JBoss™

A Developer's Notebook™

Norman Richards
& Sam Griffith, Jr.

- Installation
- App deployment
- Database config
- Security
- Logging
- Monitoring

O'REILLY®

CHAPTER 9

Rolling Out JBoss

JBoss comes in a state that is extremely developer friendly. JBoss doesn't use an installer and doesn't require any additional configuration to start. In most cases, your application won't require any JBoss-specific configuration to deploy and run. The default configuration comes with the services most commonly needed, already running and in a state that makes them easily accessible.

In this chapter:

- *Securing the Management Consoles*
- *Securing the JMX Invoker*
- *Removing the HTTP Invokers*
- *Configuring the JMS Invokers*
- *Removing Hypersonic*
- *Configuring Tomcat Connectors*
- *Setting a Root Web Application*
- *Removing the Class Download Service*

It is an ideal world for developers, but when the time comes to put your application in production, you really won't want your application server to be so friendly. You'll want to limit the services provided to just the ones you need, especially when those services offer entry points for the outside world to access the server, and you'll want to make sure the remaining services are configured in a production-friendly manner. In this chapter, we'll walk through the default JBoss installation and see what updates are needed to get a JBoss instance ready to be exposed to the outside world.

As we go through the services, keep in mind that although they require a bit of work, you'll need to do this work only once. You learned how to create a new server configuration in Chapter 1. We recommend that you create a server configuration for your deployment environment. You can use this configuration as the starting point for any installation you do, and you'll be sure that each machine has identically configured services.

For consistency, we'll continue to work from the default configuration in these examples.

Securing the Management Consoles

You've used the JMX Console and Web Console applications extensively throughout the book. Given that you can completely control a JBoss

instance through those applications, you might have found it a bit odd that you never had to specify a username and password to access them.

Both applications are accessible to anyone who has access to your server. If you plan to use those applications, you'll need to perform a few configuration steps to get them ready.

How do I do that?

Before we start, it should be obvious that if you don't need the management consoles, you can get rid of them and not need to worry about whether remote users can access them. The JMX Console lives in *deploy/jmx-console.war* and the Web Console lives in *deploy/management*. If you remove both of those directories, JBoss will forget all about the management applications.

Unfortunately, by removing those directories you also lose the ability to manage your server through them. In some cases this makes perfect sense, but we think the management consoles are useful enough that you'll want to keep them around.

If you decide you want to keep them around, you'll need to link them to a security domain and apply security to the web applications, just as you did for the ToDo application. To make things easier, JBoss provides preconfigured, but commented out, security declarations that you just need to enable.

In the *jmx-console.war* directory, you'll need to edit *WEB-INF/web.xml* by uncommenting the following security constraint:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>HtmlAdaptor</web-resource-name>
        <description>
            An example security config that only allows users with the
            role JBossAdmin to access the HTML JMX console web application
        </description>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>JBossAdmin</role-name>
    </auth-constraint>
</security-constraint>
```

This restricts the entire JMX Console to users with the `JBossAdmin` role. If you recall the security discussion in Chapter 5, you'll remember that

along with this you'll need to link to the security domain in *WEB-INF/jboss-web.xml*:

```
<jboss-web>
    <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

This security domain is already deployed:

```
<application-policy name="jmx-console">
    <authentication>
        <login-module
           code="org.jboss.security.auth.spi.UsersRolesLoginModule"
           flag="required">
            <module-option name="usersProperties">
                props/jmx-console-users.properties
            </module-option>
            <module-option name="rolesProperties">
                props/jmx-console-roles.properties
            </module-option>
        </login-module>
    </authentication>
</application-policy>
```

You saw in Chapter 5 how to replace and customize a security domain. For now, we'll stick with the properties file mechanism. The two properties files mentioned, *jmx-console-users.properties* and *jmx-console-roles.properties*, are in the *conf/props* directory. Keeping the files in a shared location rather than in the web application directory allows you to later share the user information with the Web Console.

The two properties files declare a user named `admin` whose password is `admin`. You might want to take the time to change that now. Keeping the default name and password for the JMX Console is nearly as bad as not protecting it at all.

That's all you have to do for the JMX Console. Now we'll make the same changes to the Web Console. You'll have to look all the way down in *management/console-mgr.sar/web-console.war/WEB-INF* to find the configuration files for the Web Console.

You'll need to uncomment the security constraint in *web.xml*, and set a security domain in *jboss-web.xml*. Instead of uncommenting the `java:/jaas/web-console` security domain currently referenced, set the security domain to `java:/jass/jmx-console`. This will cause both applications to be authenticated from the same security domain, instead of using two separate logins for the applications.

Although you can redeploy the applications, the fastest way to make the changes take hold is to restart the server.

What just happened?

You added user security to the two management console applications. When you access either of them, JBoss will ask you to log in. The *jmx-console-users.properties* and *jmx-console-roles.properties* files in the *conf/props* directory provide the authentication information, but you can change the authentication details in *login-config.xml*.

There is still one problem here. If you access the management consoles over normal HTTP, your username and password are sent in the clear. You should enable SSL, as shown in Chapter 5, and always access the application over HTTPS when connecting over a nontrusted network.

Securing the JMX Invoker

The management consoles are not the only management access points into the system. The shutdown and twiddle scripts are programs that interact with JBoss at a management level. They don't use the web applications. Instead, they interact with the JMX invoker.

An invoker is a service in JBoss that accepts *invocations*, which are requests for actions to be performed on the server, from external clients. To interact with the JMX microkernel, you use the JMX invoker.

Whenever you hear the term *invoker*, you should immediately wonder whether you really need to accept a particular kind of request from the outside world. If you don't, it is usually best to simply remove the invoker. You can remove the JMX invoker if you will never need to perform a remote shutdown or any sort of programmatic remote management. However, since those management actions are quite useful, we think it is better to leave the JMX invoker active and apply security to it.

How do I do that?

The JMX invoker service is configured in *deploy/jmx-invoker-service.xml* and is quite simple to secure. The invoke operation on the `jboss.jmx:type=adaptor,name=Invoker` managed bean (MBean) needs to have an authentication interceptor added to it that links to the appropriate security domain. That doesn't sound easy, but the appropriate interceptor is already in the file, waiting to be uncommented:

```
<descriptors>
    <interceptors>
        <interceptor
            code="org.jboss.jmx.connector.invoker.AuthenticationInterceptor"
            securityDomain="java:/jaas/jmx-console"/>
```

```
        </interceptors>
    </descriptors>
```

This requires the user to be authenticated by the `java:/jaas/jmx-console` security domain to invoke management operations remotely. To be sure the interceptor is working, try to shut down the server using the shutdown command:

```
 [bin]$ ./shutdown.sh -S
Exception in thread "main" java.lang.SecurityException: Failed to
authenticate principal=null, securityDomain=jmx-console
        at org.jboss.jmx.connector.invoker.AuthenticationInterceptor.invoke(
           AuthenticationInterceptor.java:76)
        at org.jboss.mx.server.Invocation.invoke(Invocation.java:74)
        at ...
```

The shutdown script will fail unless you specify a valid username and password to the script. The `-u` and `-p` arguments specify the username and password, respectively:

```
[bin]$ sh ./shutdown.sh -S -u admin -p admin
```

If you have already removed the `admin` user or changed the password, make sure you use the correct username and password.

What just happened?

You secured the JMX invoker from unauthenticated access by attaching an authentication interceptor to it. This interceptor was linked to the `java:/jaas/jmx-console` security domain, allowing you to use the same administrative usernames and passwords you already defined for the JMX Console and Web Console applications.

Removing the HTTP Invokers

Invokers can be scary to have around because they give the outside world a potential access point to your server. When properly secured, they don't cause you much worry, but you do have to know they are there before you can make sure they are secure.

Invokers that listen on a specific port for remote connections are easy to track down. Standard networking tools such as `nmap`, `lsof`, and `netstat` can tell you what ports are in use on a machine. What isn't so easy is tracking down invokers that tunnel over other protocols, HTTP being the primary offender.

HTTP invocations are important. Oftentimes overzealous network administrators limit the access between two networks to port 80. That forces you, the developer, to find ways to tunnel application access over that

one port. JBoss can perform every type of remote access over port 80: JNDI, JMX invocations, EJB invocations, etc....

JBoss is configured to allow access over HTTP by default; however, most applications don't need to tunnel over port 80. In that case, having the HTTP invokers is more of a liability, and the best policy is to just remove them.

How do I do that?

There's that pesky invoker word again.

Removing the HTTP invokers is easy. The HTTP invokers are managed by a single service, *http-invoker.sar*. Just remove that entire directory from the deploy directory and you will be all set, knowing that remote clients cannot access through the HTTP invoker back door.

Configuring the JMS Invokers

We didn't use messaging in the ToDo application, but messaging is an important part of many enterprise applications. JBoss supports JMS, the Java Message Service, and provides a fully spec-compliant JMS implementation, complete with invokers for accessing JMS destinations outside of the application server.

Fortunately, setting up JMS for production use is quite easy. You need to know only two things: whether your applications uses JMS, and whether external applications use your JMS. With those two pieces of information, you can configure the JMS invokers.

How do I do that?

All of the JMS services are configured under the *deploy/jms* directory. If you aren't using JMS at all, all you need to do is delete the entire *jms* directory. That will remove not only the JMS invokers, but also the entire JMS subsystem. If you aren't writing a messaging application, there's no need to take up valuable memory and processor time tending to those services.

JMS is common enough that we can't just leave it at that. If you do need to keep messaging around, you'll need to decide which JMS invokers to keep around.

The JMS invokers are the services in the *jms* directory. The first is *jvm-il-service.xml*. This invocation layer is used only inside the application

servers. It doesn't provide any access to the outside world, so don't worry about leaving it around.

Next is *jbossmq-httpil.sar*. This is the HTTP invoker for JMS. If you need external clients that tunnel over HTTP, this invoker will save your life. But you likely won't need it, so just remove this service and don't worry about it.

The last invoker is *uil2-service.xml*. If you are curious, that stands for unified invocation layer, version 2. This gives access to JBoss messaging queues. If you have no external clients accessing your destinations, you can safely remove the service. Otherwise, you'll have to leave it and make sure your destinations are properly secured.

Destinations are configured in *jbossmq-destinations.xml* in *deploy/jms*. Here's the configuration for a topic:

```
<mbean code="org.jboss.mq.server.jmx.Topic"
         name="jboss.mq.destination:service=Topic,name=testTopic">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
        <security>
            <role name="guest"        read="true" write="true"/>
            <role name="publisher"    read="true" write="true"
                                      create="false"/>
            <role name="durpublisher" read="true" write="true"
                                      create="true"/>
        </security>
    </attribute>
  </mbean>
```

The SecurityConf attribute defines the permissions for each logical role you are interested in. The read and write permissions correspond to the ability to read messages from and send messages to a destination. The create permission is the ability to create a durable subscription to a destination.

The SecurityManager attribute links to the security interceptor defined in jbossmq-service:

```
<mbean code="org.jboss.mq.security.SecurityManager"
       name="jboss.mq:service=SecurityManager">
    <attribute name="DefaultSecurityConfig">
        <security>
            <role name="guest" read="true" write="true" create="true"/>
        </security>
    </attribute>
```

```
        <attribute name="SecurityDomain">java:/jaas/jbossmq</attribute>
        <depends optional-attribute-name="NextInterceptor">
            jboss.mq:service=DestinationManager
        </depends>
    </mbean>
```

The security manager defines both the default security configuration, if one isn't defined on the destination in *jbossmq-destinations.xml*, and the security domain to be used to verify JMS permissions. `java:/jaas/jbossmq` is already defined in *login-config.xml*:

```
    <application-policy name="jbossmq">
        <authentication>
            <login-module
               code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
               flag="required">
                <module-option name="unauthenticatedIdentity">
                   guest
                </module-option>
                <module-option name="dsJndiName">java:/DefaultDS</module-option>
                <module-option name="principalsQuery">
                    SELECT PASSWD FROM JMS_USERS WHERE USERID=?
                </module-option>
                <module-option name="rolesQuery">
                    SELECT ROLEID, 'Roles' FROM JMS_ROLES WHERE USERID=?
                </module-option>
            </login-module>
        </authentication>
    </application-policy>
```

This is a standard `DatabaseServerLoginModule`. You saw how to configure it in Chapter 5. What's interesting about this configuration is the use of the `unauthenticatedIdentity` option. This says to assign unauthenticated users the `guest` role. When it's combined with the `DefaultSecurityConf` of the JMS security manager, anyone can access your JMS destinations. You'll want to remove the permissions for the `guest` role if you have remote access into JMS.

What just happened?

You saw the necessary steps to configure JMS in a production system. If you weren't using a service, you removed it completely. Otherwise, you had to choose which of the three invokers you needed. To allow remote JMS clients, you had to keep the UIL2 service, which required you to pay careful attention to the security configuration.

Removing Hypersonic

Hypersonic is a great database for testing an application, but it isn't production ready. In Chapter 4, you saw how to switch to another database. If you aren't using Hypersonic, you can remove the entire service. Hypersonic doesn't listen for connections from the outside world unless you ask it to, so you don't need to worry about securing it. However, you might not want an extra relational database hanging around in memory. It's fairly lightweight, but you'll probably be better off just getting rid of it.

How do I do that?

The *hsqldb-ds.xml* file not only configures the Hypersonic datasources, but also it has the MBeans that control the entire embedded database service. Removing this file will remove all traces of Hypersonic.

It sounds trivial, and it would be except that several other services depend on Hypersonic. We'll go through the changes now, assuming you have created a MySQL database as shown in Chapter 4, and that the ToDo application is already switched over to using that.

The first service is the EJB timer service, in ejb-deployer.xml. You should replace the reference to `DefaultDS` with a reference to the correct datasource. In Chapter 4, we chose the name `MySqlDS`. The change is simple:

```
<!-- A persistence policy that persists timers to a database -->
<mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
       name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
    <!-- DataSource JNDI name -->
    <depends optional-attribute-name="DataSource">
        jboss.jca:service=DataSourceBinding,name=MySqlDS
    </depends>
    <!-- The plugin that handles database persistence -->
    <attribute name="DatabasePersistencePlugin">
        org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin
    </attribute>
</mbean>
```

JMS also requires persistence changes to be made. If you aren't using JMS and you chose to remove it from your configuration, you can stop here. Otherwise, go to the *deploy/jms* directory and look `for` *hsqldb-jdbc2-service.xml* and *hsqldb-jdbc-state-service.xml*.

hsqldb-jdbc2-service.xml controls message persistence. You can remove it completely and replace it with a database-specific file from *docs/examples/jms*. For MySQL, copy *mysql-jdbc2-service.xml* into the *jms* directory.

hsqldb-jdbc-state-service.xml isn't quite so simple. There are no database-specific templates for this. You'll need to update the datasource dependency manually:

```
<mbean code="org.jboss.mq.sm.jdbc.JDBCStateManager"
       name="jboss.mq:service=StateManager">
    <depends optional-attribute-name="ConnectionManager">
        jboss.jca:service=DataSourceBinding,name=MySqlDS
    </depends>
    <!-- ... -->
</mbean>
```

This datasource reference needs to be the same as the datasource you saw earlier, in the `jbossmq` security domain in *login-config.xml*. You'll want to update that now too:

```
<application-policy name="jbossmq">
    <authentication>
        <login-module
           code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
           flag="required">
            <module-option name="unauthenticatedIdentity">guest
            </module-option>
            <module-option name="dsJndiName">java:/MySqlDS</module-option>
            <module-option name="principalsQuery">
                SELECT PASSWD FROM JMS_USERS WHERE USERID=?
            </module-option>
            <module-option name="rolesQuery">
                SELECT ROLEID, 'Roles' FROM JMS_ROLES WHERE USERID=?
            </module-option>
        </login-module>
    </authentication>
</application-policy>
```

The final dependency on `DefaultDS` is *uuid-key-generator.sar*. If you aren't using the JBoss UUID key generator, remove this service from the deploy directory. If you are, you'll need to update the *META-INF/jboss-service.xml* file to reference the preferred datasource:

```
<mbean code="org.jboss.ejb.plugins.keygenerator.hilo.
HiLoKeyGeneratorFactory"
         name="jboss:service=KeyGeneratorFactory,type=HiLo">
    <!-- ... -->
    <depends optional-attribute-name="DataSource">
        jboss.jca:service=DataSourceBinding,name=MySqlDS
    </depends>
    <!-- ... -->
</mbean>
```

Unfortunately, this service is not provided in exploded form, so you'll have you unpack the archive to get to it.

What just happened?

You removed Hypersonic and updated, or removed, the services that depend on it. All of the persistent services will write their tables into the new database. If you've kept them around and restarted JBoss, you should find several new database tables in your external database:

```
mysql> show tables in jbossdb;
+-------------------+
| Tables_in_jbossdb |
+-------------------+
| Comment           |
| HILOSEQUENCES     |
| JMS_ROLES         |
| JMS_SUBSCRIPTIONS |
| JMS_TRANSACTIONS  |
| JMS_USERS         |
| TIMERS            |
| Task              |
| jms_messages      |
+-------------------+
9 rows in set (0.14 sec)
```

You could have saved yourself some trouble by creating a replacement `DefaultDS`. Then it wouldn't have been necessary to change all the datasource references. However, it's important to be explicit about your configuration in a production machine. The extra effort is worth the peace of mind of knowing that all your persistent services are configured correctly.

Configuring Tomcat Connectors

In Tomcat, connectors allow web requests to come in from the outside world. Conceptually, they are like the invokers you saw earlier in this chapter, only they accept HTTP requests against web applications and not general invocation requests against services in JBoss. HTTP connectors aren't so innocuous when there are web applications such as the HTTP invoker, which translates HTTP requests into service invocations. However, you don't need to be concerned about HTTP connectors. The actual web applications behind them are the concern. Your only concern with connectors is to have the correct ones enabled and to have them listening on the right ports.

How do I do that?

Two connectors are defined in *jbossweb-tomcat55.sar/server.xml*. The first is the AJP connector that listens for requests proxied from an Apache server running mod_jk:

```
<!-- A AJP 1.3 Connector on port 8009 -->
<Connector port="8009" address="${jboss.bind.address}"
           emptySessionPath="true" enableLookups="false"
           redirectPort="8443" protocol="AJP/1.3"/>
```

This connector listens on port 8009, but since external clients do not see this port, it doesn't need to be changed. If you aren't using mod_jk, you can remove this connector to save resources.

The normal Tomcat service listens for HTTP requests on port 8080. The connector looks like this:

```
<Connector port="8080" address="${jboss.bind.address}"
           maxThreads="250" strategy="ms" maxHttpHeaderSize="8192"
           emptySessionPath="true"
           enableLookups="false" redirectPort="8443" acceptCount="100"
           connectionTimeout="20000" disableUploadTimeout="true"/>
```

If you have a Unix box and you change the port to something below 1024, you'll need to be running JBoss as root.

If your web server is directly visible to the outside world, you'll want to use the standard port 80, unless you have a frontend load balancer that can redirect the standard port 80 to 8080 on your machine. To do that, change the `port` attribute to `80`.

There is a third connector you might add, the SSL connector. Chapter 5 showed you how to enable SSL, but you used port 8443 at the time. The standard HTTPS port is 443, so you'll want to change the port number on that connector. That SSL connector looks like this:

```
<Connector port="443" address="${jboss.bind.address}"
           maxThreads="100" strategy="ms" maxHttpHeaderSize="8192"
           emptySessionPath="true" scheme="https"
           secure="true" clientAuth="false"
           keystoreFile="${jboss.server.home.dir}/conf/ssl.keystore"
           keystorePass="mypassword" keyAlias="testkey1"
           sslProtocol="TLS" />
```

If you do use a different SSL port, you will need to make sure you change the redirect port of the HTTP connector port to be the SSL port (443). Both the HTTP and AJP connectors have a `redirectPort` attribute that should be set to the right value:

```
<Connector port="80" address="${jboss.bind.address}"
           maxThreads="250" strategy="ms" maxHttpHeaderSize="8192"
           emptySessionPath="true"
           enableLookups="false" redirectPort="443" acceptCount="100"
           connectionTimeout="20000" disableUploadTimeout="true"/>
```

Restart JBoss to make these changes take effect. You should be able to access the server at *http://localhost/* with no port number listed in the URL.

What just happened?

You saw where the Tomcat connectors are configured, and you changed the ports Tomcat uses for HTTP requests. There weren't any direct security concerns around the connectors. All you needed to worry about was making the right connectors available on the right ports.

What about...

...not being able to access the server on port 80?

Check the console log for errors as you start up. You may have another web server running on port 80 on your machine, or you may not be running as a user with permissions to access a privileged port such as port 80. The log messages will help you determine what is stopping JBoss from using port 80.

Setting a Root Web Application

When you access JBoss for the first time, JBoss provides you with the default root application, shown in Figure 9-1.

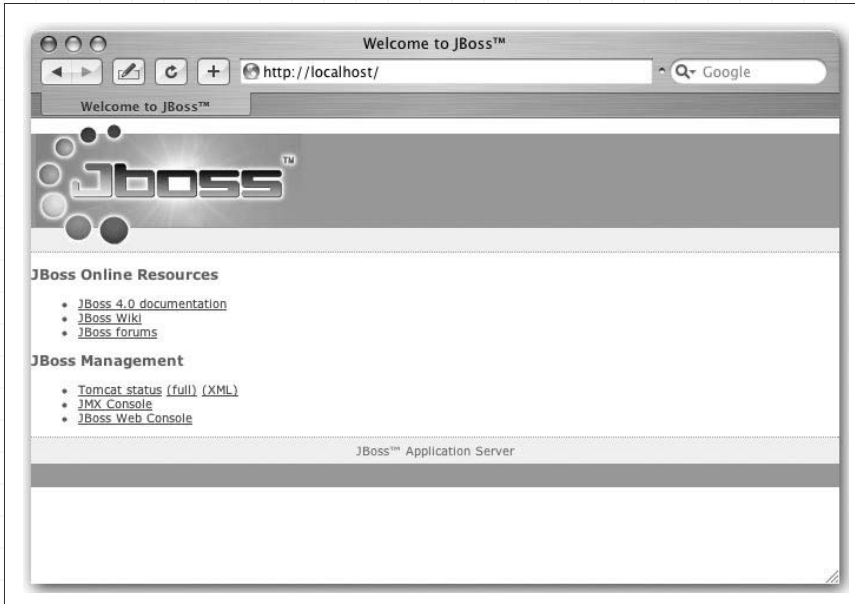

Figure 9-1. Default root application

In production, you'll want your own application to serve the root context.

How do I do that?

Controlling the context of a web application is different for WAR files deployed inside of an EAR file and for WAR files deployed as standalone files.

If you didn't change the HTTP port to 80, make sure you use http://localhost:8080/.

EAR files contain a standard J2EE mechanim for controlling the context of the web application. The `context-root` element of a web module in *application.xml* controls the root web application. Setting the root context to / will place the ToDo application at the root context. In the ToDo application the change will look like this:

```
<application xmlns="http://java.sun.com/xml/ns/j2ee" version="1.4"
    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">
    <display-name>JBoss Notebook ToDo Application</display-name>
    <description>JBoss Notebook ToDo Application</description>
    <module>
        <ejb>todo.jar</ejb>
    </module>
    <module>
        <web>
            <web-uri>todo.war</web-uri>
            <context-root>/</context-root>
        </web>
    </module>
</application>
```

If you make this change and redeploy the application, accessing *http://localhost/* will bring you directly to the ToDo application.

JBoss won't complain if you have multiple applications wanting to serve the root context, but only the most recently deployed one will be active.

For a standalone WAR file, the context-root element should be added to *jboss-web.xml*. We'll go back to the quote machine web application from Chapter 2. Create a *jboss-web.xml* file in `quote/src/metadata` that looks like this:

```
<!DOCTYPE jboss-web PUBLIC
          "-//JBoss//DTD Web Application 2.4//EN"
          "http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">
<jboss-web>
        <context-root>/</context-root>
</jboss-web>.
```

Now that you've done that, you can redeploy your application. When you go to *http://localhost/*, you'll see the quote application running.

What just happened?

You saw how to set an application to serve the root context in JBoss. With EAR files, declaring the context root can be done in an application-server-independent way. However, with a standalone WAR file, you had to use the *jboss-web.xml* file to declare the context.

Removing the Class Download Service

The last service you'll need to worry about is the class download service. The class download service allows remote RMI clients to download Java classes from the server. It's a nice idea because then you don't have to distribute your server classes to your external clients.

However, leaving the class download service running exposes anything in the server's classpath to external clients. Try accessing *http://localhost:8083/login-config.xml*, or worse yet, *http://localhost:8083/props/jmx-console-users.properties*, to see the potential problems with the class download service.

How do I do that?

To remove the ability to download class files, you need to edit *conf/jboss-service.xml* and remove the `jboss:service=WebService` MBean. It looks like this:

```
<!-- ==================================================== -->
<!-- Class Loading                                        -->
<!-- ==================================================== -->

<mbean code="org.jboss.web.WebService"
    name="jboss:service=WebService">
    <attribute name="Port">8083</attribute>
    <!-- Should resources and non-EJB classes be downloadable -->
    <attribute name="DownloadServerClasses">true</attribute>
    <attribute name="Host">${jboss.bind.address}</attribute>
    <attribute name="BindAddress">${jboss.bind.address}</attribute>
</mbean>
```

If you do really need the class download service for remote clients, you should at least set `DownloadServerClasses` to `false` to restrict the service to supply only EJB-related classes.

What just happened?

You saw how to remove the class download service. If you really need the class download service to facilitate the process of managing external clients, you should set `DownloadServerClasses` to `false` to limit the amount of server-side information exposed to clients.