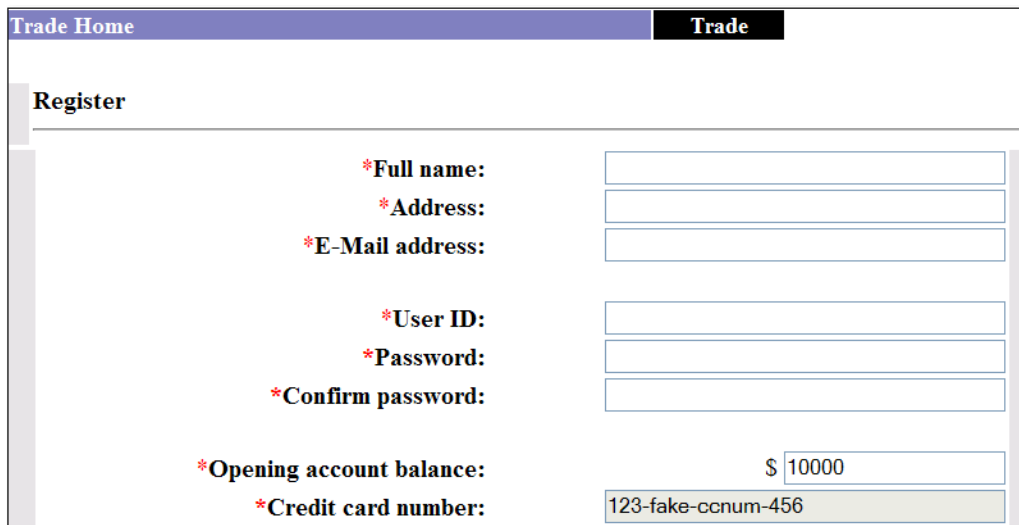


Figure 4-9: DayTrader JSP registration page served by the Jetty6 plugin

Note that the application will fail if you open a page that calls EJBs. The reason is that we have only deployed the Web application here, but the EJBs and all the back end code has not been deployed. In order to make it work you'd need to have your EJB container started with the DayTrader code deployed in it. In practice it's easier to deploy a full EAR as you'll see below.

Now let's try to modify the content of this JSP by changing the opening account balance. Edit `web/src/main/webapp/register.jsp`, search for "10000" and replace it with "90000" (a much better starting amount!):

```
<TD colspan="2" align="right">${B} </B><INPUT size="20" type="text"
  name="money" value='<%= money==null ? "90000" : money %>'></TD>
```

Now simply refresh your browser (usually the F5 key) and the new value will appear as shown in Figure 4-10:

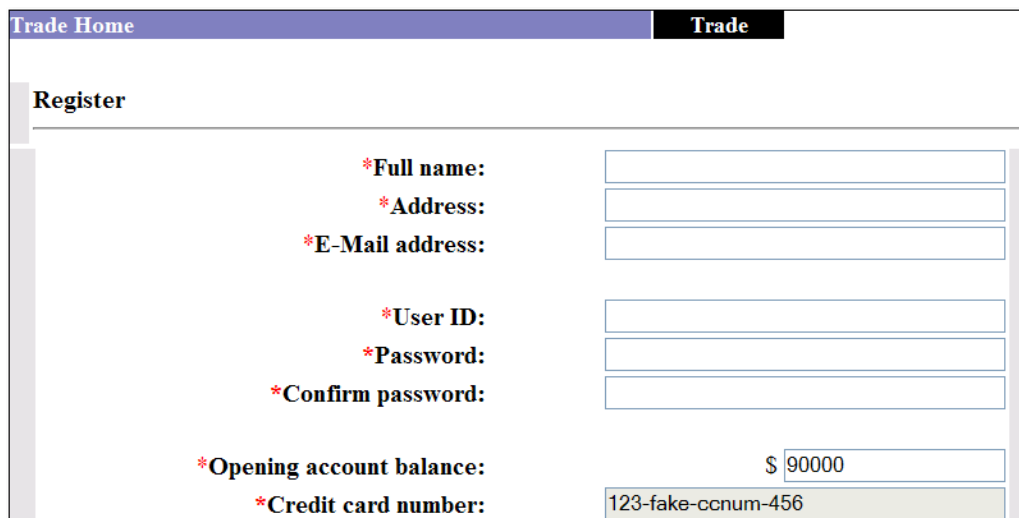


Figure 4-10: Modified registration page automatically reflecting our source change

That's nifty, isn't it? What happened is that the Jetty6 plugin realized the page was changed and it redeployed the Web application automatically. The Jetty container automatically recompiled the JSP when the page was refreshed.

There are various configuration parameters available for the Jetty6 plugin such as the ability to define Connectors and Security realms. For example if you wanted to run Jetty on port 9090 with a user realm defined in `etc/realm.properties`, you would use:

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty6-plugin</artifactId>
  <configuration>
    [...]
    <connectors>
      <connector implementation=
        "org.mortbay.jetty.nio.SelectChannelConnector">
        <port>9090</port>
        <maxIdleTime>60000</maxIdleTime>
      </connector>
    </connectors>
    <userRealms>
      <userRealm implementation=
        "org.mortbay.jetty.security.HashUserRealm">
        <name>Test Realm</name>
        <config>etc/realm.properties</config>
      </userRealm>
    </userRealms>
  </configuration>
</plugin>
```

You can also configure the context under which your Web application is deployed by using the `contextPath` configuration element. By default the plugin uses the module's `artifactId` from the POM.

It's also possible to pass in a `jetty.xml` configuration file using the `jettyConfig` configuration element. In that case anything in the `jetty.xml` file will be applied first. For a reference of all configuration options see the Jetty6 plugin documentation at <http://jetty.mortbay.org/jetty6/maven-plugin/index.html>.

Now imagine that you have an awfully complex Web application generation process, that you have custom plugins that do all sorts of transformations to Web application resource files, possibly generating some files, and so on. The strategy above would not work as the Jetty6 plugin would not know about the custom actions that need to be executed to generate a valid Web application. Fortunately there's a solution.

The WAR plugin has an exploded goal which produces an expanded Web application in the target directory. Calling this goal ensures that the generated Web application is the correct one. The Jetty6 plugin also contains two goals that can be used in this situation:

- `jetty6:run-war`: The plugin first runs the package phase which generates the WAR file. Then the plugin deploys the WAR file to the Jetty server and it performs hot redeployments whenever the WAR is rebuilt (by calling `mvn package` from another window, for example) or when the `pom.xml` file is modified.
- `jetty6:run-exploded`: The plugin runs the package phase as with the `jetty6:run-war` goal. Then it deploys the unpacked Web application located in `target/` (whereas the `jetty6:run-war` goal deploys the WAR file). The plugin then watches the following files: `WEB-INF/lib`, `WEB-INF/classes`, `WEB-INF/web.xml` and `pom.xml`; any change to those files results in a hot redeployment.

To demonstrate, execute `mvn jetty6:run-exploded` goal on the web module:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn jetty6:run-exploded
[...]
```

```
[INFO] [war:war]
[INFO] Exploding webapp...
[INFO] Copy webapp resources to
C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Assembling webapp daytrader-web in
C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Generating war C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
[INFO] Building war: C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
[INFO] [jetty6:run-exploded]
[INFO] Configuring Jetty for project: DayTrader :: Web Application
[INFO] Starting Jetty Server ...
0 [main] INFO org.mortbay.log - Logging to org.slf4j.impl.SimpleLogger@78bc3b via
org.mortbay.log.Slf4jLog
[INFO] Context path = /daytrader-web
2214 [main] INFO org.mortbay.log - Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Scanning ...
[INFO] Scan complete at Wed Feb 15 11:59:00 CET 2006
[INFO] Starting scanner at interval of 10 seconds.
```

As you can see the WAR is first assembled in the `target` directory and the Jetty plugin is now waiting for changes to happen. If you open another shell and run `mvn package` you'll see the following in the first shell's console:

```
[INFO] Scan complete at Wed Feb 15 12:02:31 CET 2006
[INFO] Calling scanner listeners ...
[INFO] Stopping webapp ...
[INFO] Reconfiguring webapp ...
[INFO] Restarting webapp ...
[INFO] Restart completed.
[INFO] Listeners completed.
[INFO] Scanning ...
```

You're now ready for productive web development. No more excuses!

4.10. Deploying Web Applications

You have already seen how to deploy a Web application for in-place Web development in the previous section, so now the focus will be on deploying a packaged WAR to your target container. This example uses the Cargo Maven plugin to deploy to any container supported by Cargo (see <http://cargo.codehaus.org/Containers>). This is very useful when you're developing an application and you want to verify it works on several containers.

First, edit the web module's `pom.xml` file and add the Cargo configuration:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      <containerId>${containerId}</containerId>
      <zipUrlInstaller>
        <url>${url}</url>
        <installDir>${installDir}</installDir>
      </zipUrlInstaller>
    </container>
  </configuration>
  <properties>
    <cargo.servlet.port>8280</cargo.servlet.port>
  </properties>
</configuration>
</plugin>
```

As you can see this is a configuration similar to the one you have used to deploy your EJBs in the *Deploying EJBs* section of this chapter. There are two differences though:

- Two new properties have been introduced (`containerId` and `url`) in order to make this build snippet generic. Those properties will be defined in a Profile.
- A `cargo.servlet.port` element has been introduced to show how to configure the containers to start on port 8280 instead of the default 8080 port. This is very useful if you have containers already running your machine and you don't want to interfere with them.

As seen in the *Deploying EJBs* section the `installDir` property is user-dependent and should be defined in a `profiles.xml` file. However, the `containerId` and `url` properties should be shared for all users of the build. Thus, add the following profiles to the `web/pom.xml` file:

```
[...]
</build>
<profiles>
  <profile>
    <id>jboss4x</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <containerId>jboss4x</containerId>
      <url>http://ovh.dl.sourceforge.net/sourceforge/jboss/jboss4.0.2.zip</url>
    </properties>
  </profile>
  <profile>
    <id>tomcat5x</id>
    <properties>
      <containerId>tomcat5x</containerId>
      <url>http://www.apache.org/dist/jakarta/tomcat-5/v5.0.30/bin/
        jakarta-tomcat-5.0.30.zip</url>
    </properties>
  </profile>
</profiles>
</project>
```

You have defined two profiles: one for JBoss and one for Tomcat and the JBoss profile is defined as active by default (using the `activation` element). You could add as many profiles as there are containers you want to execute your Web application on.

Executing `mvn install cargo:start` generates the WAR, starts the JBoss container and deploys the WAR into it:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn install cargo:start
[...]  
[INFO] [cargo:start]  
[INFO] [talledLocalContainer] Parsed JBoss version = [4.0.2]  
[INFO] [talledLocalContainer] JBoss 4.0.2 starting...  
[INFO] [talledLocalContainer] JBoss 4.0.2 started on port [8280]  
[INFO] Press Ctrl-C to stop the container...
```

To deploy the WAR using Tomcat tell Maven to execute the `tomcat5x` profile by typing `mvn cargo:start -Ptomcat5x`:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn cargo:start -Ptomcat5x
[...]  
[INFO] [cargo:start]  
[INFO] [talledLocalContainer] Tomcat 5.0.30 starting...  
[INFO] [CopyingLocalDeployer] Deploying  
[INFO] [C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war]  
[INFO] to [C:\[...]\Temp\cargo\50866\webapps]...  
[INFO] [talledLocalContainer] Tomcat 5.0.30 started on port [8280]  
[INFO] Press Ctrl-C to stop the container...
```

This is useful for development and to test that your code deploys and works. However, once this is verified you'll want a solution to deploy your WAR into an integration platform. One solution is to have your container running on that integration platform and to perform a remote deployment of your WAR to it.

To deploy the DayTrader's WAR to a running JBoss server on machine `remoteserver` and executing on port 80, you would need the following Cargo plugin configuration in `web/pom.xml`:

```
<plugin>  
  <groupId>org.codehaus.cargo</groupId>  
  <artifactId>cargo-maven2-plugin</artifactId>  
  <configuration>  
    <container>  
      <containerId>jboss4x</containerId>  
      <type>remote</type>  
    </container>  
    <configuration>  
      <type>runtime</type>  
      <properties>  
        <cargo.hostname>${remoteServer}</cargo.hostname>  
        <cargo.servlet.port>${remotePort}</cargo.servlet.port>  
        <cargo.remote.username>${remoteUsername}</cargo.remote.username>  
        <cargo.remote.password>${remotePassword}</cargo.remote.password>  
      </properties>  
    </configuration>  
  </configuration>  
</plugin>
```


When compared to the configuration for a local deployment above, the changes are:

- A remote container and configuration type to tell Cargo that the container is remote and not under Cargo's management,
- Several configuration properties (especially a user name and password allowed to deploy on the remote JBoss container) to specify all the details required to perform the remote deployment. All the properties introduced need to be declared inside the POM for those shared with other users and in the `profiles.xml` file (or the `settings.xml` file) for those user-dependent. Note that there was no need to specify a deployment URL as it is computed automatically by Cargo.

Check the Cargo reference documentation for all details on deployments at <http://cargo.codehaus.org/Deploying+to+a+running+container>.

4.11. Building an EAR Project

You have now built all the individual modules. It's time to package the server module artifacts (EJB and WAR) into an EAR for convenient deployment. The ear module's directory structure can't be any simpler... it solely consists of a `pom.xml` file (see Figure 4-11).

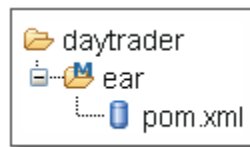


Figure 4-11: Directory structure of the ear module

As usual the magic happens in the `pom.xml` file. Start by defining that this is an EAR project by using the `packaging` element:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>daytrader-ear</artifactId>
  <name>DayTrader :: Enterprise Application</name>
  <packaging>ear</packaging>
  <description>DayTrader EAR</description>
```

Next, define all of the dependencies that need to be included in the generated EAR:

```
<dependencies>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-wsappclient</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-web</artifactId>
    <version>1.0</version>
    <type>war</type>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-ejb</artifactId>
    <version>1.0</version>
    <type>ejb</type>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-streamer</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
```

Finally, you need to configure the Maven EAR plugin by giving it the information it needs to automatically generate the `application.xml` deployment descriptor file. This includes the display name to use, the description to use, and the J2EE version to use. It is also necessary to tell the EAR plugin which of the dependencies are Java modules, Web modules, and EJB modules. At the time of writing, the EAR plugin supports the following module types: `ejb`, `war`, `jar`, `ejb-client`, `rar`, `ejb3`, `par`, `sar` and `wsr`.

By default, all dependencies are included, with the exception of those that are optional, or those with a scope of test or provided. However, it is often necessary to customize the inclusion of some dependencies such as shown in this example:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-ear-plugin</artifactId>
      <configuration>
        <displayName>Trade</displayName>
        <description>
          DayTrader Stock Trading Performance Benchmark Sample
        </description>
        <version>1.4</version>
        <modules>
          <javaModule>
            <groupId>org.apache.geronimo.samples.daytrader</groupId>
            <artifactId>daytrader-streamer</artifactId>
            <includeInApplicationXml>true</includeInApplicationXml>
          </javaModule>
          <javaModule>
            <groupId>org.apache.geronimo.samples.daytrader</groupId>
            <artifactId>daytrader-wsappclient</artifactId>
            <includeInApplicationXml>true</includeInApplicationXml>
          </javaModule>
          <webModule>
            <groupId>org.apache.geronimo.samples.daytrader</groupId>
            <artifactId>daytrader-web</artifactId>
            <contextRoot>/daytrader</contextRoot>
          </webModule>
        </modules>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

Here, the `contextRoot` element is used for the `daytrader-web` module definition to tell the EAR plugin to use that context root in the generated `application.xml` file.

You should also notice that you have to specify the `includeInApplicationXml` element in order to include the `streamer` and `wsappclient` libraries into the EAR. By default, only EJB client JARs are included when specified in the Java modules list.

It is also possible to configure where the JARs' Java modules will be located inside the generated EAR. For example, if you wanted to put the libraries inside a lib subdirectory of the EAR you would use the `bundleDir` element:

```
<javaModule>
  <groupId>org.apache.geronimo.samples.daytrader</groupId>
  <artifactId>daytrader-streamer</artifactId>
  <includeInApplicationXml>true</includeInApplicationXml>
  <bundleDir>lib</bundleDir>
</javaModule>
<javaModule>
  <groupId>org.apache.geronimo.samples.daytrader</groupId>
  <artifactId>daytrader-wsappclient</artifactId>
  <includeInApplicationXml>true</includeInApplicationXml>
  <bundleDir>lib</bundleDir>
</javaModule>
```

In order not to have to repeat the `bundleDir` element for each Java module definition you can instead use the `defaultJavaBundleDir` element:

```
[...]
<defaultBundleDir>lib</defaultBundleDir>
<modules>
  <javaModule>
    [...]
  </javaModule>
[...]
```

There are some other configuration elements available in the EAR plugin which you can find out by checking the reference documentation on <http://maven.apache.org/plugins/maven-ear-plugin>.

The streamer module's build is not described in this chapter because it's a standard build generating a JAR. However the ear module depends on it and thus you'll need to have the Streamer JAR available in your local repository before you're able to run the ear module's build. Run `mvn install` in `daytrader/streamer`.

To generate the EAR, run `mvn install`:

```
C:\dev\m2book\code\j2ee\daytrader\ear>mvn install
[...]
[INFO] [ear:generate-application-xml]
[INFO] Generating application.xml
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [ear:ear]
[INFO] Copying artifact [jar:org.apache.geronimo.samples.daytrader:
daytrader-streamer:1.0] to [daytrader-streamer-1.0.jar]
[INFO] Copying artifact [jar:org.apache.geronimo.samples.daytrader:
daytrader-wsappclient:1.0] to
[daytrader-wsappclient-1.0.jar]
[INFO] Copying artifact [war:org.apache.geronimo.samples.daytrader:
daytrader-web:1.0] to [daytrader-web-1.0.war]
[INFO] Copying artifact [ejb:org.apache.geronimo.samples.daytrader:
daytrader-ejb:1.0] to [daytrader-ejb-1.0.jar]
[INFO] Copying artifact
[ejb-client:org.apache.geronimo.samples.daytrader:
daytrader-ejb:1.0] to [daytrader-ejb-1.0-client.jar]
[INFO] Could not find manifest file:
C:\dev\m2book\code\j2ee\daytrader\ear\src\main\application\
META-INF\MANIFEST.MF - Generating one
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\ear\
target\daytrader-ear-1.0.ear
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\ear\
target\daytrader-ear-1.0.ear to
C:\[...]\.m2\repository\org\apache\geronimo\samples\
daytrader\daytrader-ear\1.0\daytrader-ear-1.0.ear
```

You should review the generated `application.xml` to prove that it has everything you need:

```
<?xml version="1.0" encoding="UTF-8"?>
<application 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/application_1_4.xsd"
  version="1.4">
  <description>
    DayTrader Stock Trading Performance Benchmark Sample
  </description>
  <display-name>Trade</display-name>
  <module>
    <java>daytrader-streamer-1.0.jar</java>
  </module>
  <module>
    <java>daytrader-wsappclient-1.0.jar</java>
  </module>
  <module>
    <web>
      <web-uri>daytrader-web-1.0.war</web-uri>
      <context-root>/daytrader</context-root>
    </web>
  </module>
  <module>
    <ejb>daytrader-ejb-1.0.jar</ejb>
  </module>
</application>
```

This looks good. The next section will demonstrate how to deploy this EAR into a container.

4.12. Deploying a J2EE Application

You have already learned how to deploy EJBs and WARs into a container individually. Deploying EARs follows the same principle. In this example, you'll deploy the DayTrader EAR into Geronimo. Geronimo is somewhat special among J2EE containers in that deploying requires calling the *Deployer* tool with a deployment plan.

A plan is an XML file containing configuration information such as how to map CMP entity beans to a specific database, how to map J2EE resources in the container, etc. Like any other container, Geronimo also supports having this deployment descriptor located within the J2EE archives you are deploying.

However, it is recommended that you use an external plan file so that the deployment configuration is independent from the archives getting deployed, enabling the Geronimo plan to be modified to suit the deployment environment.

The DayTrader application does not deploy correctly when using the JDK 5 or newer. You'll need to use the JDK 1.4 for this section and the following.

To get started, store the deployment plan in `ear/src/main/deployment/geronimo/plan.xml`, as shown on Figure 4-2.

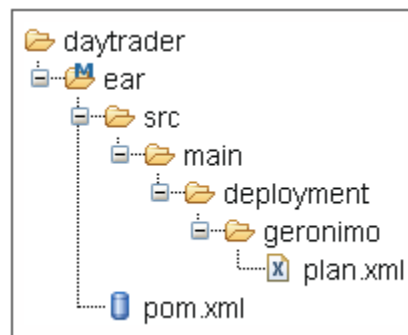


Figure 4-12: Directory structure of the ear module showing the Geronimo deployment plan

How do you perform the deployment with Maven? One option would be to use Cargo as demonstrated earlier in the chapter. You would need the following `pom.xml` configuration snippet:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      <containerId>geronimo1x</containerId>
      <zipUrlInstaller>
        <url>http://www.apache.org/dist/geronimo/1.0/
          geronimo-tomcat-j2ee-1.0.zip</url>
        <installDir>${installDir}</installDir>
      </zipUrlInstaller>
    </container>
    <deployer>
      <deployables>
        <deployable>
          <properties>
            <plan>${basedir}/src/main/deployment/geronimo/plan.xml</plan>
          </properties>
        </deployable>
      </deployables>
    </deployer>
  </configuration>
</plugin>
```

However, in this section you'll learn how to use the Maven Exec plugin. This plugin can execute any process. You'll use it to run the Geronimo Deployer tool to deploy your EAR into a running Geronimo container. Even though it's recommended to use a specific plugin like the Cargo plugin (as described in 4.13 Testing J2EE Applications), learning how to use the Exec plugin is useful in situations where you want to do something slightly different, or when Cargo doesn't support the container you want to deploy into. Modify the `ear/pom.xml` to configure the Exec plugin:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <configuration>
    <executable>java</executable>
    <arguments>
      <argument>-jar</argument>
      <argument>${geronimo.home}/bin/deployer.jar</argument>
      <argument>--user</argument>
      <argument>system</argument>
      <argument>--password</argument>
      <argument>manager</argument>
      <argument>deploy</argument>
      <argument>
        ${project.build.directory}/${project.build.finalName}.ear
      </argument>
      <argument>
        ${basedir}/src/main/deployment/geronimo/plan.xml
      </argument>
    </arguments>
  </configuration>
</plugin>
```


You may have noticed that you're using a `geronimo.home` property that has not been defined anywhere. As you've seen in the EJB and WAR deployment sections above and in previous chapters it's possible to create properties that are defined either in a properties section of the POM or in a Profile. As the location where Geronimo is installed varies depending on the user, put the following profile in a `profiles.xml` or `settings.xml` file:

```
<profiles>
  <profile>
    <id>vmassol</id>
    <properties>
      <geronimo.home>c:/apps/geronimo-1.0-tomcat</geronimo.home>
    </properties>
  </profile>
</profiles>
```

At execution time, the Exec plugin will transform the executable and arguments elements above into the following command line:

```
java -jar c:/apps/geronimo-1.0-tomcat/bin/deployer.jar
-user system -password manager deploy
C:\dev\m2book\code\j2ee\daytrader\ear\target\daytrader-ear-1.0.ear
C:\dev\m2book\code\j2ee\daytrader\ear\src\main\deployment\geronimo\plan.xml
```

First, start your preinstalled version of Geronimo and run `mvn exec:exec`:

```
C:\dev\m2book\code\j2ee\daytrader\ear>mvn exec:exec
[...]
```

```
[INFO] [exec:exec]
[INFO]      Deployed Trade
[INFO]
[INFO]      -> daytrader-web-1.0-SNAPSHOT.war
[INFO]
[INFO]      -> daytrader-ejb-1.0-SNAPSHOT.jar
[INFO]
[INFO]      -> daytrader-streamer-1.0-SNAPSHOT.jar
[INFO]
[INFO]      -> daytrader-wsappclient-1.0-SNAPSHOT.jar
[INFO]
[INFO]      -> TradeDataSource
[INFO]
[INFO]      -> TradeJMS
```

You can now access the DayTrader application by opening your browser to <http://localhost:8080/daytrader/>.

You will need to make sure that the DayTrader application is not already deployed before running the `exec:exec` goal or it will fail. Since Geronimo 1.0 comes with the DayTrader application bundled, you should first stop it, by creating a new execution of the `Exec` plugin or run the following:

```
C:\apps\geronimo-1.0-tomcat\bin>deploy stop
geronimo/daytrader-derby-tomcat/1.0/car
```

If you need to undeploy the DayTrader version that you've built above you'll use the "Trade" identifier instead:

```
C:\apps\geronimo-1.0-tomcat\bin>deploy undeploy Trade
```

4.13. Testing J2EE Application

In this last section you'll learn how to automate functional testing of the EAR built previously. At the time of writing, Maven only supports integration and functional testing by creating a separate module. To achieve this, create a functional-tests module as shown in Figure 4-13.

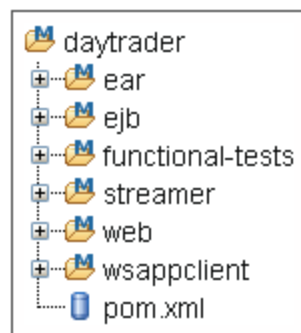


Figure 4-13: The new functional-tests module amongst the other DayTrader modules

You need to add this module to the list of modules in the `daytrader/pom.xml` so that it's built along with the others. Functional tests can take a long time to execute, so you can define a profile to build the functional-tests module only on demand. For example, modify the `daytrader/pom.xml` file as follows:

```
<modules>
  <module>ejb</module>
  <module>web</module>
  <module>streamer</module>
  <module>wsappclient</module>
  <module>ear</module>
</modules>
<profiles>
  <profile>
    <id>functional-test</id>
    <activation>
      <property>
        <name>enableCiProfile</name>
        <value>true</value>
      </property>
    </activation>
    <modules>
      <module>functional-tests</module>
    </modules>
  </profile>
</profiles>
```

For more information on the `ciProfile` configuration, see Chapter 7.

This means that running `mvn install` will not build the `functional-tests` module, but running `mvn install -Pfunctional-test` will.

Now, take a look in the `functional-tests` module itself. Figure 4-1 shows how it is organized:

- Functional tests are put in `src/it/java`,
- Classpath resources required for the tests are put in `src/it/resources` (this particular example doesn't have any resources).
- The Geronimo deployment Plan file is located in `src/deployment/geronimo/plan.xml`.

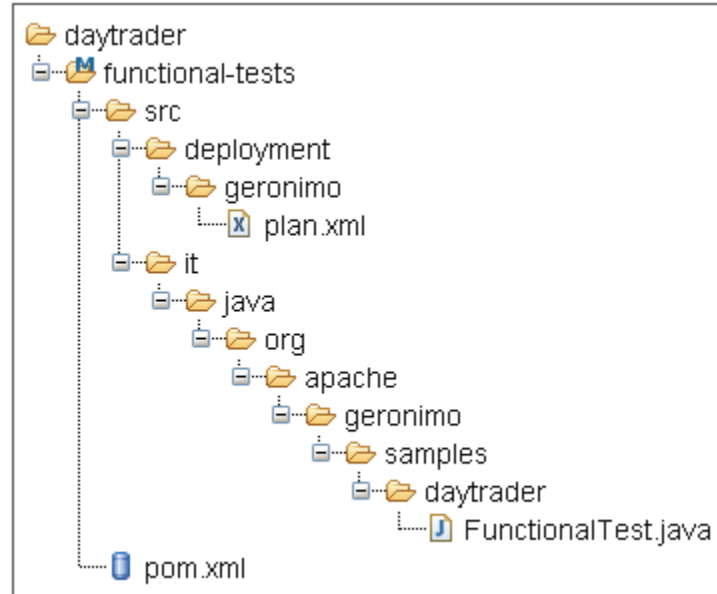


Figure 4-14: Directory structure for the functional-tests module

As this module does not generate an artifact, the packaging should be defined as `pom`. However, the compiler and Surefire plugins are not triggered during the build life cycle of projects with a `pom` packaging, so these need to be configured in the `functional-tests/pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>daytrader-tests</artifactId>
  <name>DayTrader :: Functional Tests</name>
  <packaging>pom</packaging>
  <description>DayTrader Functional Tests</description>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.samples.daytrader</groupId>
      <artifactId>daytrader-ear</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>ear</type>
      <scope>provided</scope>
    </dependency>
    [...]
  </dependencies>
  <build>
    <testSourceDirectory>src/it</testSourceDirectory>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <executions>
          <execution>
            <phase>integration-test</phase>
            <goals>
              <goal>test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
</project>
```

As you can see there is also a dependency on the `daytrader-ear` module. This is because the EAR artifact is needed to execute the functional tests. It also ensures that the `daytrader-ear` module is built before running the functional-tests build when the full DayTrader build is executed from the top-level in `daytrader/`.

For integration and functional tests, you will usually utilize a real database in a known state. To set up your database you can use the DBUnit Java API (see <http://dbunit.sourceforge.net/>). However, in the case of the DayTrader application, Derby is the default database configured in the deployment plan, and it is started automatically by Geronimo. In addition, there's a DayTrader Web page that loads test data into the database, so DBUnit is not needed to perform any database operations.

You may be asking how to start the container and deploy the DayTrader EAR into it. You're going to use the Cargo plugin to start Geronimo and deploy the EAR into it.

As the Surefire plugin's test goal has been bound to the integration-test phase above, you'll bind the Cargo plugin's start and deploy goals to the preintegration-test phase and the stop goal to the post-integration-test phase, thus ensuring the proper order of execution.

Start by adding the Cargo dependencies to the `functional-tests/pom.xml` file:

```
<project>
  [...]
  <dependencies>
    [...]
    <dependency>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-core-uberjar</artifactId>
      <version>0.8</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-ant</artifactId>
      <version>0.8</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Then create an execution element to bind the Cargo plugin's start and deploy goals:

```
<build>
  <plugins>
    [...]
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <configuration>
        <wait>false</wait>
        <container>
          <containerId>geronimo1x</containerId>
          <zipUrlInstaller>
            <url>http://www.apache.org/dist/geronimo/1.0/
              geronimo-tomcat-j2ee-1.0.zip</url>
            <installDir>${installDir}</installDir>
          </zipUrlInstaller>
        </container>
      </configuration>
      <executions>
        <execution>
          <id>start-container</id>
          <phase>pre-integration-test</phase>
          <goals>
            <goal>start</goal>
            <goal>deploy</goal>
          </goals>
          <configuration>
            <deployer>
              <deployables>
                <deployable>
                  <groupId>org.apache.geronimo.samples.daytrader</groupId>
                  <artifactId>daytrader-ear</artifactId>
                  <type>ear</type>
                  <properties>
                    <plan>${basedir}/src/deployment/geronimo/plan.xml</plan>
                  </properties>
                  <pingURL>http://localhost:8080/daytrader</pingURL>
                </deployable>
              </deployables>
            </deployer>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The `deployer` element is used to configure the Cargo plugin's deploy goal. It is configured to deploy the EAR using the Geronimo Plan file. In addition, a `pingURL` element is specified so that Cargo will ping the specified URL till it responds, thus ensuring that the EAR is ready for servicing when the tests execute.

Last, add an execution element to bind the Cargo plugin's stop goal to the post-integration-test phase:

```
[...]
    <execution>
      <id>stop-container</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
</project>
```

The functional test scaffolding is now ready. The only thing left to do is to add the tests in `src/it/java`.

An alternative to using Cargo's Maven plugin is to use the Cargo Java API directly from your tests, by wrapping it in a JUnit `TestSetup` class to start the container in `setUp()` and stop it in `tearDown()`.

You're going to use the HttpUnit testing framework (<http://httpunit.sourceforge.net/>) to call a Web page from the DayTrader application and check that it's working. Add the JUnit and HttpUnit dependencies, with both defined using a test scope, as you're only using them for testing:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>httpunit</groupId>
  <artifactId>httpunit</artifactId>
  <version>1.6.1</version>
  <scope>test</scope>
</dependency>
```


Next, add a JUnit test class called

`src/it/java/org/apache/geronimo/samples/daytrader/FunctionalTest.java`. In the class, the `http://localhost:8080/daytrader` URL is called to verify that the returned page has a title of “DayTrader”:

```
package org.apache.geronimo.samples.daytrader;

import junit.framework.*;
import com.meterware.httpunit.*;

public class FunctionalTest extends TestCase
{
    public void testDisplayMainPage() throws Exception
    {
        WebConversation wc = new WebConversation();
        WebRequest request = new GetMethodWebRequest(
            "http://localhost:8080/daytrader");
        WebResponse response = wc.getResponse(request);
        assertEquals("DayTrader", response.getTitle());
    }
}
```

It's time to reap the benefits from your build. Change directory into `functional-tests`, type `mvn install` and relax:

```
C:\dev\m2book\code\j2ee\daytrader\functional-tests>mvn install
[...]
[INFO] [cargo:start {execution: start-container}]
[INFO] [cargo:deploy {execution: start-container}]
[INFO] [surefire:test {execution: default}]
[INFO] Setting reports dir: C:\dev\m2book\code\j2ee\daytrader\functional-
tests\target\surefire-reports

-----
T E S T S
-----

[surefire] Running org.apache.geronimo.samples.daytrader.FunctionalTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,531 sec
[INFO] [cargo:stop {execution: stop-container}]
```

4.14. Summary

You have learned from chapters 1 and 2 how to build any type of application and this chapter has demonstrated how to build J2EE applications. In addition you've discovered how to automate starting and stopping containers, deploying J2EE archives and implementing functional tests. At this stage you've pretty much become an expert Maven user! The following chapters will show even more advanced topics such as how to write Maven plugins, how to gather project health information from your builds, how to effectively set up Maven in a team, and more.