

JBoss Administration and Development

Second Edition

**Scott Stark and
The JBoss Group**

© JBoss Group, LLC
2520 Sharondale Dr.
Atlanta, GA 30305 USA
sales@jbossgroup.com

Last Updated: September 30, 2002 8:04 am

Contents

: List of Listings - - - - -	ix
: List of Figures - - - - -	xiii

CHAPTER I *Preface* 15

i: Forward - - - - -	15
i: About the Authors - - - - -	16
i: About Open Source - - - - -	17
i: About JBoss - - - - -	17
JBoss: A Full J2EE Implementation with JMX - - - - -	18
What this Book Covers - - - - -	18

CHAPTER 1 *Installing and Building the JBoss Server* 21

1: Getting the Binary - - - - -	22
Prerequisites - - - - -	22
Installing the Binary Package - - - - -	22
Directory Structure - - - - -	22
- - - - -	24
The Default Server Configuration File Set - - - - -	25
conf/auth.conf - - - - -	26
conf/jboss-minimal.xml - - - - -	27
conf/jboss-server.xml - - - - -	27
conf/jbossmq-state.xml - - - - -	27
conf/jndi.properties - - - - -	27
conf/log4j.xml - - - - -	27
conf/login-config.xml - - - - -	27
conf/server.policy - - - - -	27
conf/standardjaws.xml - - - - -	27
conf/standardjbosscmp-jdbc.xml - - - - -	28
conf/standardjboss.xml - - - - -	28
deploy/counter-service.xml - - - - -	28
deploy/ejb-management.jar - - - - -	28
deploy/hsqldb-service.xml - - - - -	28
deploy/jboss-local-jdbc.rar - - - - -	28
deploy/jboss-xa.rar - - - - -	28
deploy/jbossmq-destinations-service.xml - - - - -	28
deploy/jbossmq-service.xml - - - - -	28
deploy/jbossweb.sar/ - - - - -	29
deploy/jca-service.xml - - - - -	29
deploy/jms-ra.rar - - - - -	29
deploy/jms-service.xml - - - - -	29
deploy/jmx-console.war/ - - - - -	29
deploy/jmx-ejb-adaptor.jar, deploy/jmx-ejb-connector-server.sar - - - - -	29
deploy/jmx-rmi-adaptor.sar - - - - -	29
deploy/mail-service.xml - - - - -	29
deploy/properties-service.xml - - - - -	29
deploy/scheduler-service.xml - - - - -	30
deploy/user-service.xml - - - - -	30

Basic Installation Testing	- - - - -	30
Building the Server from Source Code	- - - - -	31
Accessing the JBoss CVS Repositories at SourceForge	- - - - -	32
Understanding CVS	- - - - -	32
Anonymous CVS Access	- - - - -	32
Obtaining a CVS Client	- - - - -	32
Building the JBoss-3.0.2 Distribution Using the Source Code	- - - - -	33
Building the JBoss-3.0.2 Distribution Using the CVS Source Code	- - - - -	33
Building the JBoss-3.0.3/Tomcat-4.0.5 Integrated Bundle	- - - - -	34
An Overview of the JBoss CVS Source Tree	- - - - -	34
Using the JBossTest unit testsuite	- - - - -	35

CHAPTER 2 *The JBoss JMX Microkernel* 39

2: JMX	- - - - -	39
An Introduction to JMX	- - - - -	40
Instrumentation Level	- - - - -	41
Agent Level	- - - - -	42
Distributed Services Level	- - - - -	42
JMX Component Overview	- - - - -	43
<i>Managed Beans or MBeans</i>	- - - - -	43
<i>Notification Model</i>	- - - - -	44
<i>MBean Metadata Classes</i>	- - - - -	44
<i>MBean Server</i>	- - - - -	44
<i>Agent Services</i>	- - - - -	45
JBoss JMX Implementation Architecture	- - - - -	46
The JBoss ClassLoader Architecture	- - - - -	46
<i>Scoping Classes Using EARs</i>	- - - - -	48
Connecting to the JMX Server	- - - - -	49
Inspecting the Server - the JMX Console Web Application	- - - - -	49
<i>Securing the JMX Console</i>	- - - - -	50
Connecting to JMX Using RMI	- - - - -	52
JBoss and JMX	- - - - -	59
The Startup Process	- - - - -	59
<i>The Service Life Cycle Interface</i>	- - - - -	64
<i>The ServiceController MBean</i>	- - - - -	65
<i>Specifying Service Dependencies</i>	- - - - -	67
<i>Identifying Unsatisfied Dependencies</i>	- - - - -	69
Writing A JBoss MBean Service	- - - - -	69
<i>A Custom MBean Example</i>	- - - - -	70
The Core JBoss MBeans	- - - - -	74
<i>org.jboss.logging.Log4jService</i>	- - - - -	78
<i>org.jboss.web.WebService</i>	- - - - -	78
<i>org.jboss.deployment.scanner.URLDeploymentScanner</i>	- - - - -	78
Deployment Ordering and Dependencies	- - - - -	79
The JBoss Deployer Architecture	- - - - -	91

CHAPTER 3 *Naming on JBoss - The JNDI Naming Service* 95

3: An Overview of JNDI	- - - - -	95
The JNDI API	- - - - -	96
Names	- - - - -	96

Contexts	97
<i>Obtaining a Context using InitialContext</i>	97
J2EE and JNDI – The Application Component Environment	98
ENC Usage Conventions	99
<i>The ejb-jar.xml ENC Elements</i>	100
<i>The web.xml ENC Elements</i>	101
<i>The jboss.xml ENC Elements</i>	103
<i>The jboss-web.xml ENC Elements</i>	104
<i>Environment Entries</i>	105
<i>EJB References</i>	106
<i>EJB References with jboss.xml and jboss-web.xml</i>	108
<i>EJB Local References</i>	109
<i>Resource Manager Connection Factory References</i>	111
<i>Resource Manager Connection Factory References with jboss.xml and jboss-web.xml</i>	112
<i>Resource Environment References</i>	113
<i>Resource Environment References and jboss.xml, jboss-web.xml</i>	114
3-15: The JBossNS Architecture	115
The Naming InitialContext Factories	117
The HTTP InitialContext Factory Implementation	118
The Login InitialContext Factory Implementation	119
Accessing JNDI over HTTP	120
Securing Access to JNDI over HTTP	122
Securing Access to JNDI with a Read-Only Unsecured Context	124
Additional Naming MBeans	125
org.jboss.naming.ExternalContext MBean	125
The org.jboss.naming.NamingAlias MBean	127
The org.jboss.naming.JNDIView MBean	128

CHAPTER 4 *Transactions on JBoss - The JTA Transaction Service* 133

4: Transaction/JTA Overview	133
Pessimistic and optimistic locking	134
The components of a distributed transaction	135
The two-phase XA protocol	135
Heuristic exceptions	136
Transaction IDs and branches	136
•: JBoss Transaction Internals	137
Adapting a Transaction Manager to JBoss	137
The Default Transaction Manager	138
org.jboss.tm.XidFactory	138
UserTransaction Support	139

CHAPTER 5 *EJBs on JBoss - The EJB Container Configuration and Architecture* 141

5: The EJB Client Side View	141
•: The EJB Server Side View	145
Detached Invokers - The Transport Middlemen	145
The LocalInvoker - In VM transport	146
The JRMPInvoker - RMI/JRMP Transport	147
The HttpInvoker - RMI/HTTP Transport	148
The HA JRMPInvoker - Clustered RMI/JRMP Transport	149

5-5: The EJB Container	150
EJBDeployer MBean	151
Verifying EJB deployments	151
Deploying EJBs Into Containers	152
Container configuration information	152
<i>The container-name Element</i>	157
<i>The call-logging Element</i>	157
<i>The container-invoker and container-invoker-conf Elements</i>	157
<i>The container-interceptors Element</i>	158
<i>The instance-pool and container-pool-conf Elements</i>	158
<i>The instance-cache and container-cache-conf Elements</i>	159
<i>The persistence-manager Element</i>	160
<i>The transaction-manager Element</i>	160
<i>The locking-policy Element</i>	161
<i>The commit-option and optiond-refresh-rate Element</i>	161
<i>The security-domain Element</i>	162
Container Plug-in Framework	162
org.jboss.ejb.ContainerPlugin	162
org.jboss.ejb.Interceptor	163
org.jboss.ejb.InstancePool	164
org.jboss.ebj.InstanceCache	165
org.jboss.ejb.EntityPersistenceManager	166
org.jboss.ejb.StatefulSessionPersistenceManager	172
5-14: Entity Bean Locking and Deadlock Detection	173
Why JBoss Needs Locking	174
Entity Bean Lifecycle	174
Default Locking Behavior	174
Method Lock	175
Transaction Lock	175
Pluggable Interceptors and Locking Policy	175
Deadlock	176
Deadlock Detection	177
<i>Catching ApplicationDeadlockException</i>	178
Advanced Configurations and Optimizations	178
Short-lived Transactions	179
Ordered Access	179
Read-Only Beans	179
Explicitly Defining Read-Only Methods	179
Instance Per Transaction Policy	180
Running Within a Cluster	182
Troubleshooting	182
Locking Behavior Not Working	183
IllegalStateException	183
Hangs and Transaction Timeouts	183
6: JMS Examples	185
A Point-To-Point Example	186
A Pub-Sub Example	188
A Pub-Sub With Durable Topic Example	194
A Point-To-Point With MDB Example	197

CHAPTER 6 *Messaging on JBoss - JMS Configuration and Architecture* 185

6: JMS Examples	185
A Point-To-Point Example	186
A Pub-Sub Example	188
A Pub-Sub With Durable Topic Example	194
A Point-To-Point With MDB Example	197

•: JBoss Messaging Overview	204
Invocation Layer	205
RMI IL	205
OIL IL	205
UIL IL	205
JVM IL	205
Security Manager	206
Destination Manager	206
Message Cache	206
State Manager	206
Persistence Manager	206
File PM	206
Rolling Logged PM	207
JDBC2 PM	207
Destinations	207
Queues	207
Topics	207
•: JBoss Messaging Configuration and MBeans	208
org.jboss.mq.il.jvm.JVMServerILService	208
org.jboss.mq.il.rmi.RMIServerILService	209
org.jboss.mq.il.oil.OILServerILService	209
org.jboss.mq.il.ul.UILServerILService	210
org.jboss.mq.server.jmx.Invoker	210
org.jboss.mq.server.jmx.InterceptorLoader	210
org.jboss.mq.security.SecurityManager	211
org.jboss.mq.server.jmx.DestinationManager	211
org.jboss.mq.server.MessageCache	212
org.jboss.mq.pm.file.CacheStore	212
org.jboss.mq.sm.file.DynamicStateManager	212
org.jboss.mq.pm.file.PersistenceManager	212
org.jboss.mq.pm.rollinglogged.PersistenceManager	213
org.jboss.mq.pm.jdbc2.PersistenceManager	213
Destination MBeans	214
org.jboss.mq.server.jmx.Queue	214
org.jboss.mq.server.jmx.Topic	214
Destination Security Configuration	215
Administration Via JMX	215
Creating Queues At Runtime	216
Creating Topics At Runtime	216
Managing a JBossMQ User IDs at Runtime	216
Checking how many messages are on a Queue	217
Checking to see how the Message Cache is performing	217

CHAPTER 7 *Connectors on JBoss - The JCA Configuration and Architecture* 219

7: JCA Overview	219
7-2: An Overview of the JBossCX Architecture	223
BaseConnectionManager2 MBean	224
RARDeployment MBean	225
JBossManagedConnectionPool MBean	226
CachedConnectionManager MBean	226

A Sample Skeleton JCA Resource Adaptor	226
Example Configurations	234

CHAPTER 8 Security on JBoss - J2EE Security Configuration and Architecture 237

8: J2EE Declarative Security Overview	237
Security References	239
Security Identity	240
Security roles	241
EJB method permissions	242
Web content security constraints	245
Enabling Declarative Security in JBoss	246
8-5: An Introduction to JAAS	246
What is JAAS?	246
The JAAS Core Classes	247
<i>Subject and Principal</i>	247
<i>Authentication of a Subject</i>	248
6: The JBoss Security Model	251
Enabling Declarative Security in JBoss Revisited	254
8-8: The JBoss Security Extension Architecture	259
How the JaasSecurityManager Uses JAAS	261
The JaasSecurityManagerService MBean	264
An Extension to JaasSecurityManager, the JaasSecurityDomain MBean	266
An XML JAAS Login Configuration MBean	266
The JAAS Login Configuration Management MBean	268
Using and Writing JBossSX Login Modules	269
org.jboss.security.auth.spi.IdentityLoginModule	269
org.jboss.security.auth.spi.UsersRolesLoginModule	270
org.jboss.security.auth.spi.LdapLoginModule	271
org.jboss.security.auth.spi.DatabaseServerLoginModule	274
org.jboss.security.auth.spi.ProxyLoginModule	276
org.jboss.security.auth.spi.RunAsLoginModule	276
org.jboss.security.ClientLoginModule	276
Writing Custom Login Modules	277
Support for the Subject Usage Pattern	278
A Custom LoginModule Example	282
8-13: The Secure Remote Password (SRP) Protocol	286
Providing Password Information for SRP	290
Inside of the SRP algorithm	292
An SRP example	295
8-17: Running JBoss with a Java 2 security manager	298
8-19: Using SSL with JBoss using JSSE	300

CHAPTER 9 Integrating Servlet Containers 307

9: The AbstractWebContainer Class	307
The AbstractWebContainer Contract	308
Creating an AbstractWebContainer Subclass	313
Use the Thread Context Class Loader	313
Integrate Logging Using log4j	314

Delegate web container authentication and authorization to JBossSX - - - - -	314
9-3: JBoss/Tomcat-4.x bundle notes - - - - -	315
The Embedded Tomcat Configuration Elements - - - - -	317
Server - - - - -	317
Service - - - - -	317
Connector - - - - -	317
<i>The HTTP Connector</i> - - - - -	318
<i>The AJP Connector</i> - - - - -	318
<i>The Warp Connector</i> - - - - -	318
Engine - - - - -	318
Host - - - - -	319
<i>Alias</i> - - - - -	319
DefaultContext - - - - -	319
<i>Manager</i> - - - - -	319
Logger - - - - -	320
Valve - - - - -	320
Listener - - - - -	320
Using SSL with the JBoss/Tomcat bundle - - - - -	320
Setting up Virtual Hosts with the JBoss/Tomcat-4.x bundle - - - - -	325
Using Apache with the JBoss/Tomcat-4.x bundle - - - - -	327
Using Clustering - - - - -	328
5: JBoss/Jetty-4.0.0 Bundle Notes - - - - -	329
Integration with JBoss - - - - -	329
Deployment - - - - -	330
Configuration - - - - -	330
<i>Unpacking wars on deployment</i> - - - - -	333
<i>Classloading behaviour</i> - - - - -	333
<i>Changing the default HTTP listener port</i> - - - - -	334
<i>Changing other HTTP listener port attributes</i> - - - - -	334
<i>Using SSL</i> - - - - -	334
<i>Using JAAS</i> - - - - -	335
<i>Using Distributed HttpSession</i> - - - - -	336
Other Jetty Configuration Tips - - - - -	337
Deploying a war to context '/' - - - - -	337
Using virtual hosts - - - - -	337
<i>Running on port 80</i> - - - - -	337
<i>Running with Apache front-ending Jetty</i> - - - - -	337

CHAPTER 10 MBean Services Miscellany 339

10: System Properties Management - - - - -	339
10-1: Property Editor Management - - - - -	340
•: Scheduling Tasks - - - - -	341
org.jboss.varia.scheduler.Scheduler - - - - -	341

Appendix A The JBoss Group and Our LGPL License 345

A: About The JBoss Group - - - - -	345
A: The GNU Lesser General Public License (LGPL) - - - - -	345

Appendix B The JBoss DTDs **357**

B: The jboss_3_0.dtd	- - - - -	357
B: The jbosscmp-jdbc_3_0.dtd DTD	- - - - -	375
B: The jboss-web_3_0.dtd DTD	- - - - -	386
B: The security_config.dtd DTD	- - - - -	388

Appendix C Book Example Installation **391**

List of Listings

1-1 Listing 1-1, the JBoss 3.0.x branch build process.....	33
2-1 An example jboss-app.xml descriptor for enabled scoped class loading at the ear level	48
2-2 The jmx-console.war web.xml and jboss-web.xml descriptors with the security elements uncom- mented.....	51
2-3 The RMIAdaptor interface.....	52
2-4 A JMX client that uses the RMIAdaptor	55
2-5 The org.jboss.system.Service interface.....	64
2-6 Service descriptor fragments illustrating the usage of the depends and depends-list elements.	68
2-7 An example of using the depends element to specify the complete configuration of a depended on service	68
2-8 JNDIMapMBean interface and implementation based on the service interface method pattern	70
2-9 JNDIMap MBean interface and implementation based on the ServiceMBean interface and Ser- viceMBeanSupport class.....	72
2-10 The example 1 JNDIMap MBean service descriptor and a client usage code fragment	74
2-11 The default jboss-service.xml configuration file from the standard JBoss distribution	74
2-12 An example ear with an MBean that depends on an EJB	80
2-13 A DynamicMBean service that uses and EJB.....	80
2-14 The standard MBean interface for Listing 2-13.....	86
2-15 The DynamicMBean jboss-service.xml descriptor.....	88
2-16 The org.jboss.deployment.SubDeployer interface.....	93
3-1 A sample jndi.properties file.....	98
3-2 ENC access sample code	99
3-3 An example ejb-jar.xml env-entry fragment.....	106
3-4 ENC env-entry access code fragment	106
3-5 An example ejb-jar.xml ejb-ref descriptor fragment	107
3-6 ENC ejb-ref access code fragment.....	108
3-7 An example jboss.xml ejb-ref fragment.....	109
3-8 An example ejb-jar.xml ejb-local-ref descriptor fragment	110
3-9 ENC ejb-local-ref access code fragment.....	110
3-10 A web.xml resource-ref descriptor fragment.....	111
3-11 ENC resource-ref access sample code fragment.....	112
3-12 A sample jboss-web.xml resource-ref descriptor fragment	113
3-13 An example ejb-jar.xml resource-env-ref fragment.....	113
3-14 ENC resource-env-ref access code fragment	114
3-15 A sample jboss.xml resource-env-ref descriptor fragment	114
3-16 An example web.xml descriptor for secured access to the JNDI servlets	122
3-17 The additional web.xml descriptor elements needed for read-only access.....	124
3-18 ExternalContext MBean configurations	126
5-1 The client-interceptors from the “Standard Stateless SessionBean” configuration.....	144
5-2 The org.jboss.invocation.Invoker interface	145
5-3 A custom JRMPInvoker example that enables compressed sockets for session bean.....	147
5-4 A sample jboss.xml descriptor for enabling RMI/HTTP for a stateless session bean.....	149
5-5 A jboss.xml stateless session configuration for HA-RMI/HTTP	150
5-6 An example of a complex container-configuration element from the server/default/conf/standard-	

jboss.xml file.....	153
5-7 An example of overriding the standardjboss.xml container stateless session beans configuration to enable secured access.	156
5-8 The org.jboss.ejb.ContainerPlugin interface.....	163
5-9 The org.jboss.ejb.Interceptor interface	163
5-10 The org.jboss.ejb.InstancePool interface	164
5-11 The org.jboss.ejb.InstanceCache interface	165
5-12 The org.jboss.ejb.EntityPersistenceManager interface	166
5-13 The org.jboss.ejb.EntityPersistanceStore interface.....	169
5-14 The org.jboss.ejb.StatefulSessionPersistenceManager interface	173
5-15 The “Standard CMP 2.x EntityBean” interceptor definition	175
5-16 The org.jboss.ejb.plugins.lock.BeanLockSupport deadlockDetection method	177
5-17 Marking an entity bean read-only using jboss.xml	179
5-18 Defining entity bean methods as read-only	179
5-19 An example of using the Instance Per Transaction policy available in JBoss 3.0.1+.....	180
5-20 The Instance Per Transaction configuration	181
6-1 A P2P JMS client example	186
6-2 A Pub-Sub JMS client example	188
6-3 A JMS publisher client	191
6-4 A JMS subscriber client.....	192
6-5 A durable topic JMS client example.....	194
6-6 A TextMessage processing MDB	197
6-7 The MDB ejb-jar.xml and jboss.xml descriptors.....	199
6-8 A JMS client that interacts with the TextMDB	200
6-9 The default login-config.xml configuration for JBoss messaging.....	211
6-10 Default SqlProperties	213
6-11 Sample Destination Security Configuration	215
7-1 The nontransactional file system resource adaptor deployment descriptor.....	228
7-2 The nontransactional file system resource adaptor MBeans service descriptor.	229
7-3 The stateless session bean <u>echo</u> method code which shows the access of the resource adaptor connection factory.	232
8-1 An example ejb-jar.xml and web.xml descriptor fragments which illustrate the security-role-ref element usage.....	240
8-2 An example ejb-jar.xml descriptor fragment which illustrates the security-identity element usage.	241
8-3 An example ejb-jar.xml and web.xml descriptor fragments which illustrate the security-role element usage.....	242
8-4 An example ejb-jar.xml descriptor fragment which illustrates the method-permission element usage.	243
8-5 A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements.	245
8-6 An illustration of the steps of the authentication process from the application perspective	249
8-7 The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.	255
8-8 The jboss.xml descriptor which configures the EchoSecurityProxy as the custom security proxy for the EchoBean.	257
8-9 A sample login module configuration suitable for use with XMLLoginConfig	267

8-10 A JndiUserAndPass custom login module.....	283
8-11 The chap8-ex2 secured client access output	285
8-12 The chap8-ex2 server side behavior of the JndiUserAndPass	285
8-13 The chap8-ex2 security domain and login module configuration	286
8-14 The SRPVerifierStore interface	290
8-15 The chap8-ex3 jar contents	295
8-16 The chap8-ex3.sar jboss-service.xml descriptor for the SRP services	296
8-17 The chap8-ex3 client side and server side SRP login module configurations.....	297
8-18 The modifications to the Win32 run.bat start script to run JBoss with a Java 2 security manager. 299	
8-19 The modifications to the UNIX/Linux run.sh start script to run JBoss with a Java 2 security manager.....	299
8-20 A sample JaasSecurityDomain config for RMI/SSL	303
8-21 The jboss-service.xml and jboss.xml configurations to enable SSL with the example 4 stateless session bean	304
9-1 Key methods of the AbstractWebContainer class.	308
9-2 A pseudo-code description of authenticating a user via the JBossSX API and the java:comp/env/ security JNDI context.	314
9-3 A pseudo-code description of authorization a user via the JBossSX API and the java:comp/env/ security JNDI context.	315
9-4 The JaasSecurityDoman and EmbeddedCatalinaSX MBean configurations for setting up Tomcat- 4.x to use SSL as its primary connector protocol.	321
9-5 The JaasSecurityDoman and EmbeddedCatalinaSX MBean configurations for setting up Tomcat- 4.x to use both non-SSL and SSL enabled HTTP connectors.	322
9-6 An example virtual host configuration.	325
9-7 An example jboss-web.xml descriptor for deploying a WAR to the www.starkinternational.com virtual host	326
9-8 Output from the www.starkinternational.com Host component when the Listing 9-7 WAR is deployed.....	327
9-9 An example EmbeddedCatalinaSX MBean configuration that supports integration with Apache using the Ajpv13 protocol connector.	327
9-10 Standard Jetty service configuration file jboss-service.xml	330
9-11 Jetty listener port attributes	334
9-12 Using the SecurityDomainListener configure SSL for Jetty	335
10-1 An example SystemPropertiesService jboss-service descriptor	339
10-2 An example Scheduler jboss-service descriptor	342
11-1 The GNU lesser general public license text.....	345

List of Figures

i-1	the JBoss JMX integration bus and the standard JBossXX components.....	18
1-1	A view of the JBoss server installation directory structure with the default server configuration file set expanded.	23
1-2	an expanded view of the default server configuration file set conf and deploy directories.....	26
1-3	the testsuite CVS module directory structure	35
1-4	An example testsuite run report status html view as generated by the testsuite.	37
2-1	The JBoss JMX integration bus and the standard JBoss components.	40
2-2	The Relationship between the components of the JMX architecture	41
2-3	The JBoss 3.x class loading architecture.	46
2-4	The JBoss JMX console web application agent view	49
2-5	The MBean view for the “jboss.system:type=Server” MBean.....	50
2-6	The jmx-console basic HTTP login dialog presented after making the changes shown in Listing 2-2.	52
2-7	the DTD for the MBean service descriptor parsed by the SARDeployer.....	62
2-8	A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service.	63
2-9	The interaction between the SARDeployer and ServiceController to start a service.	66
2-10	The EjbMBeanAdaptor MBean operations JMX console view	91
2-11	The deployment layer classes	92
3-1	The ENC elements in the standard ejb-jar.xml 2.0 deployment descriptor.	101
3-2	The ENC elements in the standard servlet 2.3 web.xml deployment descriptor.	103
3-3	The ENC elements in the JBoss 3.0 jboss.xml deployment descriptor.	104
3-4	The ENC elements in the JBoss 3.0 jboss-web.xml deployment descriptor.	105
3-5	Key components in the JBossNS architecture.	115
3-6	The HTTP invoker proxy/server structure for a JNDI Context	121
3-7	The HTTP JMX agent view of the configured JBoss MBeans.....	129
3-8	The HTTP JMX MBean view of the JNDIView MBean.	130
3-9	The HTTP JMX view of the JNDIView list operation output.....	131
5-1	The composition of an EJBHome proxy in JBoss.	142
5-2	The jboss.xml descriptor client side interceptor configuration elements.	143
5-3	The transport invoker server side architecture.....	146
5-4	The jboss.xml descriptor container invoker configuration elements.	146
5-5	The jboss_3_0 DTD elements related to container configuration.	153
5-6	The jboss.xml descriptor EJB to container configuration mapping elements	155
5-7	Deadlock definition example	176
7-1	The relationship between a J2EE application server and a JCA resource adaptor	221
7-2	The JCA 1.0 specification class diagram for the connection management architecture.	222
7-3	The JBoss JCA implementation components	223
7-4	The file system RAR class diagram.....	227
7-5	A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor that result when the <u>EchoBean</u> accesses the resource adaptor connection factory.....	233
8-1	A subset of the EJB 2.0 deployment descriptor content model that shows the security related elements.....	238

8-2	A subset of the Servlet 2.2 deployment descriptor content model that shows the security related elements.....	239
8-3	The key security model interfaces and their relationship to the JBoss server EJB container elements.....	252
8-4	The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.....	253
8-5	The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.....	255
8-6	The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.....	260
8-7	An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.....	262
8-8	The XMLLoginConfig DTD	267
8-9	An LDAP server configuration compatible with the testLdap sample configuration.	274
8-10	The JBossSX components of the SRP client-server framework.....	288
8-11	The SRP client-server authentication algorithm sequence diagram.	293
8-12	A sequence diagram illustrating the interaction of the SRPCacheLoginModule with the SRP session cache.....	295
9-1	The complete jboss-web.xml descriptor DTD.....	308
9-2	An overview of the Tomcat-4.0.4 configuration DTD supported by the EmbeddedCatalinaServiceSX Config attribute.	317
9-3	The Internet Explorer 5.5 security alert dialog.....	324
9-4	The Internet Explorer 5.5 SSL certificate details dialog.	325

This introductory overview gives a quick run down of what JBoss is about, and who the JBoss Group is.

Forward

If you are reading this foreword, first of all I want to thank you for buying our products. This is one of the ways in which you can support the development effort and ensure that JBoss continues to thrive and deliver the most technologically advanced web application server possible. The time this book was written corresponds to an interesting point in the evolution of Open Source. There are many projects out there and once the initial excitement has faded, the will to continue requires some professional dedication. JBoss seeks to define the forefront of "Professional Open Source" through commercial activities that subsidize the development of the free core product.

JBoss' modules are growing fast. The JMX base allows us to integrate all these disparate modules together using the MBeanServer of JMX as the basic abstraction for their lifecycle and management.

In this book, we cover the configuration and administration of all our MBeans. We also provide a comprehensive snapshot of the state of JBoss server modules, documented in a professional fashion by one of our very best developers. From the basic architecture, to the advanced modules like JBossSX for security and our CMP engine, you will find the information you need "to get the job done." In addition, we provide a wealth of information on all the modules you will want to understand better and eventually master as you progress in your day-to-day usage of JBoss.

JBoss has achieved a reputation for technical savvy and excellence. I would like this reputation to evolve a bit. Don't get me wrong, I am extremely proud of the group of people gathered around JBoss for the past 2+ years, but I want to make the circle bigger. I want to include all of you reading this book. Think of JBoss, not only as a great application server, but also as a community that thrives by the addition of new minds. We are not simply interested in gaining users; we are interested in giving you the tools and the knowledge necessary to master our product to the point of becoming a contributor. Understanding JBoss' configuration and architecture is a necessary step, not only for your day job using JBoss in development and production, but also an initiation into the joy of technology, as experienced in Open Source.

We hope this book will fulfill its potential to bring as many of you as possible to a strong enough understanding of the modules' functionality to dream up new tools and new functionalities, maybe even new modules. When you reach that point, make sure to come online, where you will find a thriving community of committed professionals sharing a passion for good technology. At www.jboss.org, you can also find additional information, forums, and the latest binaries.

Again thank you for buying our documentation. We hope to see you around. In the meantime, learn, get the job done and, most of all, enjoy,

xxxxxx
Marc Fleury
President
JBoss Group, LLC
xxxxxx

About the Authors

Scott Stark, Ph.D., was born in Washington State of the U.S. in 1964. He started out as a chemical engineer and graduated with a B.S. from the University of Washington, and later a PhD from the University of Delaware. While at Delaware it became apparent that computers and programming were to be his passion and so he made the study of applying massively parallel computers to difficult chemical engineering problems the subject of his PhD research. It has been all about distributed programming ever since. Scott currently serves as the Chief Technology Officer of the JBoss Group, LLC.

Marc Fleury, Ph.D., was born in Paris in 1968. Marc started in Sales at Sun Microsystems France. A graduate of the Ecole Polytechnique, France's top engineering school, and an ex-Lieutenant in the

paratroopers, he has a master in Theoretical Physics from the ENS ULM and a PhD in Physics for work he did as a visiting scientist at MIT (X-Ray Lasers). Marc currently serves as the President of the JBoss Group, LLC; an elite services company based out of Atlanta, GA.

JBoss Group LLC, headed by Marc Fleury, is composed of over 100 developers worldwide who are working to deliver a full range of J2EE tools, making JBoss the premier Enterprise Java application server for the Java 2 Enterprise Edition platform.

JBoss is an Open Source, standards-compliant, J2EE application server implemented in 100% Pure Java. The JBoss/Server and complement of products are delivered under a public license. With upwards of 100,000 downloads per month, JBoss is the most downloaded J2EE based server in the industry.

About Open Source

The basic idea behind open source is very simple: When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing. Open Source is an often-misunderstood term relating to free software. The Open Source Initiative (OSI) web site provides a number of resources that define the various aspects of Open Source including an Open Source Definition at: <http://www.opensource.org/docs/definition.html>. The following quote from the OSI home page summarizes the key aspects as they relate to JBoss nicely:

We in the open source community have learned that this rapid evolutionary process produces better software than the traditional closed model, in which only a very few programmers can see the source and everybody else must blindly use an opaque block of bits.

Open Source Initiative exists to make this case to the commercial world.

Open source software is an idea whose time has finally come. For twenty years it has been building momentum in the technical cultures that built the Internet and the World Wide Web. Now it's breaking out into the commercial world, and that's changing all the rules. Are you ready?

About JBoss

JBoss, one of the leading java Open Source groups, integrates and develops these services for a full J2EE-based implementation. JBoss provides JBossServer, the basic EJB container, and Java Manage-

ment Extension (JMX) infrastructure. It also provides JBossMQ, for JMS messaging, JBossTX, for JTA/JTS transactions, JBossCMP for CMP persistence, JBossSX for JAAS based security, and JBossCX for JCA connectivity. Support for web components, such as servlets and JSP pages, is provided by an abstract integration layer. Implementations of the integration service are provided for third party servlet engines like Tomcat and Jetty. JBoss enables you to mix and match these components through JMX by replacing any component you want with a JMX compliant implementation for the same APIs. JBoss doesn't even impose the JBoss components. Now that is modularity.

JBoss: A Full J2EE Implementation with JMX

Our goal is to provide the full Open Source J2EE stack. We have met our goal, and the reason for our success lies on JMX. JMX, or Java Management Extension, is the best weapon we have found for integration of software. JMX provides a common spine that allows one to integrate modules, containers, and plug-ins. illustrates how JMX is used a bus through which the components of the JBoss architecture interact.

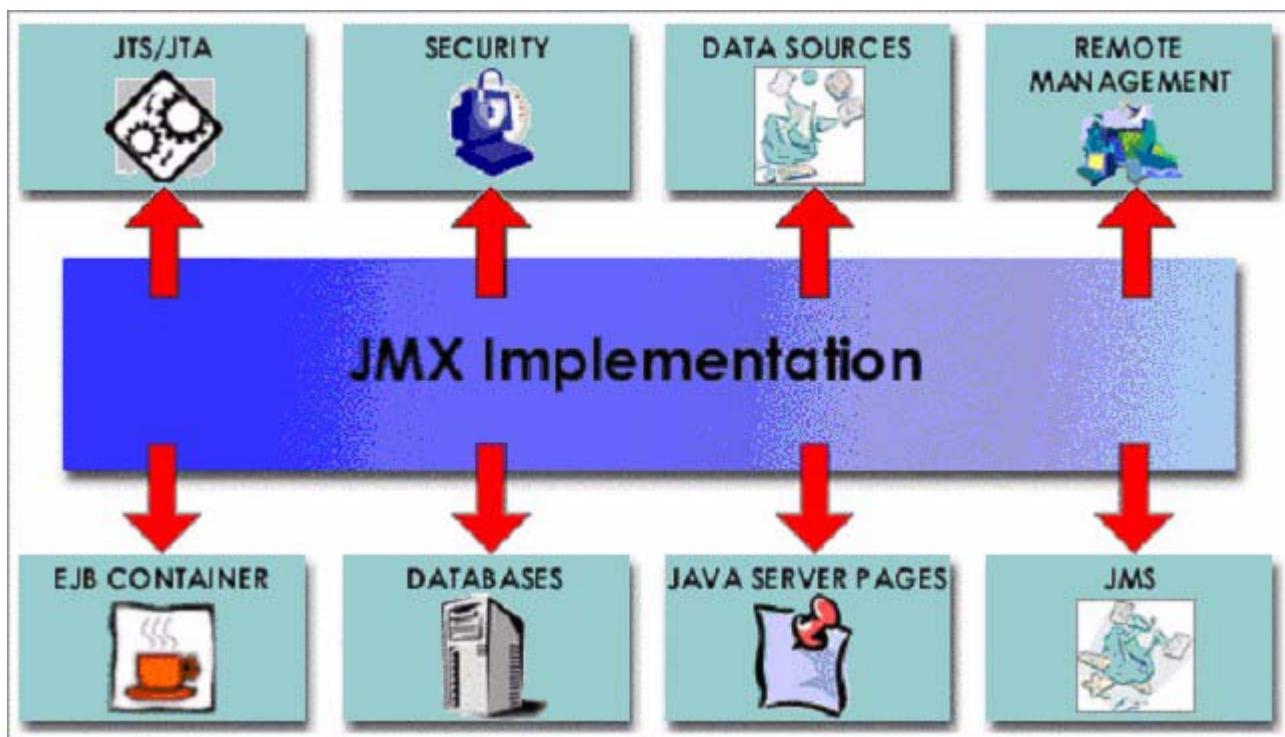


FIGURE i-1. the JBoss JMX integration bus and the standard JBossXX components.

What this Book Covers

The primary focus of this book is the presentation of the standard JBoss 3.0.3 architecture components from both the perspective of their configuration and architecture. As a user of a standard JBoss distribution you will be given an understanding of how to configure the standard components.

As a JBoss developer, you will be given a good understanding of the architecture and integration of the standard components to enable you to extend or replace the standard components for your infrastructure needs. We also show you how to obtain the JBoss source code, along with how to build and debug the JBoss server.

Installing and Building the JBoss Server

JBoss is the highly popular, free J2EE compatible application server that has become the most widely used Open Source application server. The highly flexible and easy to use server architecture has made JBoss the ideal choice for users just starting out with J2EE, as well as senior architects looking for a customizable middleware platform. The server is available as a binary distribution with or without a bundled servlet container. The source code for each binary distribution is also available from the server source repository located at SourceForge. The source code availability allows you to debug the server, learn its inner workings and create customized versions for your personal use.

This chapter presents a step-by-step tutorial on how to install and configure JBoss 3.0.x. You will learn how to obtain updated binaries from the JBoss SourceForge project site, install the binary, and test the installation. You will also learn about the installation directory structure as well as the key configuration files that an administrator may want to use to customize the JBoss installation. You will also learn how to obtain the source code for the 3.0.x release from the SourceForge CVS repository, and how to build the server distribution.

Getting the Binary

The most recent release of JBoss is available from the SourceForge JBoss project files page at <http://sourceforge.net/projects/jboss>. Here you will also find previous releases as well as betas of upcoming versions.

Prerequisites

Before installing and running the server, you should check that your JDK 1.3+ installation is working. The simplest way to do this is to execute the `java -version` command to ensure that the `java` executable is in your path, and that you're using at least version 1.3. For example, running this command on a Linux system with the Sun 1.3.1 JDK produces:

```
/tmp 1206>java -version  
java version "1.3.1_03"  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_03-b03)  
Java HotSpot(TM) Client VM (build 1.3.1_03-b03, mixed mode)
```

It does not matter where you install JBoss. Note, however, that installation of JBoss into a directory that has a name containing spaces causes problems in some situations with Sun based VMs. This is due to bugs with file URLs not correctly escaping the spaces in the resulting URL. There is no requirement for root access to run JBoss on Unix/Linux systems because none of the default ports are below the 0-1023 privileged port range.

Installing the Binary Package

Once you have the binary archive you want to install, use the JDK jar tool, or any other zip extraction tool to unzip the archive contents into a location of your choice. The extraction process will create a `jboss-3.0.2` directory. We'll look at the contents of this directory next.

Directory Structure

Installation of the JBoss distribution creates a `jboss-3.0.2` directory that contains server start scripts, jars, server configuration sets and working directories. You do need to know your way around the distribution layout to locate jars for compilation, updating configurations, deploying your code, etc. Figure 1-1 shows the installation directory of the JBoss server.

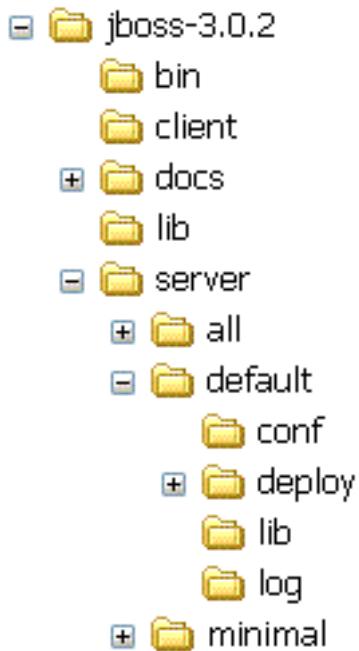


FIGURE 1-1. A view of the JBoss server installation directory structure with the default server configuration file set expanded.

Throughout the book we will refer to the top-level directory as the JBOSS_DIST directory. In Figure 1-1, the default server configuration file set is shown expanded and it contains a number of subdirectories; conf, db, deploy, lib, log and tmp. In a clean installation only the conf, deploy and lib directories will exist. The purposes of the various directories are discussed in Table 1-1 and Table 1-2. In these tables the “ServerConfig Property” column refers to the org.jboss.system.server.ServerConfig constant and its corresponding system property string if the directory has a corresponding value.

TABLE 1-1. The JBoss server installation top-level directories and descriptions

Directory	Description	ServerConfig Property
bin	All the entry point jars and start scripts included with the JBoss distribution are located in this directory.	
client	Jars required for clients are found in the client directory. A typical client requires jboss-client.jar, jboss-common-client.jar, jbosssx-client.jar, jaas.jar, jnp-client.jar, jboss-j2ee.jar, and log4j.jar. If you use JBossMQ JMS provider, you will also need the jbossmq-client.jar and oswego-concurrent.jar.	
server	The JBoss server configuration sets are located under this directory. The default server configuration set is the server/default set. JBoss ships with minimal, default and all configuration sets. The subdirectories and key configuration files contained in the default configuration set will be discussed in more detail in Table 1-2, common subdirectories used by server configuration sets 1-2 and the following section, “The default Server Configuration File Set.”	SERVER_BASE_DIR = “jboss.server.base.dir”
lib	This directory contains startup jars used by JBoss. You do not place your own libraries here.	LIBRARY_URL = “jboss.lib.url”

TABLE 1-2. common subdirectories used by server configuration sets

Directory	Description	ServerConfig Property
conf	A directory for configuration files of services as well as the core jboss-service.xml MBean descriptor.	SERVER_CONFIG_URL = “jboss.server.config.url”
db	This is the directory services use for persistent disk storage.	SERVER_DATA_DIR = “jboss.server.data.dir”
deploy	This is the server’s deployment directory. Drop your jars here and they will be deployed automatically.	
lib	This is the server’s jar library directory. Any jars located in this directory are automatically added to the server class repository on startup.	SERVER_LIBRARY_URL = “jboss.server.lib.url”
log	JBoss log files are located in this directory. File logging is turned on by default and produces boot.log and server.log files in this directory	
tmp	A working directory used by JBoss during deployment of content found in the deploy directory.	SERVER_TEMP_DIR = “jboss.server.temp.dir”

The Default Server Configuration File Set

The JBOSS_DIST/server directory contains one or more configuration file sets. The default JBoss configuration file set is located in the JBOSS_DIST/server/default directory. JBoss allows the possibility of more than one configuration set so that a server can easily be run using alternate configurations. Creating a new configuration file set typically starts with copying the default file set into a new directory name and then modifying the configuration files as desired. The contents of the default configuration file set are shown in Figure 1.2.

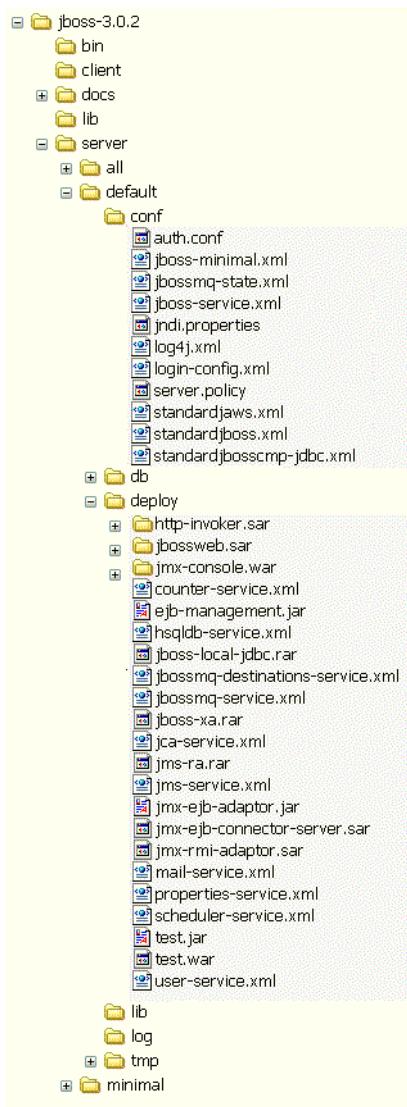


FIGURE 1-2. an expanded view of the default server configuration file set conf and deploy directories.

conf/auth.conf

The auth.conf file is a JAAS login module configuration file as supported by the default javax.security.auth.login.Configuration implementation. This file is deprecated in 3.0 and has been replaced by the login-config.xml file.

conf/jboss-minimal.xml

This is a minimalist example of the jboss-server.xml configuration file. It is the jboss-server.xml file used in the minimal configuration file set.

conf/jboss-server.xml

The jboss-server.xml defines the core services configurations. The complete DTD and syntax of this file is described in along with the details on integrating custom services as JMX MBeans.

conf/jbossmq-state.xml

The jbossmq-state.xml is the JBossMQ configuration file that specifies the user to password mappings file, and the user to durable subscription.

conf/jndi.properties

The jndi.properties file specifies the JNDI InitialContext properties that are used within the JBoss server whenever an InitialContext is created using the no-arg constructor.

conf/log4j.xml

The log4j.xml file configures the Apache log4j framework category priorities and appenders used by the JBoss server code. See the JBoss/Log4j book for details on configuring and using log4j with JBoss.

conf/login-config.xml

The login-config.xml file contains sample server side authentication configurations that are applicable when using JAAS based security. This See Chapter , "," for additional details on the JBoss security framework.

conf/server.policy

The server.policy file is a place holder for Java2 security permissions. The default file simply grants all permissions to all codebases.

conf/standardjaws.xml

The standardjaws.xml provides a default configuration file for the legacy EJB 1.1 JBossCMP engine. The cmp layer has been rewritten in JBoss 3.0 and is fully documented in the JBossCMP book. An introduction to the 3.0 version of the JBossCMP engine is given in XXX.

conf/standardjbosscmp-jdbc.xml

The standardjbosscmp-jdbc.xml provides a default configuration file for the JBoss 3.0 EJB 2.0 JBoss-CMP engine. The cmp layer has been rewritten in JBoss 3.0 and is fully documented in the JBoss-CMP book. An introduction to the 3.0 version of the JBossCMP engine is given in “x”.

conf/standardjboss.xml

The standardjboss.xml file provides the default container configurations. Use of this file is covered in Chapter “The EJB Container Configuration and Architecture”

deploy/counter-service.xml

This is the configuration file for an MBean service offering accumulator style counters to help in diagnosing performance issues.

deploy/ejb-management.jar

This is an EJB deployment that implements the JSR-77 management EJB interfaces.

deploy/hsqldb-service.xml

This is the Hypersonic 1.61 embedded database service configuration file. It sets up the embedded database and related connection factories.

deploy/jboss-local-jdbc.rar

A JCA resource adaptor that implements the JCA ManagedConnectionFactory interface for JDBC drivers that support the DataSource interface but not JCA.

deploy/jboss-xa.rar

A JCA resource adaptor that implements the JCA ManagedConnectionFactory interface for JDBC drivers that support the XADatasource interface but not JCA.

deploy/jbossmq-destinations-service.xml

This file configures a number of JMS queues and topics used by the JMS unit tests.

deploy/jbossmq-service.xml

This file configures the JBossMQ JMS service. This includes the various invocation layers, persistence managers and cache.

[deploy/jbossweb.sar/](#)

The jbossweb.sar directory is an unpackaged MBean service archive for the configuration of the Jetty servlet engine. The SAR is unpackaged rather than deployed as a jar archive so that the jbossweb.sar/META-INF/jboss-service.xml descriptor can be easily edited.

[deploy/jca-service.xml](#)

This contains configuration for the RARDeployer and the three ConnectionManagerFactories supplied with jboss. You should not need to alter this configuration. Configure your ConnectionFactoryLoaders in separate *-service.xml or sar archives. Consult the hsqldb-service.xml for an example.

[deploy/jms-ra.rar](#)

A JCA resource adaptor that implements the JCA ManagedConnectionFactory interface for JMS ConnectionFactories.

[deploy/jms-service.xml](#)

This file configures the JBossMQ JMS provider for use with the jms-ra.rar JCA resource adaptor.

[deploy/jmx-console.war/](#)

The jmx-console.war directory is an unpackaged web application archive that provides an html adaptor for the JMX MBeanServer. The war is unpackaged rather than deployed as a jar archive so that the jmx-console.war/WEB-INF/*.xml descriptors may be edited to configure role based security easily.

[deploy/jmx-ejb-adaptor.jar, deploy/jmx-ejb-connector-server.sar](#)

Collectively this jar and sar deploy an EJB that exposes a subset of the JMX MBeanServer interface methods as an EJB to enable secure remote access to the JMX core functionality.

[deploy/jmx-rmi-adaptor.sar](#)

This is an MBean service archvie that exposes a subset of the JMX MBeanServer interface methods as an an RMI interface to enable remote access to the JMX core functionality.

[deploy/mail-service.xml](#)

This is an MBean service descriptor that provides JavaMail sessions for use inside of the JBoss server.

[deploy/properties-service.xml](#)

This is an MBean service descriptor that allows for customization of the JavaBeans PropertyEditors as well as the definition of system properties.

deploy/scheduler-service.xml

This is an MBean service descriptor that provides a cron type of service.

deploy/user-service.xml

This is a template MBean service descriptor to which you may add your own custom MBean services.

Basic Installation Testing

Once you have installed the JBoss distribution, it is wise to perform a simple startup test to validate that there are no major problems with your Java VM/operating system combination. To test your installation, move to the JBOSS_DIST/bin directory and execute the run.bat or run.sh script as appropriate for your operating system. Your output should be similar to that shown below and contain no error or exception messages:

```
[starksm@banshee bin]$ run.sh
=====
JBoss Bootstrap Environment

JBOSS_HOME: /tmp/jboss-3.0.2
JAVA: /home/starksm/Java/jdk1.3.1_04/bin/java
JAVA_OPTS: -server -Dprogram.name=run.sh
CLASSPATH: /tmp/jboss-3.0.2/bin/run.jar:/home/starksm/Java/jdk1.3.1_04/lib/
tools.jar
=====

19:28:54,690 INFO [Server] JBoss Release: JBoss-3.0.2 CVSTag=JBoss_3_0_2
19:28:54,714 INFO [Server] Home Dir: /home/starksm/JBoss/jboss-3.0.2
19:28:54,715 INFO [Server] Home URL: file:/home/starksm/JBoss/jboss-3.0.2/
19:28:54,716 INFO [Server] Library URL: file:/home/starksm/JBoss/jboss-3.0.2/
lib/
19:28:54,719 INFO [Server] Patch URL: null
19:28:54,719 INFO [Server] Server Name: default
19:28:54,720 INFO [Server] Server Home Dir: /home/starksm/JBoss/jboss-3.0.2/
server/default
19:28:54,721 INFO [Server] Server Home URL: file:/home/starksm/JBoss/jboss-
3.0.2/server/default/
19:28:54,722 INFO [Server] Server Data Dir: /home/starksm/JBoss/jboss-3.0.2/
server/default/db
19:28:54,723 INFO [Server] Server Temp Dir: /home/starksm/JBoss/jboss-3.0.2/
server/default/tmp
19:28:54,724 INFO [Server] Server Config URL: file:/home/starksm/JBoss/jboss-
3.0.2/server/default/conf/
19:28:54,725 INFO [Server] Server Library URL: file:/home/starksm/JBoss/jboss-
3.0.2/server/default/lib/
19:28:54,726 INFO [Server] Root Deployemnt Filename: jboss-service.xml
```

```
19:28:54,732 INFO [Server] Starting General Purpose Architecture (GPA)...
19:28:55,332 INFO [ServerInfo] Java version: 1.3.1_04,Sun Microsystems Inc.
19:28:55,333 INFO [ServerInfo] Java VM: Java HotSpot(TM) Server VM 1.3.1_04-
b02,Sun Microsystems Inc.
19:28:55,333 INFO [ServerInfo] OS-System: Linux 2.4.18-3,i386
19:28:55,413 INFO [ServiceController] Controller MBean online
...
14:09:45,389 INFO [Server] JBoss (MX MicroKernel) [3.0.1RC1 Date:200207080851]
Started in 0m:20s:930ms
```

If your output is similar to this (accounting for installation directory differences), you should now be ready to use JBoss. To shutdown the server, simply issue a Ctrl-C sequence in the console in which JBoss was started.

This starts the server using the default server configuration file set. To start with an alternate configuration set pass in the name of the directory under JBOSS_DIST/server you wish to use as the value to the -c command line option. For example, to start with the minimal configuration file set you would specify:

```
[starksm@banshee bin]$ run.sh -c minimal
...
19:31:20,143 INFO [Server] JBoss (MX MicroKernel) [3.0.2 Date:200208271339]
Started in 0m:4s:172ms
```

To view all of the supported command line options for the JBoss server bootstrap class issue `run -h` command, and the output will be:

```
usage: run.sh [options]

options:
  -h, --help                      Show this help message
  -V, --version                    Show version information
  --
  -D<name>[=<value>]              Stop processing options
  -p, --patchdir=<dir>             Set a system property
  -n, --netboot=<url>              Set the patch directory; Must be absolute
  -c, --configuration=<name>       Boot from net with the given url as base
  -j, --jaxp=<type>                Set the server configuration name
  -L, --library=<filename>         Set the JAXP impl type (ie. crimson)
  -C, --classpath=<url>            Add an extra library to the loaders classpath
  -P, --properties=<url>           Add an extra url to the loaders classpath
                                  Load system properties from the given url
```

Building the Server from Source Code

Source code is available for every JBoss module, and you can build any version of JBoss from source by downloading the appropriate version of the code from SourceForge.

Accessing the JBoss CVS Repositories at SourceForge

The JBoss source is hosted at SourceForge, a great Open Source community service provided by VA Linux Systems. With nearly 43,000 Open Source projects and over 440,000 registered users, SourceForge.net is the largest Open Source hosting service available. Many of the top Open Source projects have moved their development to the SourceForge.net site. The services offered by SourceForge include hosting of project CVS repositories and a web interface for project management that includes bug tracking, release management, mailing lists and more. Best of all, these services are free to all Open Source developers. For additional details and to browse the plethora of projects, see the SourceForge home page: (<http://sourceforge.net/>).

Understanding CVS

CVS (Concurrent Versions System) is an Open Source version control system that is used pervasively throughout the Open Source community. CVS is a Source Control or Revision Control tool designed to keep track of source changes made by groups of developers who are working on the same files. CVS enables developers to stay in sync with each other as each individual chooses.

Anonymous CVS Access

The JBoss project's SourceForge CVS repository can be accessed through anonymous (pserver) CVS with the following instruction set. The module you want to check out must be specified as the modulename. When prompted for a password for anonymous, simply press the Enter key. The general syntax of the command line version of CVS for anonymous access to the JBoss repositories is:

```
cvs -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss login  
cvs -z3 -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss co  
modulename
```

The first command logs into JBoss CVS repository as an anonymous user. This command only needs to be performed once for each machine on which you use CVS because the login information will be saved in your HOME/.cvspass file or equivalent for your system. The second command checks out a copy of the modulename source code into the directory from which you run the cvs command. To avoid having to type the long cvs command line each time, you can set up a CVSROOT environment variable with the value “:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss” and then use the following abbreviated versions of the previous commands:

```
cvs login  
cvs -z3 co modulename
```

Obtaining a CVS Client

The command line version of the CVS program is freely available for nearly every platform, and is included by default on most Linux and Unix distributions. A good port of CVS as well as numerous other Unix programs for Win32 platforms is available from Cygwin at <http://sources.redhat.com/cygwin/>. The syntax of the command line version of CVS will be examined because this is common across all platforms.

For complete documentation on CVS, check out the CVS home page at <http://www.cvshome.org/>.

Building the JBoss-3.0.2 Distribution Using the Source Code

Every JBoss release includes a source archive that contains everything needed to build the release and is available from the files section of the JBoss project site here: <http://sourceforge.net/projects/jboss/>. The source directory structure matches that of the cvs source tree described below so once you have the source distribution you can build the release by following the instructions given in the next section, beginning with the instructions after the step to obtain the jboss-all source tree.

Building the JBoss-3.0.2 Distribution Using the CVS Source Code

This section will guide you through the task of building a JBoss distribution from the CVS source code. To start, create a directory into which you want to download the CVS source tree, and move into the newly created directory. This directory is referred to as the CVS_WD directory for CVS working directory. The example build in this book will check out code into a /tmp/3.0.2 directory on a Linux system. Next, obtain the 3.0.2 version of the source code as shown here:

```
[starksm@main /tmp]$ cd /tmp/3.0.2
[starksm@main 3.0.2]$ cvs co -r JBoss_3_0_2 jboss-all
cvs server: Updating jboss-all
...
cvs server: Updating console/src/resources/org/jboss
cvs server: Updating console/src/resources/org/jboss/console
cvs server: Updating console/src/resources/org/jboss/console/twiddle
```

The resulting jboss-all directory structure contains all of the cvs modules required to build the server. To perform the build, cd to the jboss-all/build directory and execute the build.sh or build.bat file as appropriate for your OS. You will need to set the JAVA_HOME environment variable to the location of the JDK you wish to use for compilation. For this example build, JDK_HOME is set to a Sun JDK 1.3 installation located under /home/starksm/Java/jdk1.3.1_04.

LISTING 1-1. Listing 1-1, the JBoss 3.0.x branch build process

```
[starksm@main 3.0.2]$ cd jboss-all/build/
[starksm@banshee build]$ ./build.sh
Searching for build.xml ...
Buildfile: /tmp/3.0.2/jboss-all/build/build.xml
...
<much_output_deleted>
...
install:

most:

main:

BUILD SUCCESSFUL

Total time: 2 minutes 31 seconds
```

The build process is driven by an Ant based configuration. The main Ant build script is the build.xml file located in the jboss-all/build directory. This script uses a number of custom Ant tasks masked a

buildmagic constructs. The purpose of the main build.xml file is to compile the various module directories under jboss-all and then to integrate their output to produce the binary release. The binary release structure is found under the jboss-all/build/output directory.

Building the JBoss-3.0.3/Tomcat-4.0.5 Integrated Bundle

The default release build performed in the previous section included the Jetty servlet container. A service for embedding the Tomcat-4.0.x servlet container into JBoss is also supported. To build the embedded JBoss/Tomcat bundle follow these steps:

1. Download the Tomcat binary. The Jakarta-Tomcat-4.0.5 binary can be obtained from the Apache site here <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.5/bin/jakarta-tomcat-4.0.5.zip>.
2. Either unpack the bundle into the jboss-all/catalina directory and rename it jakarta-tomcat, or create a jboss-all/catalina/local.properties file that contains a definition for the tomcat.server.root property. This must point to the location of the Tomcat binary. For example: tomcat.server.root=/tmp/jakarta-tomcat-4.0.5-LE-jdk14
3. Perform the main build from the jboss-all/build directory of the JBoss_3_0_3 cvs snapshot or source download if you have not done so already.
4. Perform a build from the jboss-all/catalina directory and specify “bundle” as the target name.

This will create a jboss-all/catalina/output/jboss-3.0.3 directory that includes the embedded tomcat service in place of the default jbossweb.sar. This directory need simply be zipped up into an archive and it is equivalent to the archive available from the SourceForge download page.

An Overview of the JBoss CVS Source Tree

The top-level directory structure under the jboss-all source tree is illustrated in Figure 1-3, the CVS source tree top-level directories. Table 1-3 gives the primary purpose of each of the top-level directories.

TABLE 1-3. Descriptions of the top-level directories of the JBoss CVS source tree.

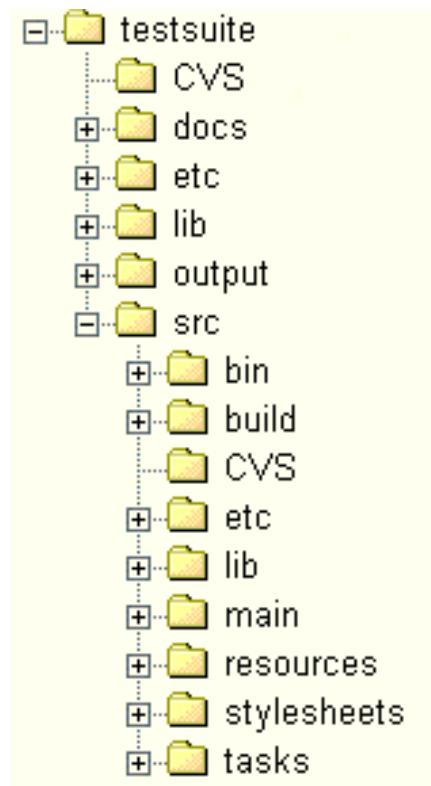
Directory	Description
admin	An experimental administration client not going anywhere currently
build	The main build directory from which the release builds are initiated
catalina	The Tomcat-4.0.x embedded service source module.
cluster	The clustering support services source module.
common	A source module of common utility type code used by many of the other source modules.
connector	The JCA support and application server integration source module.
ejb	Unused
iiop	The RMI/IOP transport service source module.
j2ee	A source module of standard J2EE API interfaces and classes.
jboss.net	A web services support source module that provides support for using SOAP to invoke operations on EJBs and Mbeans.
jetty	The Jetty servlet container source module.
jmx	The JBoss JMX implementation source module.
management	The JBoss JSR-77 source module.
messaging	The JBoss JMS 1.0.2 implementation source module.
naming	The JBoss JNDI 1.2.1 implementation source module.
pool	An obsolete pooling module used by earlier versions of the JCA implementation.
security	The JBoss standard J2EE declarative security implementation based on JAAS.
server	The EJB 2.0 container implementation related source.
system	The JMX microkernel based bootstrap services and standard deployment services source module.
testsuite	The JUnit unit test source module.
thirdparty	A module containing the third-party binary jars used by the JBoss modules.
tools	The jars used by the JBoss build process.
varia	Various utility services that have not or will not been integrated into one of the higher-level modules.

Using the JBoss Test unit testsuite

More advanced testing of the JBoss installation and builds can be done using the JBoss testsuite. The JBossTest suite is a collection of client oriented unit tests of the JBoss server application. It is an Ant based package that uses the JUnit (<http://www.junit.org>) unit test framework. The JBossTest suite is used as a QA benchmark by the development team to help test new functionality and prevent introduction of bugs. It is run on a nightly basis and the results are posted to the development mailing list for all to see.

The unit tests are run using Ant and the source for the tests are contained in the jboss-all/testsuite directory of the source tree. The structure of the jbosstest CVS module is illustrated in Figure 1-3.

FIGURE 1-3. the testsuite CVS module directory structure



The two main source branches are `src/main` and `src/resources`. The `src/main` tree contains the Java source code for the unit tests. The `src/resources` tree contains resource files like deployment descriptors, jar manifests, web content, etc. The root package of every unit test is `org.jboss.test`. The typical structure below each specific unit test subpackage (for example, `security`) consists of a test package that contains the unit test classes. The test subpackage is a required naming convention as this is the only directory searched for unit tests by the Ant build scripts. If the tests involves EJBs then the convention is to include an `interfaces`, and `ejb` subpackage for these components. The unit tests themselves need to follow a naming convention for the class file. The unit test class must be named `XXXUnitTest.java`, where `XXX` is either the class being tested or the name of the functionality being tested.

To run the unit tests use the build scripts located in the `jboss-all/testsuite` directory. The key targets in the `build.xml` file include:

- `tests`: this target builds and runs all unit tests and generates html and text reports of the tests into the `testsuite/output/reports/html` and `testsuite/output/reports/text` directories respectively.
- `tests-standard-unit`: builds all unit tests and runs a subset of the key unit tests. This is useful for quick check of the server to test for gross problems.
- `test`: this target allows one to run all tests within a particular package. To run this target you need to specify a test property that specifies a package name using `-Dtest=package` command line. The package value is the name of the package below `org.jboss.test` you want to run unit tests for. So, for example, to run all unit tests in the `org.jboss.test.naming` package, you would use:
`build.sh -Dtest=naming test`

- one-test: this target allows you to run a single unit test. To run this target you need to specify a test property that specifies the classname of the unit test using -Dtest=classname on the command line. So, for example, to run the org.jboss.test.naming.test.ENCUnitTestCase, you would use:
build.sh -Dtest=org.jboss.test.naming.test.ENCUnitTestCase one-test
- tests-report: this target generates html and text reports of the tests into the testsuite/output/reports/html and testsuite/output/reports/text directories respectively using the current junit xml results in the testsuite/output/reports directory. This is useful for generating the nice html reports when you have run a subset of the tests by hand and want to generate a summary.

On completion of a test the testsuite/output/reports directory will contain one or more XML files that represent the individual junit test runs. The tests-report target collates these into an html report located in the html subdirectory along with a text report located in the text subdirectory. Figure 1-4 shows an example of the html report for a run of the test suite against the JBoss-3.0.3 release on a Mac OS X 10.2 system.

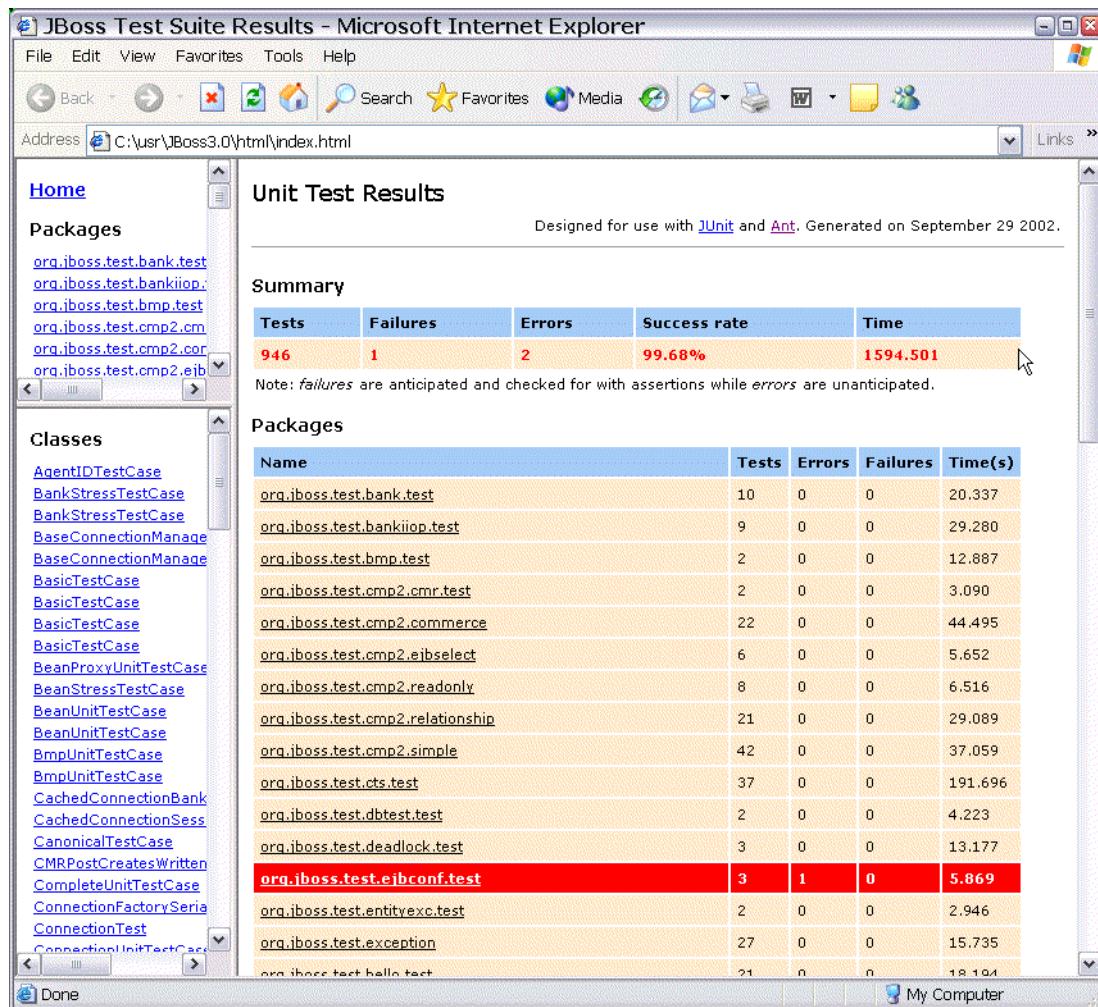


FIGURE 1-4. An example testsuite run report status html view as generated by the testsuite.

Modularly developed from the ground up, the JBoss server and container are completely implemented using component-based plug-ins. The modularization effort is supported by the use of JMX, the Java Management Extension API. Using JMX, industry-standard interfaces help manage both JBoss/Server components and the applications deployed on it. Ease of use is still the number one priority, and the JBoss Server version 3.x architecture sets a new standard for modular, plug-in design as well as ease of server and application management.

This high degree of modularity benefits the application developer in several ways. The already tight code can be further trimmed down to support applications that must have a small footprint. For example, if EJB passivation is unnecessary in your application, simply take the feature out of the server. If you later decide to deploy the same application under an Application Service Provider (ASP) model, simply enable the server's passivation feature for that Web-based deployment. Another example is the freedom you have to drop your favorite object to relational database (O-R) mapping tool, such as TOPLink, directly into the container.

This chapter will introduce you to JMX and its role as the JBoss server component bus. You will also be introduced to the JBoss MBean service notion that adds life cycle operations to the basic JMX management component.

JMX

The success of the full Open Source J2EE stack lies with the use of JMX (Java Management Extension). JMX is the best tool for integration of software. It provides a common spine that allows the user to integrate modules, containers, and plug-ins. Figure 2-1 shows the role of JMX as an integration spine or bus into which components plug. Components are declared as MBean services that are then loaded into JBoss. The components may subsequently be administered using JMX.

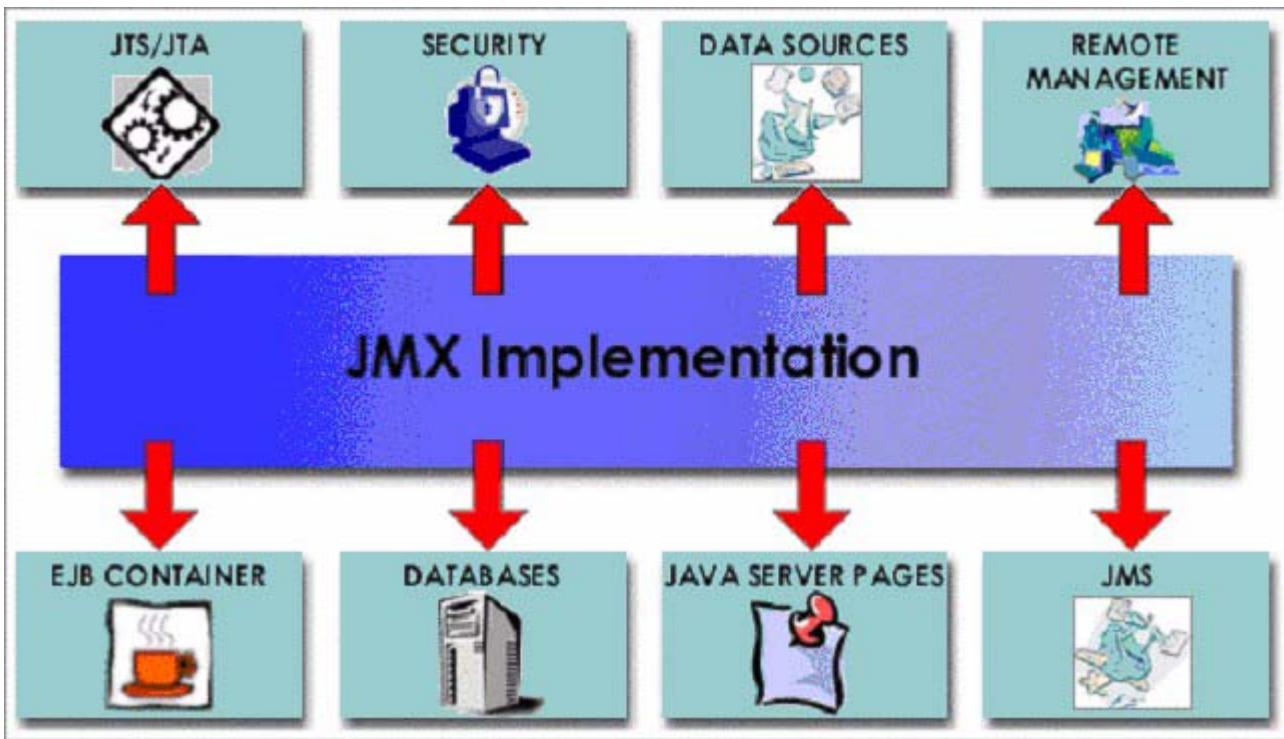


FIGURE 2-1. The JBoss JMX integration bus and the standard JBoss components.

An Introduction to JMX

Before looking at how JBoss uses JMX as its component bus, it would help to get a basic overview what JMX is by touching on some of its key aspects.

JMX components are defined by the Java Management Extensions Instrumentation and Agent Specification, v1.0, which is available from the JSR003 Web page at <http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html>. The material in this JMX overview section is derived from JMX instrumentation specification, with a focus on the aspects most used by JBoss. A more comprehensive discussion of JMX and its application can be found in *JMX: Managing J2EE with Java Management Extensions* written by Juha Lindfors (Sams, 0672322889, 2002).

JMX is about providing a standard for managing and monitoring all varieties of software and hardware components from Java. Further, JMX aims to provide integration with the large number of existing management standards. Figure 2-2, The Relationship between the components of the JMX architecture²⁻² shows examples of components found in a JMX environment, and illustrates the relationship between them as well as how they relate to the three levels of the JMX model. The three levels are:

- Instrumentation, which are the resources to manage
- Agents, which are the controllers of the instrumentation level objects

- Distributed services, the mechanism by which administration applications interact with agents and their managed objects

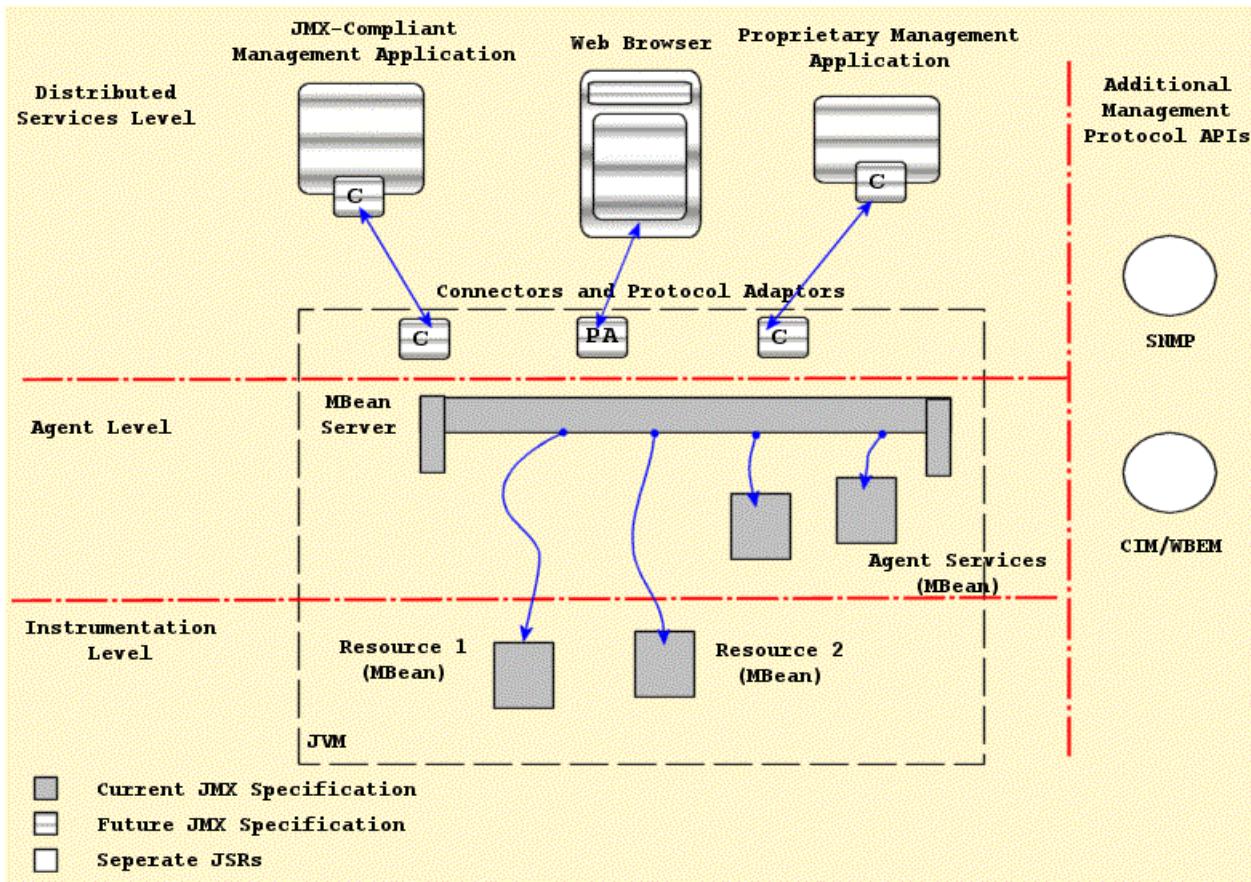


FIGURE 2-2. The Relationship between the components of the JMX architecture

Instrumentation Level

The instrumentation level defines the requirements for implementing JMX manageable resources. A JMX manageable resource can be virtually anything, including applications, service components, devices, and so on. The manageable resource exposes a Java object or wrapper that describes its manageable features, which makes the resource instrumented so that it can be managed by JMX-compliant applications.

The user provides the instrumentation of a given resource using one or more managed beans, or MBeans. There are four varieties of MBean implementations: standard, dynamic, model, and open. The differences between the various MBean types is discussed in “Managed Beans or MBeans” on page 43.

The instrumentation level also specifies a notification mechanism. The purpose of the notification mechanism is to allow MBeans to communicate changes with their environment. This is similar to the JavaBean property change notification mechanism, and can be used for attribute change notifications, state change notifications, and so on.

Agent Level

The agent level defines the requirements for implementing agents. Agents are responsible for controlling and exposing the managed resources that are registered with the agent. By default, management agents are located on the same hosts as their resources. This collocation is not a requirement.

The agent requirements make use of the instrumentation level to define a standard [MBeanServer](#) management agent, supporting services, and a communications connector. JBoss provides an html adaptor via the Sun JMX reference implementation as well as a JBoss implementation of an RMI adaptor.

The JMX agent can be located in the hardware that hosts the JMX manageable resources when a Java Virtual Machine (JVM) is available. This is currently how the JBoss server uses the [MBeanServer](#). A JMX agent does not need to know which resources it will serve. JMX manageable resources may use any JMX agent that offers the services it requires.

Managers interact with an agent's MBeans through a protocol adaptor or connector, as described in the “Distributed Services Level” on page 42 in this chapter. The agent does not need to know anything about the connectors or management applications that interact with the agent and its MBeans.

JMX agents run on the Java 2 Platform Standard Edition. The goal of the JMX specification is to allow agents to run on platforms like PersonalJava and EmbeddedJava, once these are compatible with the Java 2 platform.

Distributed Services Level

The JMX specification notes that a complete definition of the distributed services level is beyond the scope of the initial version of the JMX specification. This was indicated in by the component boxes with the horizontal lines in Figure 2-2. The general purpose of this level is to define the interfaces required for implementing JMX management applications or managers. The following points highlight the intended functionality of the distributed services level as discussed in the current JMX specification.

- Provide an interface for management applications to interact transparently with an agent and its JMX manageable resources through a connector
- Exposes a management view of a JMX agent and its MBeans by mapping their semantic meaning into the constructs of a data-rich protocol (for example HTML or SNMP)
- Distributes management information from high-level management platforms to numerous JMX agents
- Consolidates management information coming from numerous JMX agents into logical views that are relevant to the end user’s business operations
- Provides security

It is intended that the distributed services level components will allow for cooperative management of networks of agents and their resources. These components can be expanded to provide a complete management application.

JMX Component Overview

This section offers an overview of the instrumentation and agent level components. The instrumentation level components include the following:

- MBeans (standard, dynamic, open, and model MBeans)
- Notification model elements
- MBean metadata classes

The agent level components include:

- MBean server
- Agent services

MANAGED BEANS OR MBEANS

An MBean is a Java object that implements one of the standard MBean interfaces and follows the associated design patterns. The MBean for a resource exposes all necessary information and operations that a management application needs to control the resource.

The scope of the management interface of an MBean includes the following:

- Attributes values that may be accessed by name
- Operations or functions that may be invoked
- Notifications or events that may be emitted
- The constructors for the MBean's Java class

JMX defines four types of MBeans to support different instrumentation needs:

- **Standard MBeans** These use a simple JavaBean style naming convention and a statically defined management interface. This is currently the most common type of MBean used by JBoss.
- **Dynamic MBeans** These must implement the [javax.management.DynamicMBean](#) interface, and they expose their management interface at runtime when the component is instantiated for the greatest flexibility. JBoss makes use of Dynamic MBeans in circumstances where the components to be managed are not known until runtime.
- **Open MBeans** These are an extension of dynamic MBeans. Open MBeans rely on basic data types for universal manageability and which are self-describing for user-friendliness. As of the 1.0 JMX specification these are incompletely defined. JBoss currently does not use Open MBeans.
- **Model MBeans** These are also an extension of dynamic MBeans. Model MBeans must implement the [javax.management.modelmbean.ModelMBean](#) interface. Model MBeans simplify the instrumentation of resources by providing default behavior. JBoss currently does not use Model MBeans.

We will present an example of a standard MBean in the section that discusses extending JBoss with your own custom services.

NOTIFICATION MODEL

JMX Notifications are an extension of the Java event model. Both the [MBeanServer](#) and MBeans can send notifications to provide information. The JMX specification defines the [javax.management](#) package [Notification](#) event object, [NotificationBroadcaster](#) event sender, and [NotificationListener](#) event receiver interfaces. The specification also defines the [MBeanServer](#) operations that allow for the registration of notification listeners.

MBEAN METADATA CLASSES

There is a collection of metadata classes that describe the management interface of an MBean. Users can obtain a common metadata view of any of the four MBean types by querying the MBeanServer with which the MBeans are registered. The metadata classes cover an MBean's attributes, operations, notifications, and constructors. For each of these, the metadata includes a name, a description, and its particular characteristics. For example, one characteristic of an attribute is whether it is readable, writable, or both. The metadata for an operation contains the signature of its parameter and return types.

The different types of MBeans extend the metadata classes to be able to provide additional information as required. This common inheritance makes the standard information available regardless of the type of MBean. A management application that knows how to access the extended information of a particular type of MBean is able to do so.

MBEAN SERVER

A key component of the agent level is the managed bean server. Its functionality is exposed through an instance of the [javax.management.MBeanServer](#). An [MBeanServer](#) is a registry for MBeans that makes the MBean management interface available for use by management application. The MBean never directly exposes the MBean object itself; rather, its management interface is exposed through metadata and operations available in the MBeanServer interface. This provides a loose coupling between management applications and the MBeans they manage.

MBeans can be instantiated and registered with the [MBeanServer](#) by the following:

- Another MBean
- The agent itself
- A remote management application (through the distributed services)

When you register an MBean, you must assign it a unique object name. The object name then becomes the unique handle by which management applications identify the object on which to perform management operations. The operations available on MBeans through the MBeanServer include the following:

- Discovering the management interface of MBeans
- Reading and writing attribute values

- Invoking operations defined by MBeans
- Registering for notifications events
- Querying MBeans based on their object name or their attribute values

Protocol adaptors and connectors are required to access the MBeanServer from outside the agent's JVM. Each adaptor provides a view via its protocol of all MBeans registered in the MBeanServer the adaptor connects to. An example adaptor is an HTML adaptor that allows for the display MBeans using a Web browser. As was indicated in Figure 2-2, there are no protocol adaptors defined by the current JMX specification. Later versions of the specification will address the need for remote access protocols in standard ways.

A connector is an interface used by management applications to provide a common API for accessing the MBeanServer in a manner that is independent of the underlying communication protocol. Each connector type provides the same remote interface over a different protocol. This allows a remote management application to connect to an agent transparently through the network, regardless of the protocol. The specification of the remote management interface will be addressed in a future version of the JMX specification.

Adaptors and connectors make all MBean server operations available to a remote management application. For an agent to be manageable from outside of its JVM, it must include at least one protocol adaptor or connector. JBoss currently includes the HTML adaptor from the Sun JMX reference implementation and a custom JBoss RMI adaptor.

AGENT SERVICES

The JMX agent services are objects that support standard operations on the MBeans registered in the MBean server. The inclusion of supporting management services helps you build more powerful management solutions. Agent services are often themselves MBeans, which allow the agent and their functionality to be controlled through the MBean server. The JMX specification defines the following agent services:

- **A Dynamic class loading MLet (management applet) service.** This allows for the retrieval and instantiation of new classes and native libraries from an arbitrary network location.
- **Monitor services.** These observe an MBean attribute's numerical or string value, and can notify other objects of several types of changes in the target.
- **Timer services.** These provide a scheduling mechanism based on a one-time alarm-clock notification or on a repeated, periodic notification.
- **The relation service.** This service defines associations between MBeans and enforces consistency on the relationships.

Any JMX-compliant implementation will provide all of these agent services. JBoss currently does not rely on any of these standard agent services.

JBoss JMX Implementation Architecture

The JBoss ClassLoader Architecture

JBoss 3.x employs a new class loading architecture that facilitates sharing of classes across deployment units. In JBoss 2.x it was difficult to have MBean services interact with dynamically deployed J2EE components, and MBeans themselves were not hot deployable. In JBoss 3.x, everything is hot deployable, and the new deployment architecture and class loading architecture makes this possible.

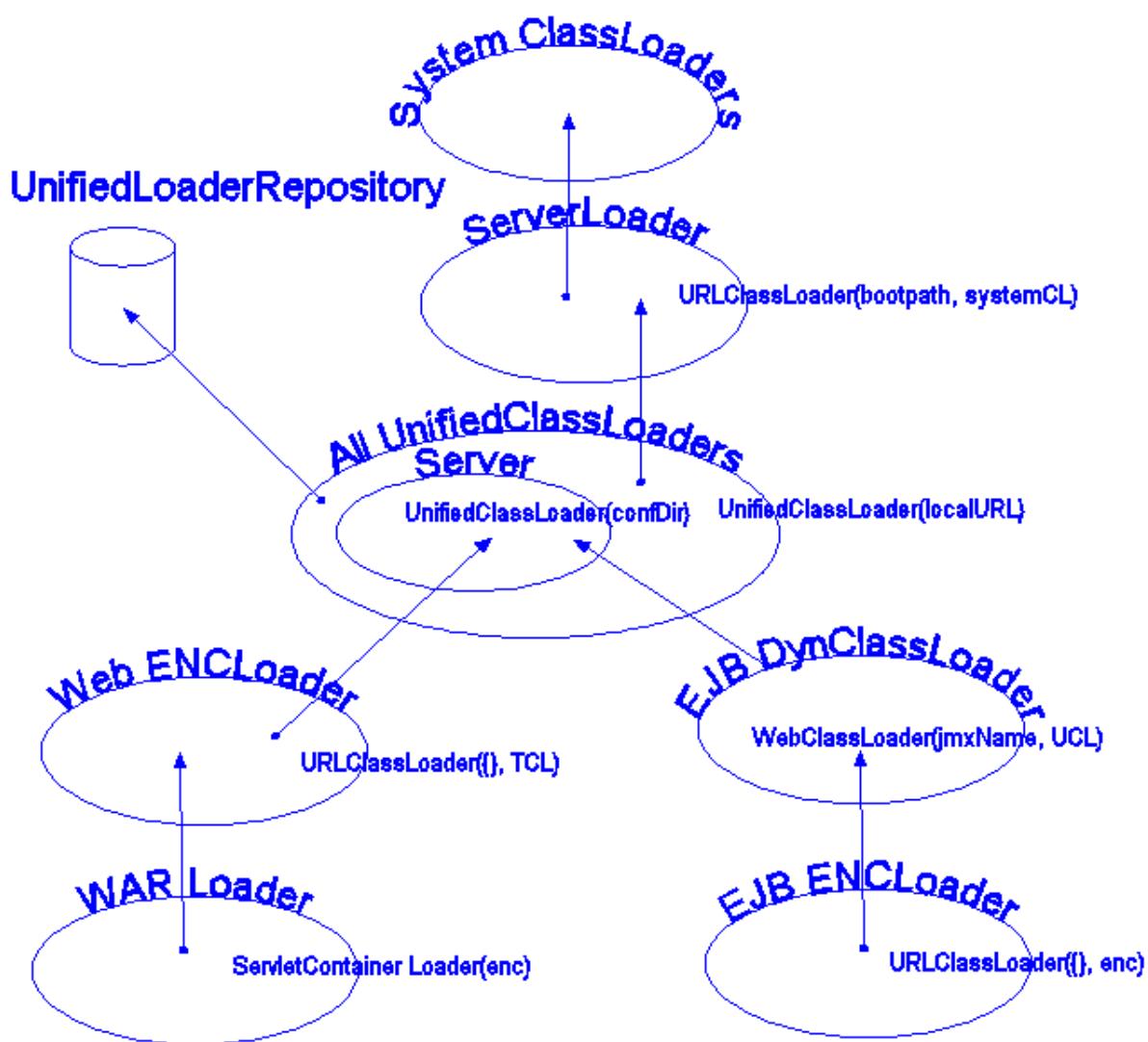


FIGURE 2-3. The JBoss 3.x class loading architecture.

The new class loading architecture centers around a common repository of classes in the form of a JMX MBean. A custom subclass of URLClassLoader known as a UnifiedClassLoader (UCL) is introduced and used as the primary class loader in JBoss 3.0. When a UCL is asked to load a class it first looks to the repository it is associated with to see if the class exist in the repository. Only if the class does not exist in the repository will it be loaded into the repository by the UCL. Currently there is a single class repository shared across all UCL instances. This means the UCLs form a single flat class loader namespace. Figure 2-3 shows an outline of the class hierarchy that would exist for an ear deployment containing EJBs and WARs.

The following points apply to Figure 2-3:

- System ClassLoaders. The System ClassLoaders node refers to either the thread context class loader (TCL) of the VM main thread or of the thread of the app that is loading the JBoss server if it is embedded.
- ServerLoader. The ServerLoader node refers to the a URLClassLoader that delegates to the System ClassLoaders and contains the following boot URLs
- All URLs referenced via the jboss.boot.library.list system property.
- These are path specifications relative to the libraryURL defined by the jboss.lib.url property. If not specified this default to the jboss.home.url + /lib/.
- The JAXP jar which is either crimson.jar or xerces.jar depending on the -j option to the Main entry point. The default is crimson.jar.
- The JBoss JMX jar, jboss-jmx.jar.
- Oswego concurrency jar, concurrent.jar
- Any jars specified as libraries via -L command line options
- Any other jars or directories specified via -C command line options
- Server. The Server node represent a collection of UnifiedClassLoaders created by the org.jboss.system.server.Server interface implementation. The default implementation creates UCLs for the patchDir entries as well as the server conf directory. The last UCL created is set as the JBoss main thread context class loader.
- All UnifiedClassLoaders. The All UnifiedClassLoaders node represents the UCLs created by deployers. This covers EARs, jars, WARs, SARs and directories seen by the deployment scanner as well as jars referenced by their manifests and any nested deployment units they may contain. This is a flat namespace and there cannot be multiple instances of a class in different deployment jars. If there are, only the first loaded will be used. If there are different versions of a class, chaos will most likely ensue. There is a mechanism for scoping visibility based on EAR deployment units that we will discuss in the next section.
- EJB DynClassLoader. The EJB DynClassLoader node is a subclass of URLClassLoader that is used to provide RMI dynamic class loading via the simple HTTP WebService. It specifies an empty URL[] and delegates to the TCL as its parent class loader. If the WebService is configured to allow system level classes to be loaded, all classes in the UnifiedLoaderRepository as well as the system classpath are available via HTTP.
- EJB ENCLoader. The EJB ENCLoader node is a URLClassLoader that exists only to provide a unique context for an EJB deployment's java:comp JNDI context. It specifies an empty URL[] and delegates to the EJB DynClassLoader as its parent class loader.

- Web ENCLoader. The Web ENCLoader node is a URLClassLoader that exists only to provide a unique context for a web deployment's java:comp JNDI context. It specifies an empty URL[] and delegates to the TCL as its parent class loader.
- WAR Loader. The WAR Loader is a servlet container specific ClassLoader that delegates to the Web ENCLoader as its parent class loader. The default behavior is to load from its parent class loader and then the war WEB-INF/{classes,lib} directories. If the servlet 2.3 class loading model is enabled it will first load from the war WEB-INF/{classes,lib} and then the parent class loader.

In its current form there are some advantages and disadvantages to the new class loading architecture. Advantages include:

- Classes do not need to be replicated across deployment units in order to have access to them.
- Many future possibilities including novel partitioning of the repositories into domains, dependency and conflict detection, etc.

Disadvantages include:

- Existing deployments may need to be repackaged to avoid duplicate classes.
- Deployments that depend on different versions of a given class need to be isolated in separate ears and a unique UnifiedLoaderRepository defined using a jboss-app.xml descriptor.

SCOPING CLASSES USING EARs

If you need to deploy multiple versions of an application the default 3.x class loading model would require that each application be deployed in a separate JBoss server. Sometimes this is desirable as you have more control over security and resource monitoring, but it can be difficult to manage multiple server instances. An alternative mechanism exists that allows multiple versions of an application to be deployed using EAR based scoping.

Such an ear creates its own class loader repository that looks first to the deployment units included in the ear before delegating to the default class loader repository. To enable an ear specific loader repository create a META-INF/jboss-app.xml descriptor as shown in Listing 2-1

LISTING 2-1. An example jboss-app.xml descriptor for enabled scoped class loading at the ear level.

```
<jboss-app>
  <loader-repository>some.dot.com:loader=webtest.ear</loader-repository>
</jboss-app>
```

where the value of the loader-repository element is the JMX ObjectName to assign to the repository created for the ear. This must be unique and be a valid JMX ObjectName.

Connecting to the JMX Server

JBoss includes adaptors that allow access to the JMX MBeanServer from outside of the JBoss server VM. The current adaptors include a web application, an RMI interface, and an EJB.

Inspecting the Server - the JMX Console Web Application

JBoss 3.0.1 comes with its own implementation of a JMX HTML adaptor that allows one to view the MBeanServer's MBeans using a standard web browser. The default URL for the console web application is <http://localhost:8080/jmx-console/>. If you browse this location you will see something similar to that presented in Figure 2-4.

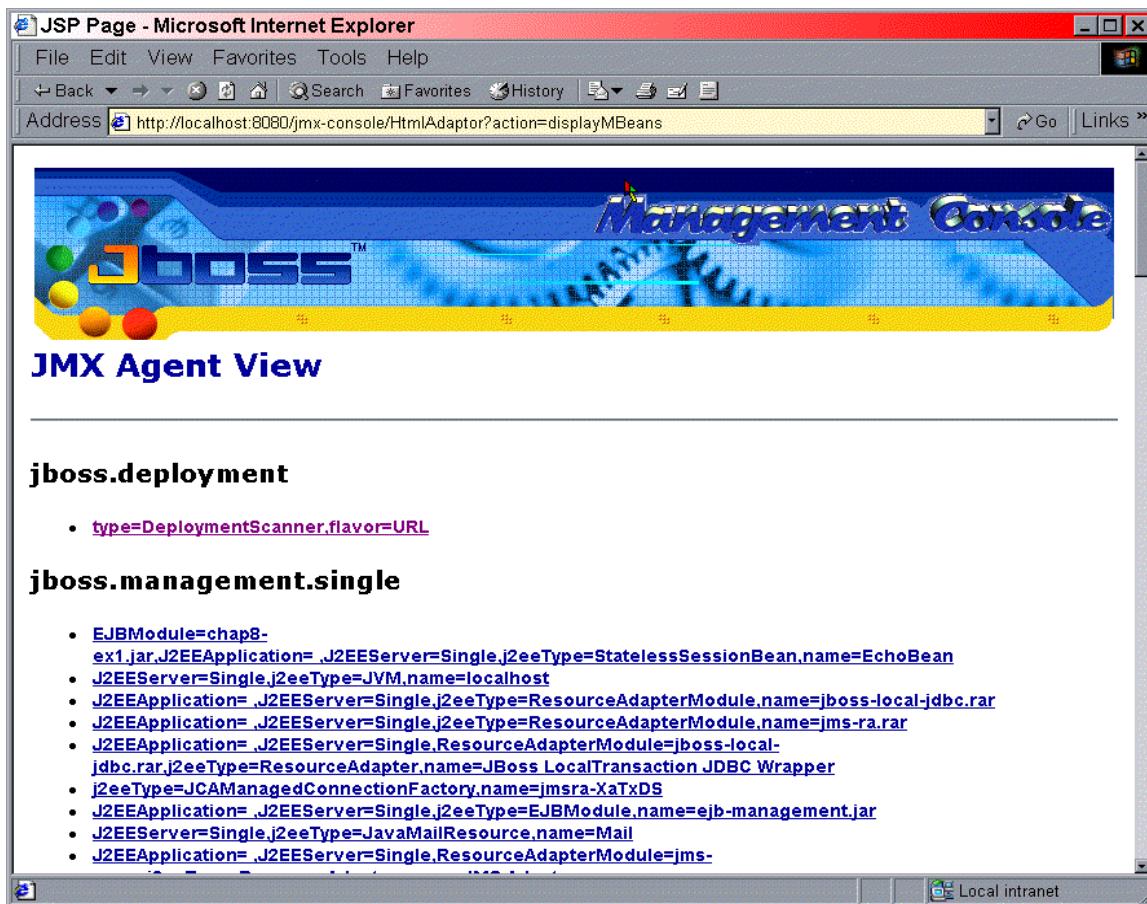


FIGURE 2-4. The JBoss JMX console web application agent view

The top view is called the agent view and it provides a listing of all MBeans registered with the MBeanServer sorted by the domain portion of the MBean ObjectName. Under each domain is the MBeans under that domain. When you select one of the MBeans you will be taken to the MBean view. This allows one to view and edit an MBean's attributes as well as invoke operations. As an example, Figure 2-5 shows the MBean view for the “jboss.system:type=Server” MBean.

The screenshot shows a Microsoft Internet Explorer window titled "MBean Inspector - Microsoft Internet Explorer". The address bar displays the URL: `http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss.system%3Atype%3DSer`. The main content area is titled "MBean View". It displays the following information:

- MBean Name: **Domain Name:** jboss.system
- type:** Server
- MBean Java Class: org.jboss.system.server.ServerImpl

A link "Back to Agent View" is visible below this information.

MBean description:

List of MBean attributes:

Name	Type	Access	Value
BuildNumber	java.lang.String	R	200206291053
VersionName	java.lang.String	R	Branch_3_0
Version	java.lang.String	R	3.0.1RC1(200206291053)
BuildDate	java.lang.String	R	June 29 2002
StartDate	java.util.Date	R	Mon Jul 15 18:16:43 PDT 2002
BuildID	java.lang.String	R	200206291053

List of MBean operations:

FIGURE 2-5. The MBean view for the “jboss.system:type=Server” MBean

The source code for the JMX console web application is located in the `jboss-all/varia` module under the `src/main/org/jboss/jmx` directory. Its web pages are located under `jboss-all/varia/src/resources/jmx`. The application is a simple MVC servlet with JSP views that utilizes the MBeanServer.

SECURING THE JMX CONSOLE

Since the JMX console web application is just a standard servlet, it may be secured using standard J2EE role based security. The jmx-console.war that ships with the 3.0.1 release is deployed as an unpacked war that includes template settings for quickly enabling simple username and password based access restrictions. If you look at the jmx-console.war in the server/default/deploy directory you will find the web.xml and jboss-web.xml descriptors in the WEB-INF directory and a roles.properties and users.properties file under WEB-INF/classes as shown here:

```
bin 875>ls server/default/deploy/jmx-console.war/WEB-INF
classes/  jboss-web.xml  web.xml
bin 876>ls server/default/deploy/jmx-console.war/WEB-INF/classes/
org/  roles.properties  users.properties
```

By uncommenting the security sections of the web.xml and jboss-web.xml descriptors as shown in Listing 2-2, you enable HTTP basic authentication that restricts access to the jmx-console application to username=admin, password=admin. The username and password are determined by the admin=admin line in the WEB-INF/classes/users.properties file.

LISTING 2-2. The jmx-console.war web.xml and jboss-web.xml descriptors with the security elements uncommented.

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
...
<!-- A security constraint that restricts access to the HTML JMX console
to users with the role JBossAdmin. Edit the roles to what you want and
uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
secured access to the HTML JMX console.
-->
<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>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>JBoss JMX Console</realm-name>
</login-config>
```

```
<security-role>
    <role-name>JBossAdmin</role-name>
</security-role>
</web-app>

<jboss-web>
    <!-- Uncomment the security-domain to enable security. You will
        need to edit the htmladaptor login configuration to setup the
        login modules used to authentication users.
    -->
    <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

Make these changes and either with the server running or prior to starting and then when you try to access the jmx-console URL you will see a dialog similar to that shown in Figure 2-6.



FIGURE 2-6. The jmx-console basic HTTP login dialog presented after making the changes shown in Listing 2-2.

For more information on configuring the security settings of JBoss see Chapter 8.

Connecting to JMX Using RMI

JBoss supplies an RMI interface for connecting to the JMX MBeanServer. This interface is org.jboss.jmx.adaptor.rmi.RMIAaptor, and it is shown in Listing 2-3.

LISTING 2-3. The RMIAaptor interface

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */
package org.jboss.jmx.adaptor.rmi;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.ObjectName;
import javax.management.QueryExp;
import javax.management.ObjectInstance;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;
import javax.management.MBeanInfo;

import javax.management.AttributeNotFoundException;
import javax.management.InstanceAlreadyExistsException;
import javax.management.InstanceNotFoundException;
import javax.management.IntrospectionException;
import javax.management.InvalidAttributeValueException;
import javax.management.ListenerNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanRegistrationException;
import javax.management.NotCompliantMBeanException;
import javax.management.OperationsException;
import javax.management.ReflectionException;

public interface RMIAaptor
    extends java.rmi.Remote
{

    public ObjectInstance createMBean(String pClassName, ObjectName pName)
        throws ReflectionException,
               InstanceAlreadyExistsException,
               MBeanRegistrationException,
               MBeanException,
               NotCompliantMBeanException,
               RemoteException;

    public ObjectInstance createMBean(String pClassName, ObjectName pName,
                                     ObjectName pLoaderName)
    throws ReflectionException,
           InstanceAlreadyExistsException,
           MBeanRegistrationException,
           MBeanException,
           NotCompliantMBeanException,
           InstanceNotFoundException,
           RemoteException;

    public ObjectInstance createMBean(String pClassName, ObjectName pName,
                                     Object[] pParams, String[] pSignature)
    throws ReflectionException,
           InstanceAlreadyExistsException,
```

```
        MBeanRegistrationException,
        MBeanException,
        NotCompliantMBeanException,
        RemoteException;

    public ObjectInstance createMBean(String pClassName, ObjectName pName,
        ObjectName pLoaderName, Object[] pParams, String[] pSignature)
throws ReflectionException,
        InstanceAlreadyExistsException,
        MBeanRegistrationException,
        MBeanException,
        NotCompliantMBeanException,
        InstanceNotFoundException,
        RemoteException;

    public void unregisterMBean(ObjectName pName)
        throws InstanceNotFoundException,
        MBeanRegistrationException,
        RemoteException;

    public ObjectInstance getObjectInstance(ObjectName pName)
throws InstanceNotFoundException,
        RemoteException;

    public Set queryMBeans(ObjectName pName, QueryExp pQuery)
throws RemoteException;

    public Set queryNames(ObjectName pName, QueryExp pQuery)
        throws RemoteException;

    public boolean isRegistered(ObjectName pName)
        throws RemoteException;

    public boolean isInstanceOf(ObjectName pName, String pClassName)
        throws InstanceNotFoundException,
        RemoteException;

    public Integer getMBeanCount()
throws RemoteException;

    public Object getAttribute(ObjectName pName, String pAttribute)
throws MBeanException,
        AttributeNotFoundException,
        InstanceNotFoundException,
        ReflectionException,
        RemoteException;

    public AttributeList getAttributes(ObjectName pName, String[] pAttributes)
        throws InstanceNotFoundException,
        ReflectionException,
        RemoteException;

    public void setAttribute(ObjectName pName, Attribute pAttribute)
        throws InstanceNotFoundException,
        AttributeNotFoundException,
```

```

        InvalidAttributeValueException,
        MBeanException,
        ReflectionException,
        RemoteException;

    public AttributeList setAttributes(ObjectName pName, AttributeList
pAttributes)
        throws InstanceNotFoundException,
        ReflectionException,
        RemoteException;

    public Object invoke(ObjectName pName, String pActionName,
        Object[] pParams, String[] pSignature)
        throws InstanceNotFoundException,
        MBeanException,
        ReflectionException,
        RemoteException;

    public String getDefaultDomain()
throws RemoteException;

    public void addNotificationListener(ObjectName pName, ObjectName pListener,
        NotificationFilter pFilter, Object pHandback)
throws InstanceNotFoundException,
        RemoteException;

    public void removeNotificationListener(ObjectName pName, ObjectName
pListener)
        throws InstanceNotFoundException,
        ListenerNotFoundException,
        RemoteException;

    public MBeanInfo getMBeanInfo(ObjectName pName)
throws InstanceNotFoundException,
        IntrospectionException,
        ReflectionException,
        RemoteException;

}

}

```

The RMIAdaptor is bound into JNDI under the name “jmx:” + serverName + “:rmi”, where serverName is the value returned by java.net.InetAddress.getLocalHost. Listing 2-4 shows a client that makes use of the RMIAdaptor interface to query the MBeanInfo for the JNDIView MBean. It also invokes the MBean’s list(boolean) method and displays the result.

LISTING 2-4. A JMX client that uses the RMIAdaptor

```

package org.jboss.chap2.ex4;

import java.net.InetAddress;
import javax.management.MBeanInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;

```

```
import javax.management.ObjectName;
import javax.naming.InitialContext;

import org.jboss.jmx.adaptor.rmi.RMIAaptor;

/** A client that demonstrates how to connect to the JMX server using the RMI
 adaptor.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class JMXBrowser
{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        // The server name to connect to
        String serverName = null;
        if( args.length == 1 )
            serverName = args[0];
        if( serverName == null )
            serverName = InetAddress.getLocalHost().getHostName();
        InitialContext ic = new InitialContext();
        RMIAaptor server = (RMIAaptor) ic.lookup("jmx:" + serverName + ":rmi");

        // Get the MBeanInfo for the JNDIView MBean
        ObjectName name = new ObjectName("jboss:service=JNDIView");
        MBeanInfo info = server.getMBeanInfo(name);
        System.out.println("JNDIView Class: " + info.getClassName());
        MBeanOperationInfo[] opInfo = info.getOperations();
        System.out.println("JNDIView Operations: ");
        for(int o = 0; o < opInfo.length; o++)
        {
            MBeanOperationInfo op = opInfo[o];
            String returnType = op.getReturnType();
            String opName = op.getName();
            System.out.print(" " + returnType + " " + opName + " ( ");
            MBeanParameterInfo[] params = op.getSignature();
            for(int p = 0; p < params.length; p++)
            {
                MBeanParameterInfo paramInfo = params[p];
                String pname = paramInfo.getName();
                String type = paramInfo.getType();
                if( pname.equals(type) )
                    System.out.print(type);
                else
                    System.out.print(type + " " + name);
                if( p < params.length-1 )
                    System.out.print(',');
            }
            System.out.println(")");
        }
    }
}
```

```

        // Invoke the list(boolean) op
        String[] sig = {"boolean"};
        Object[] opArgs = {Boolean.TRUE};
        Object result = server.invoke(name, "list", opArgs, sig);
        System.out.println("JNDIView.list(true) output:\n"+result);
    }
}
}

```

To test the client access using the RMIAdaptor, run the following:

```

examples 812>ant -Dchap=2 -Dex=4 run-example
Buildfile: build.xml
...
run-example4:
[java] JNDIView Class: org.jboss.naming.JNDIView
[java] JNDIView Operations:
[java] + java.lang.String list(boolean)
[java] + void start()
[java] + void stop()
[java] + void destroy()
[java] + void create()
[java] + java.lang.String listXML()
[java] JNDIView.list(true) output:
[java] <h1>Ejb Module: file%{D%}/usr/JBoss3.0/jboss-all/build/output/jboss-3
.0.1RC1/server/default/deploy/chap6-ex2.jar</h1>
[java] <h2>java:comp namespace of the TextMDB bean:</h2>
[java] <pre>
[java] +- env (class: org.jnp.interfaces.NamingContext)
[java] |   +- jms (class: org.jnp.interfaces.NamingContext)
[java] |   |   +- QCF[link -> ConnectionFactory] (class:
javax.naming.LinkRef)
[java] </pre>
[java] <h1>Ejb Module: file%{D%}/usr/JBoss3.0/jboss-all/build/output/jboss-
3.0.1RC1/server/default/deploy/jmx-ejb-adaptor.jar</h1>
[java] <h2>java:comp namespace of the jmx/ejb/Adaptor bean:</h2>
[java] <pre>
[java] +- env (class: org.jnp.interfaces.NamingContext)
[java] |   +- Server-Name (class: java.lang.String)
[java] </pre>
[java] <h1>Ejb Module: file%{D%}/usr/JBoss3.0/jboss-all/build/output/jboss-
3.0.1RC1/server/default/deploy/ejb-management.jar</h1>
[java] <h2>java:comp namespace of the MEJB bean:</h2>
[java] <pre>
[java] +- env (class: org.jnp.interfaces.NamingContext)
[java] |   +- Server-Name (class: java.lang.String)
[java] </pre>
[java] <h1>java: Namespace</h1>
[java] <pre>
[java] +- DefaultDS (class:
org.jboss.resource.adapter.jdbc.local.LocalDataSource)
[java]   +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java]   +- SecurityProxyFactory (class:
org.jboss.security.SubjectSecurityProxyFactory)

```

```
[java] +- DefaultJMSProvider (class: org.jboss.jms.jndi.JBossMQProvider)
[java] +- CounterService (class: org.jboss.varia.counter.CounterService)
[java] +- comp (class: javax.naming.Context)
[java] +- JmsXA (class:
org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
[java] +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] | +- JmsXARealm (class:
org.jboss.security.plugins.SecurityDomainContext) [java] +- jaas (class:
javax.naming.Context)
[java] | +- jbossmq (class:
org.jboss.security.plugins.SecurityDomainContext)
[java] +- timedCacheFactory (class: javax.naming.Context)
[java] Failed to lookup: timedCacheFactory,
errmsg=org.jboss.util.TimedCachePolicy
[java] +- TransactionPropagationContextExporter (class:
org.jboss.tm.TransactionPropagationContextFactory)
[java] +- Mail (class: javax.mail.Session)
[java] +- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)
[java] +- TransactionPropagationContextImporter (class:
org.jboss.tm.TransactionPropagationContextImporter)
[java] +- TransactionManager (class: org.jboss.tm.TxManager)
[java] </pre>
[java] <h1>Global JNDI Namespace</h1>
[java] <pre>
[java] +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- UserTransactionSessionFactory (class:
org.jboss.tm.usertx.server.UserTransactionSessionFactoryImpl)
[java] +- RMIXAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
[java] +- topic (class: org.jnp.interfaces.NamingContext)
[java] | +- testDurableTopic (class: org.jboss.mq.SpyTopic)
[java] | +- testTopic (class: org.jboss.mq.SpyTopic)
[java] | +- securedTopic (class: org.jboss.mq.SpyTopic)
[java] +- queue (class: org.jnp.interfaces.NamingContext)
[java] | +- A (class: org.jboss.mq.SpyQueue)
[java] | +- testQueue (class: org.jboss.mq.SpyQueue)
[java] | +- ex (class: org.jboss.mq.SpyQueue)
[java] | +- DLQ (class: org.jboss.mq.SpyQueue)
[java] | +- D (class: org.jboss.mq.SpyQueue)
[java] | +- C (class: org.jboss.mq.SpyQueue)
[java] | +- B (class: org.jboss.mq.SpyQueue)
[java] +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- RMICConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- UserTransaction (class:
org.jboss.tm.usertx.client.ClientUserTransaction)
[java] +- jmx:succubus-si:rmi (class:
org.jboss.jmx.adaptor.rmi.RMIAaptorImpl)
[java] +- ejb (class: org.jnp.interfaces.NamingContext)
[java] | +- mgmt (class: org.jnp.interfaces.NamingContext)
[java] | | +- MEJB (proxy: $Proxy15 implements interface javax.management.j2ee.ManagementHome,interface javax.ejb.Handle)
[java] | | +- jmx (class: org.jnp.interfaces.NamingContext)
[java] | | | +- ejb (class: org.jnp.interfaces.NamingContext)
[java] | | | | +- Adaptor (proxy: $Proxy19 implements interface org.jboss.jmx.adaptor.interfaces.AdaptorHome,interface javax.ejb.Handle)
```

```
[java] +- invokers (class: org.jnp.interfaces.NamingContext)
[java] |   +- succubus-si (class: org.jnp.interfaces.NamingContext)
[java] |   |   +- jrmp (class: org.jboss.invocation.jrmp.interfaces.JRMPI
[java] |   |   +- jrmp (class:
org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxy)
[java] |   +- UILXAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
[java]   +- UILConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)

[java] </pre>
[java]
```

BUILD SUCCESSFUL

Total time: 5 seconds

JBoss and JMX

When JBoss starts up, one of the first steps performed is to create an MBean server instance ([javax.management.MBeanServer](#)). The JMX MBean server in the Jboss architecture plays the role of a microkernel aggregator component. All other manageable MBean components are plugged into JBoss by registering with the MBean server. The kernel in that sense is only an aggregator, and not a source of actual functionality. The functionality is provided by MBeans, and in fact all major JBoss components are manageable MBeans interconnected through the MBean server.

The Startup Process

In this section we will describe the JBoss server startup process. A summary of the steps that occur during the JBoss server startup sequence is:

1. The run start script initiates the boot sequence using the [org.jboss.Main.main\(String\[\]\)](#) method entry point.
2. The [Main.main](#) method creates a thread group named “jboss” and then starts a thread belonging to this thread group. This thread invokes the [Main.boot](#) method.
3. The [Main.boot](#) method processes the [Main.main](#) arguments and then creates an [org.jboss.system.server.ServerLoader](#) using the system properties along with any additional properties specified as arguments.
4. The xml parser libraries, jboss-jmx.jar, concurrent.jar and extra libraries and classpaths given as arguments are registered with the [ServerLoader](#).
5. The JBoss server instance is created using the [ServerLoader.load\(ClassLoader\)](#) method with the current thread context class loader passed in as the [ClassLoader](#) argument. The returned server instance is an implementation of the [org.jboss.system.server.Server](#) interface. The creation of the server instance entails:
 1. Creating a [java.net.URLClassLoader](#) with the URLs of the jars and directories registered with the [ServerLoader](#). This [URLClassLoader](#) uses the ClassLoader passed in as its parent and it is pushed as the thread context class loader.

2. The class name of the implementation of the Server interface to use is determined by the “jboss.server.type” property. This defaults to org.jboss.system.server.ServerImpl.
3. The Server implementation class is loaded using the URLClassLoader created in step 6 and instantiated using its no-arg constructor. The thread context class loader present on entry into the ServerLoader.load method is restored and the server instance is returned.
6. The server instance is initialized with the properties passed to the ServerLoader constructor using the Server.init(Properties) method.
7. The server instance is then started using the Server.start() method. The default implementation performs the following steps:
 1. Set the thread context class loader to the class loader used to load the ServerImpl class.
 2. Create an MBeanServer under the “jboss” domain using the MBeanServerFactory.createMBeanServer(String) method.
 3. Register the ServerImpl and ServerConfigImpl MBeans with the MBeanServer.
 4. Initialize the unified class loader repository to contain all jars in the optional patch directory as well as the server configuration file conf directory, for example, server/default/conf. For each jar and directory an org.jboss.mx.loading.UnifiedClassLoader is created and registered with the unified repository. One of these UnifiedClassLoader is then set as the thread context class loader. This effectively makes all UnifiedClassLoaders available through the thread context class loader.
 5. The org.jboss.system.ServiceController MBean is created. The ServiceController manages the JBoss MBean services lifecycle. We will discuss the JBoss MBean services notion in detail in “The Core JBoss MBeans” on page 74.
 6. The org.jboss.deployment.MainDeployer is created and started. The MainDeployer manages deployment dependencies and directing deployments to the correct deployer.
 7. The org.jboss.deployment.JARDeployer is created and started. The JARDeployer handles the deployment of jars that are simple library jars.
 8. The org.jboss.deployment.SARDDeployer is created and started. The SARDDeployer handles the deployment of JBoss MBean services.
 9. The MainDeployer is invoked to deploy the services defined in the conf/jboss-service.xml of the current server file set.
 10. Restore the thread context class loader.

The JBoss server starts out as nothing more than a container for the JMX MBean server, and then loads its personality based on the services defined in the jboss-service.xml MBean configuration file from the named configuration set passed to the server on the command line. Because MBeans define the functionality of a JBoss server instance, it is important to understand how the core JBoss MBeans are written, and how you should integrate your existing services into JBoss using MBeans. This is the topic of the next section.

JBoss MBean Services

As we have seen, JBoss relies on JMX to load in the MBean services that make up a given server instances personality. All of the bundled functionality provided with the standard JBoss distribution is based on MBeans. The best way to add services to the JBoss server is to write your own JMX MBeans.

There are two classes of MBeans: those that are independent of JBoss services, and those that are dependent on JBoss services. MBeans that are independent of JBoss services are the trivial case. They can be written per the JMX specification and added to a JBoss server by adding an mbean tag to the deploy/user-service.xml file. Writing an MBean that relies on a JBoss service such as naming requires you to follow the JBoss service pattern. The JBoss MBean service pattern consists of a set of life cycle operations that provide state change notifications. The notifications inform an MBean service when it can create, start, stop, and destroy itself. The management of the MBean service lifecycle is the responsibility of three JBoss MBeans, SARDeployer, ServiceConfigurator and ServiceController.

The SARDeployer MBean

JBoss manages the deployment of its MBean services via a custom MBean that loads an XML variation of the standard MLet configuration file. This custom MBean is implemented in the org.jboss.deployment.SARDeployer class. The SARDeployer MBean is loaded when JBoss starts up by the bootstrap server. The SAR acronym stands for service archive.

The SARDeployer handles services archives. A service archive can be either a jar that ends with a “.sar” suffix and contains a META-INF/jboss-service.xml descriptor, or a standalone XML descriptor with a naming pattern that matches “*-service.xml”. The DTD for the service descriptor is given in Figure 2-7.

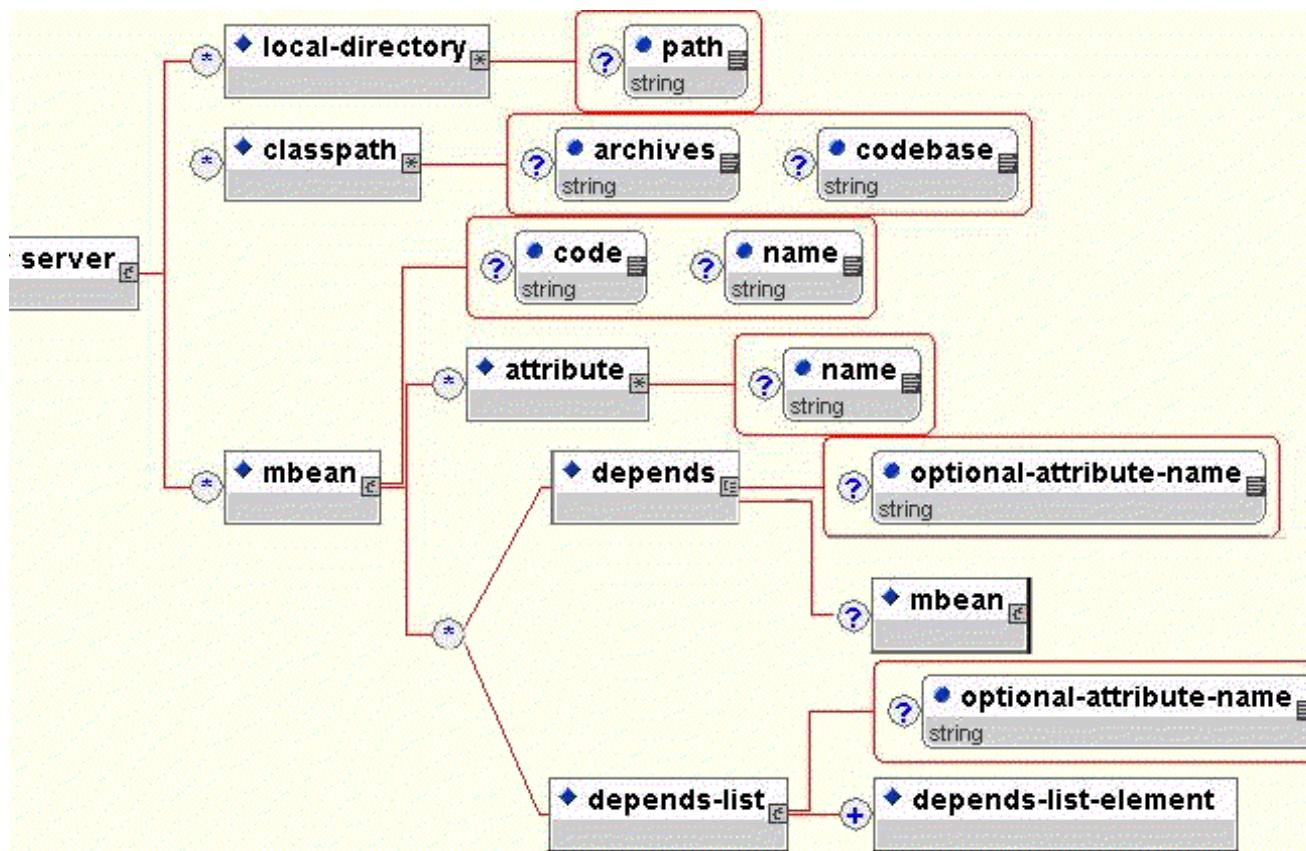


FIGURE 2-7. the DTD for the MBean service descriptor parsed by the SARDeployer.

The elements of the DTD are:

- **server/local-directory**: This element specifies a path within the deployment archive that should be copied to the server/<config>/db directory for use by the MBean. The path attribute is the name of an entry within the deployment archive.
- **server/classpath**: This element specifies one or more external jars that should be deployed with the MBean(s). The optional archives attribute specifies a comma separated list of the jar names to load, or the “*” wild card to signify that all jars should be loaded. The wild card only works with file URLs. The codebase attribute specifies the URL from which the jars specified in the archive attribute should be loaded. If the codebase is a path rather than a URL string, the full URL is built by treating the codebase value as a path relative to the JBoss dist server/<config> directory.
- **server/mbean**: This element specifies an MBean service. The required code attribute gives the fully qualified name of the MBean implementation class. The required name attribute gives the JMX ObjectName of the MBean.
- **server/mbean/attribute**: Each attribute element specifies a name/value pair of the attribute of the MBean. The name of the attribute is given by the name attribute, and the attribute element body gives the value. The body may be a text representation of the value, or an arbitrary element and child elements if the type of the MBean attribute is org.w3c.dom.Element. For text values, the text is converted to the attribute type using the JavaBean [java.beans.PropertyEditor](#) mechanism.
- **server/mbean/depends** and **server/mbean/depends-list**: these elements specify a dependency from the MBean using the element to the MBean(s) named by the depends or depends-list elements. See “Specifying Service Dependencies” on page 67 for the details of specifying dependencies.

When the SARDeployer is asked to deploy a service performs several steps. Figure 2-8 is a sequence diagram that shows the init through start phases of a service.

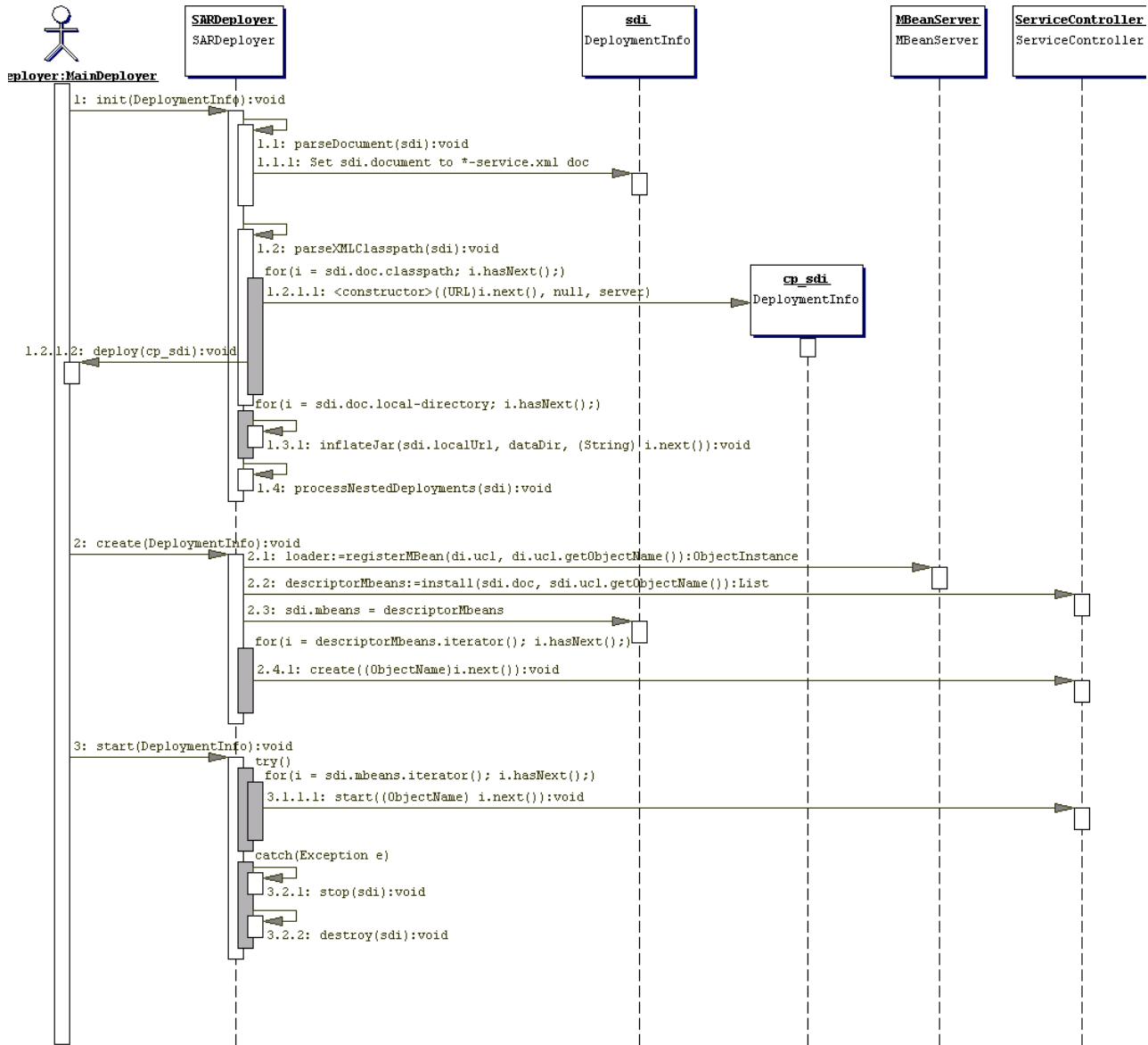


FIGURE 2-8. A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service.

In Figure 2-8 the following is illustrated:

1. Methods prefixed with 1.1 correspond to the load and parse the XML service .descriptor.
2. Methods prefixed with 1.2 correspond to processing each classpath element in the service descriptor to create an independent deployment that makes the jar or directory available through a UnifiedClassLoader registered with the unified loader repository.

3. Methods prefixed with 1.3 correspond to processing each local-directory element in the service descriptor. This does a copy of the SAR elements specified in the path attribute to the server/<config>/db directory.
4. Method 1.4. Process each deployable unit nested in the service a child deployment is created and added to the service deployment info subdeployment list.
5. Method 2.1. The UnifiedClassLoader of the SAR deployment unit is registered with the MBeanServer so that it can be used for loading of the SAR MBeans.
6. Method 2.2. For each mbean element in the descriptor, create an instance and initialize its attributes with the values given in the service descriptor. This is done by calling the [ServiceController.install](#) method.
7. Method 2.4.1. For each MBean instance created, obtain its JMX [ObjectName](#) and ask the [ServiceController](#) to handle the create step of the service lifecycle. The [ServiceController](#) handles the dependencies of the MBean service. Only if the service's dependencies are satisfied is the service create method invoked.
8. Methods prefixed with 3.1 correspond to the start of each MBean service defined in the service descriptor. For each MBean instance created, obtain its JMX [ObjectName](#) and ask the [ServiceController](#) to handle the start step of the service lifecycle. The [ServiceController](#) handles the dependencies of the MBean service. Only if the service's dependencies are satisfied is the service start method invoked.

THE SERVICE LIFE CYCLE INTERFACE

The JMX specification does not define any type of life cycle or dependency management aspect for MBeans. The JBoss [ServiceController](#) MBean introduces this notion. A JBoss MBean is an extension of the JMX MBean in that an MBean is expected to decouple creation from the life cycle of its service duties. This is necessary to implement any type of dependency management. For example, if you are writing an MBean that needs a JNDI naming service to be able to function, your MBean needs to be told when its dependencies are satisfied. This ranges from difficult to impossible to do if the only life cycle event is the MBean constructor. Therefore, JBoss introduces a service life cycle interface that describes the events a service can use to manage its behavior. Listing 2-5 shows the [org.jboss.system.Service](#) interface:

LISTING 2-5. The org.jboss.system.Service interface

```
package org.jboss.system;
public interface Service
{
    public void create() throws Exception;
    public void start() throws Exception;
    public void stop();
    public void destroy();
}
```

The [ServiceController](#) MBean invokes the methods of the [Service](#) interface at the appropriate times of the service lifecycle. We'll discuss the methods in more detail in the [ServiceController](#) section.

Note that there is a J2EE management specification request (JSR 77, <http://jcp.org/jsr/detail/77.jsp>) that introduces a state management notion that includes a start/stop lifecycle notion. When this standard is finalized JBoss will likely support an extension of the JSR 77 based service lifecycle implementation.

THE SERVICECONTROLLER MBEAN

JBoss manages dependencies between MBeans via the `org.jboss.system.ServiceController` custom MBean. The `SARDeployer` delegates to the `ServiceController` when initializing, creating, starting, stopping and destroying MBean services. Figure 2-9 shows a sequence diagram that highlights interaction between the `SARDeployer` and `ServiceController`.

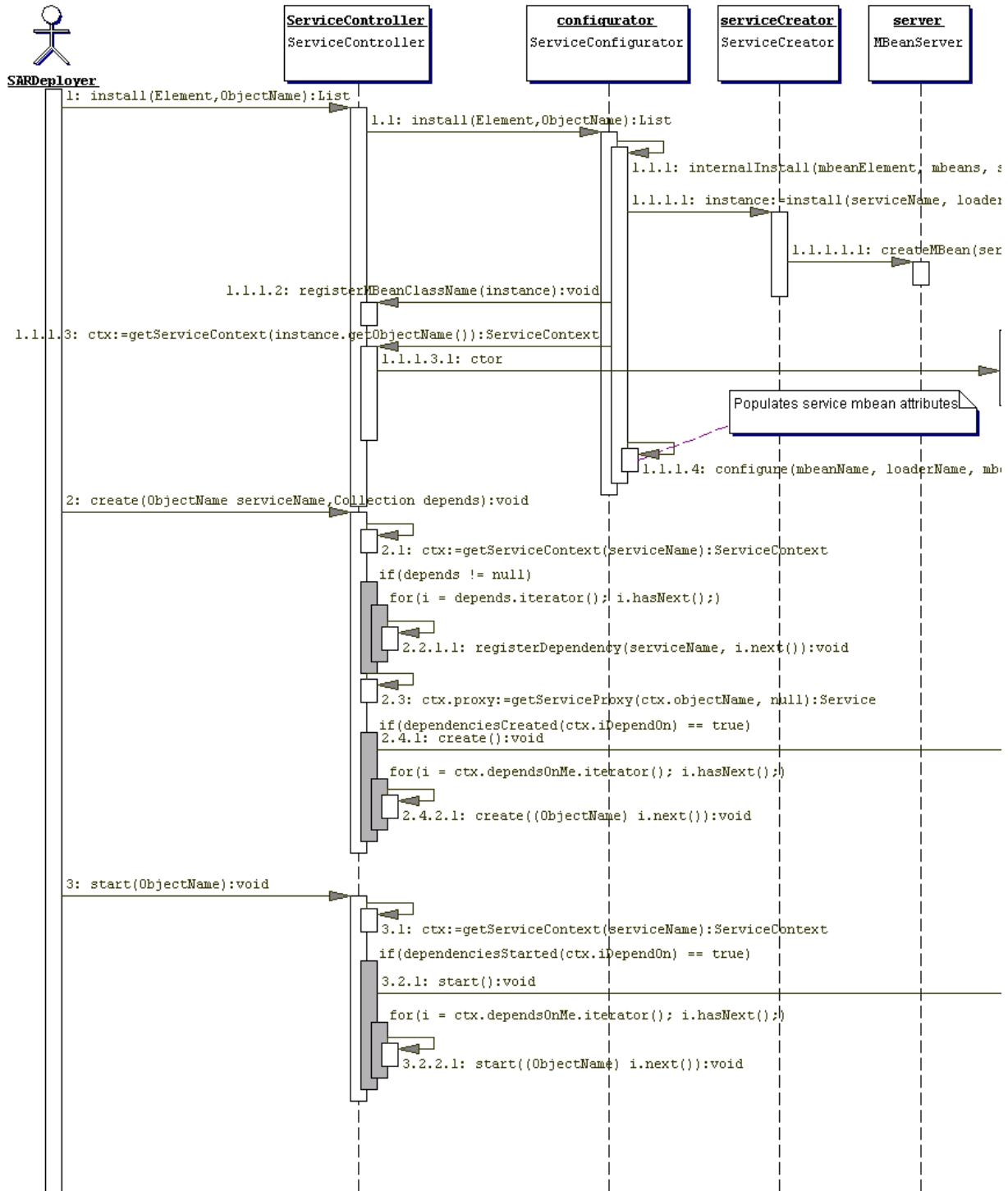


FIGURE 2-9. The interaction between the SARDeployer and ServiceController to start a service.

The ServiceController MBean has four key methods for the management of the service lifecycle: create, start, stop and destroy.

THE CREATE(OBJECTNAME) METHOD

The create(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, a notification of a new Class, or another service reaching its created state.

When a service's create method is called, all services on which the service depends have also had their create method invoked. This gives an MBean an opportunity to check that required MBeans or resources exist. A service cannot utilize other MBean services at this point, as most JBoss MBean services do not become fully functional until they have been started via their start method. Because of this, service implementations often do not implement create in favor of just the start method because that is the first point at which the service can be fully functional.

THE START(OBJECTNAME) METHOD

The start(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, a notification of a new Class, or another service reaching its started state.

When a service's start method is called, all services on which the service depends have also had their start method invoked. Receipt of a start method invocation signals a service to become fully operational since all services upon which the service depends have been created and started.

THE STOP(OBJECTNAME) METHOD

The stop(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, notification of a Class removal, or a service on which other services depend reaching its stopped state.

THE DESTROY(OBJECTNAME) METHOD

The destroy(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, notification of a Class removal, or a service on which other services depend reaching its destroyed state.

Service implementations often do not implement destroy in favor of simply implementing the stop method, or neither stop nor destroy if the service has no state or resources that need cleanup.

SPECIFYING SERVICE DEPENDENCIES

To specify that an MBean service depends on other MBean services you need to declare the dependencies in the mbean element of the service descriptor. This is done using the depends and depends-list elements. One difference between the two elements relates to the optional-attribute-name attribute usage. If you track the ObjectNames of dependencies using single valued attributes you should use the depends element. If you track the ObjectNames of dependencies using java.util.List compatible attributes you would use the depends-list element. If you only want to specify a dependency and don't care to have the associated service ObjectName bound to an attribute of your MBean then use whatever element is easiest. Listing 2-6 shows example service descriptor fragments that illustrate the usage of the dependency related elements.

LISTING 2-6. Service descriptor fragments illustrating the usage of the depends and depends-list elements.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jms.topic:service=Topic,name=testTopic">
  <!-- Declare a dependency on the "jboss.mq:service=DestinationManager" and
      bind this name to the DestinationManager attribute -->
  <depends optional-attribute-name="DestinationManager">
    jboss.mq:service=DestinationManager
  </depends>
  <!-- Declare a dependency on the "jboss.mq:service=SecurityManager" and
      bind this name to the SecurityManager attribute -->
  <depends optional-attribute-name="SecurityManager">
    jboss.mq:service=SecurityManager
  </depends>
  ...
  <!-- Declare a dependency on the "jboss.mq:service=CacheManager" without
      any binding of the name to an attribute-->
  <depends>jboss.mq:service=CacheManager</depends>
</mbean>

<mbean code="org.jboss.mq.server.jmx.TopicMgr"
       name="jboss.mq.destination:service=TopicMgr">
  <!-- Declare a dependency on the given topic destination mbeans and
      bind these names to the Topics attribute -->
  <depends-list optional-attribute-name="Topics">
    <depends-list-element>jms.topic:service=Topic,name=A</depends-list-element>
    <depends-list-element>jms.topic:service=Topic,name=B</depends-list-element>
    <depends-list-element>jms.topic:service=Topic,name=C</depends-list-element>
  </depends>
</mbean>
```

Another difference between the depends and depends-list elements is that the value of the depends element may be a complete MBean service configuration rather than just the ObjectName of the service. Listing 2-7 shows an example from the hsqldb-service.xml descriptor. In this listing the org.jboss.resource.connectionmanager.RARDeployment service configuration is defined using a nested mbean element as the depends element value. This indicates that the org.jboss.resource.connectionmanager.LocalTxConnectionManager MBean depends on this service. The “jboss.jca:service=LocalTxDS,name=hsqldbDS” ObjectName will be bound to the ManagedConnectionFactoryName attribute of the LocalTxConnectionManager class.

LISTING 2-7. An example of using the depends element to specify the complete configuration of a depended on service.

```
<mbean code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
       name="jboss.jca:service=LocalTxCM,name=hsqldbDS">
  <depends optional-attribute-name="ManagedConnectionFactoryName">
    <!--embedded mbean-->
    <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
           name="jboss.jca:service=LocalTxDS,name=hsqldbDS">
      <attribute name="JndiName">DefaultDS</attribute>
```

```

<attribute name="ManagedConnectionFactoryProperties">
  <properties>
    <config-property name="ConnectionURL" type="java.lang.String">
      jdbc:hsqldb:hsqldb://localhost:1476
    </config-property>
    <config-property name="DriverClass" type="java.lang.String">
      org.hsqldb.jdbcDriver
    </config-property>
    <config-property name="UserName" type="java.lang.String">sa
    </config-property>
    <config-property name="Password" type="java.lang.String"/>
  </properties>
</attribute>
...
</mbean>
...
</mbean>

```

IDENTITIFYING UNSATISFIED DEPENDENCIES

The ServiceController MBean supports two operations that help with debugging what MBeans are not running due to unsatisfied dependencies. The first operation is [listIncompletelyDeployed](#). This returns a [java.util.List](#) of [org.jboss.system.ServiceContext](#) objects for the MBean services that are not in the RUNNING state.

The second operation is [listWaitingMBeans](#). This operation returns a [java.util.List](#) of the JMX [ObjectNames](#) of MBean services that cannot be deployed because the class specified by the code attribute is not available.

Writing A JBoss MBean Service

Writing a custom MBean service that integrates into the JBoss server requires the use of the [org.jboss.system.Service](#) interface pattern if the custom service is dependent on other services. When a custom MBean depends on other MBean services you cannot perform any service dependent initialization in any of the [javax.management.MBeanRegistration](#) interface methods since JMX has no dependency notion. Instead, you must manage dependency state using the [Service](#) interface [create](#) and/or [start](#) methods. You can do this using any one of the following approaches:

- Add any of the [Service](#) methods that you want called on your MBean to your MBean interface. This allows your MBean implementation to avoid dependencies on JBoss specific interfaces.
- Have your MBean interface extend the [org.jboss.system.Service](#) interface.
- Have your MBean interface extend the [org.jboss.system.ServiceMBean](#) interface. This is a subinterface of [org.jboss.system.Service](#) that adds [String getName\(\)](#), [int getState\(\)](#), and [String getStateString\(\)](#) methods.

Which approach you choose depends on if you want to be associated with JBoss specific code. If you don't, then you would use the first approach. If you don't care about dependencies on JBoss classes, the simplest approach is to have your MBean interface extend from [org.jboss.system.ServiceMBean](#)

and your MBean implementation class extend from the abstract [org.jboss.system.ServiceMBeanSupport](#) class. This class implements the [org.jboss.system.ServiceMBean](#) interface. [ServiceMBeanSupport](#) provides implementations of the [create](#), [start](#), [stop](#), and [destroy](#) methods that integrate logging and JBoss service state management tracking. Each method delegates any subclass specific work to [createService](#), [startService](#), [stopService](#), and [destroyService](#) methods respectively. When subclassing [ServiceMBeanSupport](#), you would override one or more of the [createService](#), [startService](#), [stopService](#), and [destroyService](#) methods as required.

A CUSTOM MBEAN EXAMPLE

This section develops a simple MBean that binds a HashMap into the JBoss JNDI namespace at a location determined by its [JndiName](#) attribute to demonstrate what is required to create a custom MBean. Because the MBean uses JNDI, it depends on the JBoss naming service MBean and must use the JBoss MBean service pattern to be notified when the naming service is available.

The MBean you develop is called [JNDIMap](#). Version one of the [JNDIMapMBean](#) interface and [JNDIMap](#) implementation class, which is based on the service interface method pattern, is given in Listing 2-8. This version of the interface makes use of the first approach in that it incorporates the [Service](#) interface methods needed to start up correctly, but does not do so by using a JBoss-specific interface. The interface includes the [Service start](#) method, which will be informed when all required services have been started, and the [stop](#) method, which will clean up the service.

LISTING 2-8. JNDIMapMBean interface and implementation based on the service interface method pattern

```
package org.jboss.chap2.ex1;

// The JNDIMap MBean interface
import javax.naming.NamingException;

public interface JNDIMapMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
    public void start() throws Exception;
    public void stop() throws Exception;
}

package org.jboss.chap2.ex1;
// The JNDIMap MBean implementation
import java.util.HashMap;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap implements JNDIMapMBean
{
    private String jndiName;
```

```
private HashMap contextMap = new HashMap();
private boolean started;

public String getJndiName()
{
    return jndiName;
}
public void setJndiName(String jndiName) throws NamingException
{
    String oldName = this.jndiName;
    this.jndiName = jndiName;
    if( started )
    {
        unbind(oldName);
        try
        {
            rebind();
        }
        catch(Exception e)
        {
            NamingException ne = new NamingException("Failed to update jndiName");
            ne.setRootCause(e);
            throw ne;
        }
    }
}
public void start() throws Exception
{
    started = true;
    rebind();
}

public void stop()
{
    started = false;
    unbind(jndiName);
}

private void rebind() throws NamingException
{
    InitialContext rootCtx = new InitialContext();
    Name fullName = rootCtx.getNameParser("").parse(jndiName);
    System.out.println("fullName="+fullName);
    NonSerializableFactory.rebind(fullName, contextMap, true);
}
private void unbind(String jndiName)
{
    try
    {
        InitialContext rootCtx = new InitialContext();
        rootCtx.unbind(jndiName);
        NonSerializableFactory.unbind(jndiName);
    }
    catch(NamingException e)
    {
```

```
        e.printStackTrace();
    }
}
}
```

Version two of the JNDIMapMBean interface and JNDIMap implementation class, which is based on the ServiceMBean interface and ServiceMBeanSupport class, is given in Listing 2-9. In this version, the implementation class extends the ServiceMBeanSupport class and overrides the startService method and the stopService method. JNDIMapMBean also implements the abstract getName to return a descriptive name for the MBean. The JNDIMapMBean interface extends the org.jboss.system.ServiceMBean interface and only declares the setter and getter methods for the JndiName attribute because it inherits the Service life cycle methods from ServiceMBean. This is the third approach mentioned at the start of the "Writing JBoss MBean Services" section. The implementation differences between Listing 2-8 and Listing 2-9 are highlighted in bold in Listing 2-9.

LISTING 2-9. JNDIMap MBean interface and implementation based on the ServiceMBean interface and ServiceMBeanSupport class

```
package org.jboss.chap2.ex2;
// The JNDIMap MBean interface
import javax.naming.NamingException;

public interface JNDIMapMBean extends org.jboss.system.ServiceMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
}

package org.jboss.chap2.ex2;
// The JNDIMap MBean implementation
import java.util.HashMap;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap extends org.jboss.system.ServiceMBeanSupport
    implements JNDIMapMBean
{
    private String jndiName;
    private HashMap contextMap = new HashMap();

    public String getJndiName()
    {
        return jndiName;
    }
    public void setJndiName(String jndiName) throws NamingException
    {
        String oldName = this.jndiName;
        this.jndiName = jndiName;
        if( super.getState() == STARTED )
        {

```

```

        unbind(oldName);
        try
        {
            rebind();
        }
        catch(Exception e)
        {
            NamingException ne = new NamingException("Failed to update jndiName");
            ne.setRootCause(e);
            throw ne;
        }
    }
}

public void startService() throws Exception
{
    rebind();
}
public void stopService()
{
    unbind(jndiName);
}

private void rebind() throws NamingException
{
    InitialContext rootCtx = new InitialContext();
    Name fullName = rootCtx.getNameParser("").parse(jndiName);
log.info("fullName="+fullName);
    NonSerializableFactory.rebind(fullName, contextMap, true);
}
private void unbind(String jndiName)
{
    try
    {
        InitialContext rootCtx = new InitialContext();
        rootCtx.unbind(jndiName);
        NonSerializableFactory.unbind(jndiName);
    }
    catch(NamingException e)
    {
        log.error("Failed to unbind map", e);
    }
}
}

```

The source code for these MBeans along with the service descriptors is located in the examples/src/main/org/jboss/chap2/{ex1,ex2} directories. See “Appendix X” for instructions on installing the examples accompanying the book.

The example 1 service descriptor is given in Listing 2-10 along with a sample client usage code fragment. The JNDIMap MBean binds a HashMap object under the "inmemory/maps/MapTest" JNDI name and the client code fragment demonstrates retrieving the HashMap object from the "inmemory/maps/MapTest" location.

LISTING 2-10. The example 1 JNDIMap MBean service descriptor and a client usage code fragment.

```
<!-- The SAR META-INF/jboss-service.xml descriptor -->
<server>
  <mbean code="org.jboss.chap2.ex1.JNDIMap" name="chap2.ex1:service=JNDIMap">
    <attribute name="JndiName">inmemory/maps/MapTest</attribute>
    <depends>jboss:service=Naming</depends>
  </mbean>
</server>

// Sample lookup code
InitialContext ctx = new InitialContext();
HashMap map = (HashMap) ctx.lookup("inmemory/maps/MapTest");
```

The Core JBoss MBeans

Now that you have seen how to write MBean services, we will overview the JBoss core MBeans defined in the server/default/conf/jboss-service.xml file. Listing 2-11 shows the default jboss-service.xml configuration file shipped with the standard JBoss server distribution. The listing shows only those entries that are not commented out. Here we will only cover the core MBeans that are not covered elsewhere. The other core MBeans will be discussed in their respective chapters.

LISTING 2-11. The default jboss-service.xml configuration file from the standard JBoss distribution

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE server>
<!-- $Id: jboss-service.xml,v 1.44.2.7 2002/06/23 17:57:33 starks Exp $ -->

<!-- ===== -->
<!--
<!--  JBoss Server Configuration
<!--
<!-- ===== -->

<server>

  <!-- Load all library jars located in the server/<name>/lib path -->
  <classpath codebase="lib" archives="*"/>

  <!-- ===== -->
  >
    <!-- Log4j Initialization
    <!-- ===== -->
  >

    <mbean code="org.jboss.logging.Log4jService"
           name="jboss.system:type=Log4jService,service=Logging">
      <attribute name="ConfigurationURL">resource:log4j.xml</attribute>
```

```
</mbean>

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

<mbean code="org.jboss.web.WebService"
       name="jboss:service=Webserver">
  <attribute name="Port">8083</attribute>
  <!-- Should resources and non-EJB classes be downloadable -->
  <attribute name="DownloadServerClasses">true</attribute>
</mbean>

<!-- ===== -->
<!-- JSR-77 Single JBoss Server Management Domain -->
<!-- ===== -->
>

<mbean code="org.jboss.management.j2ee.SingleJBossServerManagement"
       name="jboss.management.single:j2eeType=J2EEDomain,name=Manager" >
</mbean>

<!-- ===== -->
<!-- JNDI -->
<!-- ===== -->
>

<mbean code="org.jboss.naming.NamingService"
       name="jboss:service=Naming">
  <attribute name="Port">1099</attribute>
</mbean>
<mbean code="org.jboss.naming.JNDIView"
       name="jboss:service=JNDIView"/>

<!-- ===== -->
<!-- Security -->
<!-- ===== -->
>

<mbean code="org.jboss.security.plugins.SecurityConfig"
       name="jboss.security:name=SecurityConfig">
  <attribute name="LoginConfig">jboss.security:service=XMLLoginConfig</
attribute>
</mbean>
<mbean code="org.jboss.security.auth.login.XMLLoginConfig"
```

```
        name="jboss.security:service=XMLLoginConfig">
    <attribute name="ConfigResource">login-config.xml</attribute>
</mbean>

<!-- JAAS security manager and realm mapping -->
<mbean code="org.jboss.security.plugins.JaasSecurityManagerService"
       name="jboss.security:service=JaasSecurityManager">
    <attribute name="SecurityManagerClassName">
        org.jboss.security.plugins.JaasSecurityManager
    </attribute>
</mbean>

<!-- ===== -->
>
<!-- Transactions -->
<!-- ===== -->
>

<mbean code="org.jboss.tm.XidFactory"
       name="jboss:service=XidFactory">
</mbean>

<mbean code="org.jboss.tm.TransactionManagerService"
       name="jboss:service=TransactionManager">
    <attribute name="TransactionTimeout">300</attribute>
    <depends optional-attribute-name="XidFactory">jboss:service=XidFactory</
depends>
</mbean>

<mbean code="org.jboss.tm.usertx.server.ClientUserTransactionService"
       name="jboss:service=ClientUserTransaction">
</mbean>

<!-- The CachedConnectionManager is used partly to relay started
     UserTransactions to open connections so they may be enrolled in the new tx
-->
<mbean code="org.jboss.resource.connectionmanager.CachedConnectionManager"
       name="jboss.jca:service=CachedConnectionManager">
</mbean>

<!-- ===== -->
>
<!-- The deployers... -->
<!-- ===== -->
>

<!-- Main Deployer and SARDeployer are provided by main -->

<!-- EJB deployer, remove to disable EJB behavior-->
```

```

<mbean code="org.jboss.ejb.EJBDeployer"
name="jboss.ejb:service=EJBDeployer">
    <attribute name="VerifyDeployments">true</attribute>
    <attribute name="ValidateDTDs">false</attribute>
    <attribute name="MetricsEnabled">false</attribute>
    <attribute name="VerifierVerbose">true</attribute>
    <attribute name="BeanCacheJMSMonitoringEnabled">false</attribute>
</mbean>

<!-- EAR deployer, remove if you are not using Web layers -->
<mbean code="org.jboss.deployment.EARDeployer"
name="jboss.j2ee:service=EARDeployer">
</mbean>

<!-- ===== -->
<!-- Invokers to the JMX node -->
<!-- ===== -->
>

<!-- RMI/JRMP invoker -->
<mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
      name="jboss:service=invoker,type=jrmp">
    <attribute name="RMIClientSocketFactory">custom</attribute>
    <attribute name="RMIObjectPort">4444</attribute>
    <!--
        <attribute name="RMIServerSocketFactory">custom</attribute>
        <attribute name="RMIServerSocketAddr">custom</attribute>
    -->
</mbean>

<mbean code="org.jboss.invocation.local.LocalInvoker"
      name="jboss:service=invoker,type=local">
</mbean>

<!-- ===== -->
<!-- Deployment Scanning -->
<!-- ===== -->
>

<!-- An mbean for hot deployment/undeployment of archives.
-->
<mbean code="org.jboss.deployment.scanner.URLDeploymentScanner"
      name="jboss.deployment:type=DeploymentScanner,flavor=URL">
    <depends optional-attribute-
name="Deployer">jboss.system:service>MainDeployer</depends>
    <attribute name="URLComparator">org.jboss.deployment.DeploymentSorter</
attribute>
    <attribute name="Filter">org.jboss.deployment.scanner.DeploymentFilter</
attribute>

```

```
<attribute name="ScanPeriod">5000</attribute>
<attribute name="URLs">
    ./deploy
</attribute>
</mbean>

</server>
```

ORG.JBOSS.LOGGING.LOG4JSERVICE

The Log4jService MBean configures the Apache log4j system. JBoss uses the log4j framework as its internal logging API.

- **ConfigurationURL:** The URL for the log4j configuration file. This can refer to either a XML document parsed by the org.apache.log4j.xml.DOMConfigurator or a Java properties file parsed by the org.apache.log4j.PropertyConfigurator. The type of the file is determined by the URL content type, or if this is null, the file extension. The default setting of “resource:log4j.xml” refers to the conf/log4j.xml file of the active server configuration file set.
- **RefreshPeriod:** The time in seconds between checks for changes in the log4 configuration specified by the ConfigurationURL attribute. The default value is 60 seconds.

ORG.JBOSS.WEB.WEBSERVICE

The WebService MBean provides dynamic class loading for RMI access to the server EJBs. The configurable attributes for the WebService are as follows:

- **Port:** the WebService listening port number. A port of 0 will use any available port.
- **Host:** Set the name of the public interface to use for the host portion of the RMI codebase URL.
- **BindAddress:** the specific address the WebService listens on. This can be used on a multi-homed host for a java.net.ServerSocket that will only accept connect requests to one of its addresses.
- **Backlog:** The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter. If a connection indication arrives when the queue is full, the connection is refused.
- **DownloadServerClasses:** A flag indicating if the server should attempt to download classes from thread context class loader when a request arrives that does not have a class loader key prefix.

ORG.JBOSS.DEPLOYMENT.SCANNER.URLDEPLOYMENTSCANNER

The URLDeploymentScanner MBean service provides the JBoss hot deployment capability. This service watches one or more URLs for deployable archives and deploys the archives as they appear or change. It also undeploys previously deployed applications if the archive from which the application was deployed is removed. The configurable attributes include:

- **URLs:** A comma separated list of URL strings for the locations that should be watched for changes. Strings that do not correspond to valid URLs are treated as file paths. Relative file paths

are resolved against the server config file set rootm for example, JBOSS_DIST/server/default for the default config file set. If a URL represents a file then the file is deployed and watched for subsequent updates or removal. If a URL represents a directory then the contents of the directory are treated as deployable archives and watched for updates or removal.

The default value for the URLs attribute is “./deploy” which means that any SARs, EARs, JARs, WARs, RARs, etc. dropped into the server/<name>/deploy directory will be automatically deployed and watched for updates.

A directory referred to in the URLs attribute cannot be an unpacked archive. To add an unpacked archive to the watch list you need to refer to its parent directory rather than the root directory of the unpacked archive.

- **ScanPeriod:** The time in milliseconds between runs of the scanner thread. The default is 5000 (5 seconds).
- **URLComparator:** The class name of a [java.util.Comparator](#) implementation used to specify a deployment ordering for deployments found in a scanned directory. The implementation must be able to compare two [java.net.URL](#) objects passed to its compare method. The default setting is the [org.jboss.deployment.DeploymentSorter](#) class which orders based on the deployment URL suffix. The ordering of suffixes is: "sar", "service.xml", "rar", "jar", "war", "wsr", "ear", "zip".

An alternate implementation is the [org.jboss.deployment.scanner.PrefixDeploymentSorter](#) class. This orders the URLs based on numeric prefixes. The prefix digits are converted to an int (ignoring leading zeroes), smaller prefixes are ordered ahead of larger numbers. Deployments that do not start with any digits will be deployed after all numbered deployments. Deployments with the same prefix value are further sorted by the [DeploymentSorter](#) logic.

- **Filter:** The class name of a [java.io.FileFilter](#) implementation that is used to filter the contents of scanned directories. Any file not accepted by this filter will not be deployed. The default is [org.jboss.deployment.scanner.DeploymentFilter](#) which is an implementation that rejects the following patterns:
 - "#*", "%*", ".*", ".*", "_\$*", "*#", "*\$", "*%", "*BAK", "*old", "*orig", "*rej", "*bak", "*v", "*~", ".make.state", ".nse_depinfo", "CVS", "CVS.admin", "RCS", "RCSLOG", "SCCS", "TAGS", "core", "tags"
 - **Deployer:** The JMX ObjectName string of the MBean that implements the [org.jboss.deployer](#) interface operations. The default setting is to use the [MainDeployer](#) created by the bootstrap startup process.

Deployment Ordering and Dependencies

We have seen how to manage dependencies using the service descriptor depends and depends-list tags. The deployment ordering supported by the deployment scanners provides a coarse-grained dependency management in that there is an order to deployments. If dependencies are consistent with the deployment packages then this is a simpler mechanism than having to enumerate the explicit MBean-MBean dependencies. By writing your own filters you can change the coarse grained ordering performed by the deployment scanner.

When a component archive is deployed, its nested deployment units are processed in a depth first ordering. Structuring of components into an archive hierarchy is yet another way to manage deployment ordering.

Typically you will need to explicitly state your MBean dependencies because your packaging structure does not happen to resolve the dependencies. Let's consider an example component deployment that consists of an MBean that uses an EJB. Listing 2-12 shows the example EAR structure.

LISTING 2-12. An example ear with an MBean that depends on an EJB

```
output/chap2/chap2-ex3.ear
+- META-INF/MANIFEST.MF
+- META-INF/jboss-app.xml
+- chap2-ex3.jar (archive) [EJB jar]
| +- META-INF/MANIFEST.MF
| +- META-INF/ejb-jar.xml
| +- org/jboss/chap2/ex3/EchoBean.class
| +- org/jboss/chap2/ex3/EchoLocal.class
| +- org/jboss/chap2/ex3/EchoLocalHome.class
+- chap2-ex3.sar (archive) [MBean sar]
| +- META-INF/MANIFEST.MF
| +- META-INF/jboss-service.xml
| +- org/jboss/chap2/ex3/EjbMBeanAdaptor.class
+- META-INF/application.xml
```

The EAR contains a chap2-ex3.jar and chap2-ex3.sar. The chap2-ex3.jar is the EJB archive and the chap2-ex3.sar is the MBean service archive. We have implemented the service as a DynamicMBean to provide an illustration of their use. Listing 2-13 shows the code for the EjbMBeanAdaptor MBean service.

LISTING 2-13. A DynamicMBean service that uses and EJB

```
1 package org.jboss.chap2.ex3;
2
3 import java.lang.reflect.Method;
4 import javax.ejb.CreateException;
5 import javax.management.Attribute;
6 import javax.management.AttributeList;
7 import javax.management.AttributeNotFoundException;
8 import javax.management.DynamicMBean;
9 import javax.management.InvalidAttributeValueException;
10 import javax.management.JMRuntimeException;
11 import javax.management.MBeanAttributeInfo;
12 import javax.management.MBeanConstructorInfo;
13 import javax.management.MBeanInfo;
14 import javax.management.MBeanNotificationInfo;
15 import javax.management.MBeanOperationInfo;
16 import javax.management.MBeanException;
17 import javax.management.MBeanServer;
18 import javax.management.ObjectName;
19 import javax.management.ReflectionException;
```

```
20 import javax.naming.InitialContext;
21 import javax.naming.NamingException;
22
23 import org.jboss.system.ServiceMBeanSupport;
24
25 /** An example of a DynamicMBean that exposes select attributes and operations
26 of an EJB as an MBean.
27
28 @author Scott.Stark@jboss.org
29 @version $Revision: 1.1 $
30 */
31 public class EjbMBeanAdaptor extends ServiceMBeanSupport
32     implements DynamicMBean
33 {
34     private String helloPrefix;
35     private String ejbJndiName;
36     private EchoLocalHome home;
37
38     /** These are the mbean attributes we expose
39      */
40     private MBeanAttributeInfo[] attributes = {
41         new MBeanAttributeInfo("HelloPrefix", "java.lang.String",
42             "The prefix message to append to the session echo reply",
43             true, // isReadable
44             true, // isWritable
45             false), // isIs
46         new MBeanAttributeInfo("EjbJndiName", "java.lang.String",
47             "The JNDI name of the session bean local home",
48             true, // isReadable
49             true, // isWritable
50             false) // isIs
51     };
52     /** These are the mbean operations we expose
53      */
54     private MBeanOperationInfo[] operations;
55
56
57     /** We override this method to setup our echo operation info. It could
58     also be done in a ctor.
59     */
60     public ObjectName preRegister(MBeanServer server, ObjectName name)
61         throws Exception
62     {
63         log.info("preRegister notification seen");
64
65         operations = new MBeanOperationInfo[5];
66
67         Class thisClass = getClass();
68         Class[] parameterTypes = {String.class};
69         Method echoMethod = thisClass.getMethod("echo", parameterTypes);
70         String desc = "The echo op invokes the session bean echo method and"
71             + " returns its value prefixed with the helloPrefix attribute value";
72         operations[0] = new MBeanOperationInfo(desc, echoMethod);
73
74         // Add the Service interface operations from our super class
```

```
75     parameterTypes = new Class[0];
76     Method createMethod = thisClass.getMethod("create", parameterTypes);
77     operations[1] = new MBeanOperationInfo("The JBoss Service.create",
78     createMethod);
78     Method startMethod = thisClass.getMethod("start", parameterTypes);
79     operations[2] = new MBeanOperationInfo("The JBoss Service.start",
79     startMethod);
80     Method stopMethod = thisClass.getMethod("stop", parameterTypes);
81     operations[3] = new MBeanOperationInfo("The JBoss Service.stop",
81     startMethod);
82     Method destroyMethod = thisClass.getMethod("destroy", parameterTypes);
83     operations[4] = new MBeanOperationInfo("The JBoss Service.destroy",
83     startMethod);
84     return name;
85   }
86
87
88 // --- Begin ServiceMBeanSupport overides
89 protected void createService() throws Exception
90 {
91   log.info("Notified of create state");
92 }
93 protected void startService() throws Exception
94 {
95   log.info("Notified of start state");
96   InitialContext ctx = new InitialContext();
97   home = (EchoLocalHome) ctx.lookup(ejbJndiName);
98 }
99 protected void stopService()
100 {
101   log.info("Notified of stop state");
102 }
103 // --- End ServiceMBeanSupport overides
104
105 public String getHelloPrefix()
106 {
107   return helloPrefix;
108 }
109 public void setHelloPrefix(String helloPrefix)
110 {
111   this.helloPrefix = helloPrefix;
112 }
113
114 public String getEjbJndiName()
115 {
116   return ejbJndiName;
117 }
118 public void setEjbJndiName(String ejbJndiName)
119 {
120   this.ejbJndiName = ejbJndiName;
121 }
122
123 public String echo(String arg)
124   throws CreateException, NamingException
125 {
```

```

126     log.debug("Lookup EchoLocalHome@" + ejbJndiName);
127     EchoLocal bean = home.create();
128     String echo = helloPrefix + bean.echo(arg);
129     return echo;
130 }
131
132 // --- Begin DynamicMBean interface methods
133 /**
134  * Returns the management interface that describes this dynamic resource.
135  * It is the responsibility of the implementation to make sure the
136  * description is accurate.
137  *
138  * @return the management interface descriptor.
139 */
140 public MBeanInfo getMBeanInfo()
141 {
142     String classname = getClass().getName();
143     String description = "This is an MBean that uses a session bean in the"
144         + " implementation of its echo operation.";
145     MBeanConstructorInfo[] constructors = null;
146     MBeanNotificationInfo[] notifications = null;
147     MBeanInfo mbeanInfo = new MBeanInfo(classname, description, attributes,
148         constructors, operations, notifications);
149     // Log when this is called so we know when in the lifecycle this is used
150     Throwable trace = new Throwable("getMBeanInfo trace");
151     log.info("Don't panic, just a stack trace", trace);
152     return mbeanInfo;
153 }
154 /**
155  * Returns the value of the attribute with the name matching the
156  * passed string.
157  *
158  * @param attribute the name of the attribute.
159  * @return the value of the attribute.
160  * @exception AttributeNotFoundException when there is no such attribute.
161  * @exception MBeanException wraps any error thrown by the resource when
162  * getting the attribute.
163  * @exception ReflectionException wraps any error invoking the resource.
164 */
165 public Object getAttribute(String attribute)
166     throws AttributeNotFoundException, MBeanException, ReflectionException
167 {
168     Object value = null;
169     if( attribute.equals("HelloPrefix") )
170         value = getHelloPrefix();
171     else if( attribute.equals("EjbJndiName") )
172         value = getEjbJndiName();
173     else
174         throw new AttributeNotFoundException("Unknown
175 attribute("+attribute+") requested");
176     return value;
177 }
178 /**
179  * Returns the values of the attributes with names matching the
180  * passed string array.
181  *

```

```
180     * @param attributes the names of the attribute.
181     * @return an {@link AttributeList AttributeList} of name and value pairs.
182     */
183    public AttributeList getAttributes(String[] attributes)
184    {
185        AttributeList values = new AttributeList();
186        for(int a = 0; a < attributes.length; a++)
187        {
188            String name = attributes[a];
189            try
190            {
191                Object value = getAttribute(name);
192                Attribute attr = new Attribute(name, value);
193                values.add(attr);
194            }
195            catch(Exception e)
196            {
197                log.error("Failed to find attribute: "+name, e);
198            }
199        }
200        return values;
201    }
202
203    /** Sets the value of an attribute. The attribute and new value are
204     * passed in the name value pair {@link Attribute Attribute}.
205     *
206     * @see javax.management.Attribute
207     *
208     * @param attribute the name and new value of the attribute.
209     * @exception AttributeNotFoundException when there is no such attribute.
210     * @exception InvalidAttributeValueException when the new value cannot be
211     * converted to the type of the attribute.
212     * @exception MBeanException wraps any error thrown by the resource when
213     * setting the new value.
214     * @exception ReflectionException wraps any error invoking the resource.
215     */
216    public void setAttribute(Attribute attribute)
217        throws AttributeNotFoundException, InvalidAttributeValueException,
218        MBeanException, ReflectionException
219    {
220        String name = attribute.getName();
221        if( name.equals("HelloPrefix") )
222        {
223            String value = attribute.getValue().toString();
224            setHelloPrefix(value);
225        }
226        else if( name.equals("EjbJndiName") )
227        {
228            String value = attribute.getValue().toString();
229            setEjbJndiName(value);
230        }
231        else
232            throw new AttributeNotFoundException("Unknown attribute("+name+
requested");
233    }
```

```
234
235     /** Sets the values of the attributes passed as an
236      * {@link AttributeList AttributeList} of name and new value pairs.
237      *
238      * @param attributes the name and new value pairs.
239      * @return an {@link AttributeList AttributeList} of name and value pairs
240      * that were actually set.
241      */
242     public AttributeList setAttributes(AttributeList attributes)
243     {
244         AttributeList setAttributes = new AttributeList();
245         for(int a = 0; a < attributes.size(); a++)
246         {
247             Attribute attr = (Attribute) attributes.get(a);
248             try
249             {
250                 setAttribute(attr);
251                 setAttributes.add(attr);
252             }
253             catch(Exception ignore)
254             {
255             }
256         }
257         return setAttributes;
258     }
259
260     /** Invokes a resource operation.
261      *
262      * @param actionName the name of the operation to perform.
263      * @param params the parameters to pass to the operation.
264      * @param signature the signatures of the parameters.
265      * @return the result of the operation.
266      * @exception MBeanException wraps any error thrown by the resource when
267      * performing the operation.
268      * @exception ReflectionException wraps any error invoking the resource.
269      */
270     public Object invoke(String actionName, Object[] params, String[]
signature)
271         throws MBeanException, ReflectionException
272     {
273         Object rtnValue = null;
274         log.debug("Begin invoke, actionName="+actionName);
275         try
276         {
277             if( actionName.equals("echo") )
278             {
279                 String arg = (String) params[0];
280                 rtnValue = echo(arg);
281                 log.debug("Result: "+rtnValue);
282             }
283             else if( actionName.equals("create") )
284             {
285                 super.create();
286             }
287             else if( actionName.equals("start") )
```

```
288         {
289             super.start();
290         }
291         else if( actionName.equals("stop") )
292         {
293             super.stop();
294         }
295         else if( actionName.equals("destroy") )
296         {
297             super.destroy();
298         }
299         else
300         {
301             throw new JMRuntimeException("Invalid state, don't know about
op="+actionName);
302         }
303     }
304     catch(Exception e)
305     {
306         throw new ReflectionException(e, "echo failed");
307     }
308     log.debug("End invoke, actionName="+actionName);
309     return rtnValue;
310 }
311
312 // --- End DynamicMBean interface methods
313
314 }
```

Believe it or not, this is a very trial MBean. The vast majority of the code is there to provide the MBean metadata and handle the callbacks from the [MBeanServer](#). This is required because a [DynamicMBean](#) is free to expose whatever management interface it wants. A [DynamicMBean](#) can in fact change its management interface at runtime simply by returning a different metadata value from the [getMBeanInfo](#) method. Of course, clients may not be too happy with such a dynamic object, but the [MBeanServer](#) will do nothing to prevent a [DynamicMBean](#) from changing its interface.

There are two points to this example. First, demonstrate how an MBean can depend on an EJB for some of its functionality and second, how to create MBeans with dynamic management interfaces. If we were to write a standard MBean with a static interface for this example it would be as given in Listing 2-14.

LISTING 2-14. The standard MBean interface for Listing 2-13

```
public interface EjbMBeanAdaptorMBean
{
    public String getHelloPrefix();
    public void setHelloPrefix(String prefix);
    public String getEjbJndiName();
    public void setEjbJndiName(String jndiName);
    public String echo(String arg) throws CreateException, NamingException;
    public void create() throws Exception;
    public void start() throws Exception;
```

```
    public void stop();
    public void destroy();
}
```

Moving to lines 67-83, this is where the MBean operation metadata is constructed. The echo(String), create(), start(), stop() and destroy() operations are defined by obtaining the corresponding java.lang.reflect.Method object and adding a description. Let's go through the code and discuss where this interface implementation exists and how the MBean uses the EJB. Beginning with lines 40-51, the two MBeanAttributeInfo instances created define the attributes of the MBean. These attributes correspond to the getHelloPrefix/setHelloPrefix and getEjbJndiName/setEjbJndiName of the static interface. One thing to note in terms of why one might want to use a DynamicMBean is that you have the ability to associate descriptive text with the attribute metadata. This is not something you can do with a static interface.

Lines 88-103 correspond to the JBoss service life cycle callbacks. Since we are subclassing the ServiceMBeanSupport utility class, we override the createService, startService, and stopService template callbacks rather than the create, start, and stop methods of the service interface. Note that we cannot attempt to lookup the EchoLocalHome interface of the EJB we make use of until the startService method. Any attempt to access the home interface in an earlier life cycle method would result in the name not being found in JNDI because the EJB container had not gotten to the point of binding the home interfaces. Because of this dependency we will need to specify that the MBean service depends on the EchoLocal EJB container to ensure that the service is not started before the EJB container is started. We will see this dependency specification when we look at the service descriptor.

Lines 105-121 are the HelloPrefix and EjbJndiName attribute accessors implementations. These are invoked in response to getAttribute/setAttribute invocations made through the MBeanServer.

Lines 123-130 correspond to the echo(String) operation implementation. This method invokes the EchoLocal.echo(String) EJB method. The local bean interface is created using the EchoLocalHome that was obtained in the startService method.

The remainder of the class makes up the DynamicMBean interface implementation. Lines 133-152 correspond to the MBean metadata accessor callback. This method returns a description of the MBean management interface in the form of the javax.management.MBeanInfo object. This is made up of a description, the MBeanAttributeInfo and MBeanOperationInfo metadata created earlier, as well as constructor and notification information. This MBean does not need any special constructors or notifications so this information is null.

Lines 154-258 handle the attribute access requests. This is rather tedious and error prone code so a toolkit or infrastructure that helps generate these methods should be used. A ModelMBean framework based on XML called XBeans is currently being investigated in JBoss. Other than this, no other DynamicMBean frameworks currently exist.

Lines 260-310 correspond to the operation invocation dispatch entry point. Here the request operation action name is checked against those the MBean handles and the appropriate method is invoked.

The jboss-service.xml descriptor for the MBean is given in Listing 2-15. The dependency on the EJB container MBean is highlighted in bold. The format of the EJB container MBean ObjectName is:

```
" jboss.j2ee:service=EJB,jndiName=" + <home-jndi-name>
```

where the <home-jndi-name> is the EJB home interface JNDI name.

LISTING 2-15. The DynamicMBean jboss-service.xml descriptor

```
<server>
  <mbean code="org.jboss.chap2.ex3.EjbMBeanAdaptor"
         name="jboss.book:service=EjbMBeanAdaptor">
    <attribute name="HelloPrefix">AdaptorPrefix</attribute>
    <attribute name="EjbJndiName">local/chap2.EchoBean</attribute>
    <depends>jboss.j2ee:service=EJB,jndiName=chap2.EchoBean</depends>
  </mbean>
</server>
```

Deploy the example ear by running:

```
examples 880>ant -Dchap=2 -Dex=3 run-example
Buildfile: build.xml
...
run-example3:
    [copy] Copying 1 file to D:\usr\JBoss3.0\jboss-all\build\output\jboss-3.0.1
RC1\server\default\deploy
BUILD SUCCESSFUL

Total time: 5 seconds
examples 881>
```

On the server console there will be messages similar to the following:

```
15:11:23,468 INFO [MainDeployer] Starting deployment of package: file:/D:/usr/JBoss3.0/
jboss-all/build/output/jboss-3.0.1RC1/server/default/deploy/chap2-ex3.ear
15:11:23,484 INFO [EARDeployer] Init J2EE application: file:/D:/usr/JBoss3.0/jboss-all/
build/output/jboss-3.0.1RC1/server/default/deploy/chap2-ex3.ear
15:11:23,687 INFO [EjbMBeanAdaptor] preRegister notification seen
15:11:23,687 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:149)
    at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:513)
    at org.jboss.system.ServiceConfigurator.configure(ServiceConfigurator.java:230)
    at
org.jboss.system.ServiceConfigurator.internalInstall(ServiceConfigurator.java:187)
    at org.jboss.system.ServiceConfigurator.install(ServiceConfigurator.java:130)
    at org.jboss.system.ServiceController.install(ServiceController.java:224)
    at java.lang.reflect.Method.invoke(Native Method)
    at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
    at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:491)
    at org.jboss.util.jmx.MBeanProxy.invoke(MBeanProxy.java:174)
    at $Proxy3.install(Unknown Source)
    at org.jboss.deployment.SARDeployer.create(SARDeployer.java:209)
    at org.jboss.deployment.MainDeployer.create(MainDeployer.java:749)
    at org.jboss.deployment.MainDeployer.create(MainDeployer.java:741)
    at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:615)
    at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:580)
```

```
        at java.lang.reflect.Method.invoke(Native Method)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:491)
        at org.jboss.util.jmx.MBeanProxy.invoke(MBeanProxy.java:174)
        at $Proxy4.deploy(Unknown Source)
        at
org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:427)
        at
org.jboss.deployment.scanner.URLDeploymentScanner.scanDirectory(URLDeploymentScanner.java:648)
        at
org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:499)
        at
org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(AbstractDeploymentScanner.java:202)
        at
org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(AbstractDeploymentScanner.java:191)
15:11:23,750 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
        at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:149)
        at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:513)
        at org.jboss.system.ServiceController.getServiceProxy(ServiceController.java:736)
        at org.jboss.system.ServiceController.create(ServiceController.java:276)
        at org.jboss.system.ServiceController.create(ServiceController.java:242)
        at java.lang.reflect.Method.invoke(Native Method)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:491)
        at org.jboss.util.jmx.MBeanProxy.invoke(MBeanProxy.java:174)
        at $Proxy3.create(Unknown Source)
        at org.jboss.deployment.SARDeployer.create(SARDeployer.java:217)
        at org.jboss.deployment.MainDeployer.create(MainDeployer.java:749)
        at org.jboss.deployment.MainDeployer.create(MainDeployer.java:741)
        at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:615)
        at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:580)
        at java.lang.reflect.Method.invoke(Native Method)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:491)
        at org.jboss.util.jmx.MBeanProxy.invoke(MBeanProxy.java:174)
        at $Proxy4.deploy(Unknown Source)
        at
org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:427)
        at
org.jboss.deployment.scanner.URLDeploymentScanner.scanDirectory(URLDeploymentScanner.java:648)
        at
org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:499)
        at
org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(AbstractDeploymentScanner.java:202)
        at
org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(AbstractDeploymentScanner.java:191)
15:11:24,093 INFO [EjbModule] Creating
15:11:24,140 INFO [EjbModule] Deploying chap2.EchoBean
15:11:24,187 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
```

```
        at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:149)
<snip>
...
15:11:24,250 INFO [EjbMBeanAdaptor] Creating
15:11:24,250 INFO [EjbMBeanAdaptor] Notified of create state
15:11:24,265 INFO [EjbMBeanAdaptor] Created
15:11:24,265 INFO [EjbModule] Created
15:11:24,281 INFO [EjbModule] Starting
15:11:24,296 INFO [EjbMBeanAdaptor] Starting
15:11:24,296 INFO [EjbMBeanAdaptor] Notified of start state
15:11:24,312 INFO [EjbMBeanAdaptor] Started
15:11:24,312 INFO [EjbModule] Started
15:11:24,312 INFO [MainDeployer] Deployed package: file:/D:/usr/JBoss3.0/jboss-all/build/
output/jboss-3.0.1RC1/server/default/deploy/chap2-ex3.ear
```

The stack traces are not exceptions. They are traces coming from line 150 of the EjbMBeanAdaptor code to demonstrate that clients ask for the MBean interface when they want to discover the MBean's capabilities. Notice that the EJB container (lines with [EjbModule]) is started before the example MBean (lines with [EjbMBeanAdaptor]).

Now, let's invoke the echo method using the JMX console web application. Browse to <http://localhost:8080/jmx-console/>

HtmlAdaptor?action=inspectMBean&name=jboss.book%3Aservice%3DEjbMBeanAdaptor and scroll down to the echo operation section. The view should be like that shown in Figure 2-10.

Name	Type	Access	Value
HelloPrefix	java.lang.String	RW	AdaptorPrefix
EjbJndiName	java.lang.String	RW	local/chap2.EchoBean

List of MBean operations:

java.lang.String echo()

Param	ParamType	ParamValue
arg0	java.lang.String	-echo-arg

Invoke

FIGURE 2-10. The EjbMBeanAdaptor MBean operations JMX console view

As shown, we have already entered an argument string of “-echo-arg” into the ParamValue text field. Press the Invoke button and a result string of “AdaptorPrefix-echo-arg” is displayed on the results page. The server console will show several stack traces from the various metadata queries issued by the JMX console and the MBean invoke method debugging lines:

```
15:37:41,312 INFO [EjbMBeanAdaptor] Begin invoke, actionPerformed=echo
15:37:41,312 INFO [EjbMBeanAdaptor] Lookup EchoLocalHome@local/chap2.EchoBean
15:37:41,328 INFO [EjbMBeanAdaptor] Result: AdaptorPrefix-echo-arg
15:37:41,328 INFO [EjbMBeanAdaptor] End invoke, actionPerformed=echo
```

The JBoss Deployer Architecture

JBoss has an extensible deployment architecture that allows one to incorporate components into the bare JBoss JMX microkernel. Figure 2-11 shows the classes in the deployment layer.

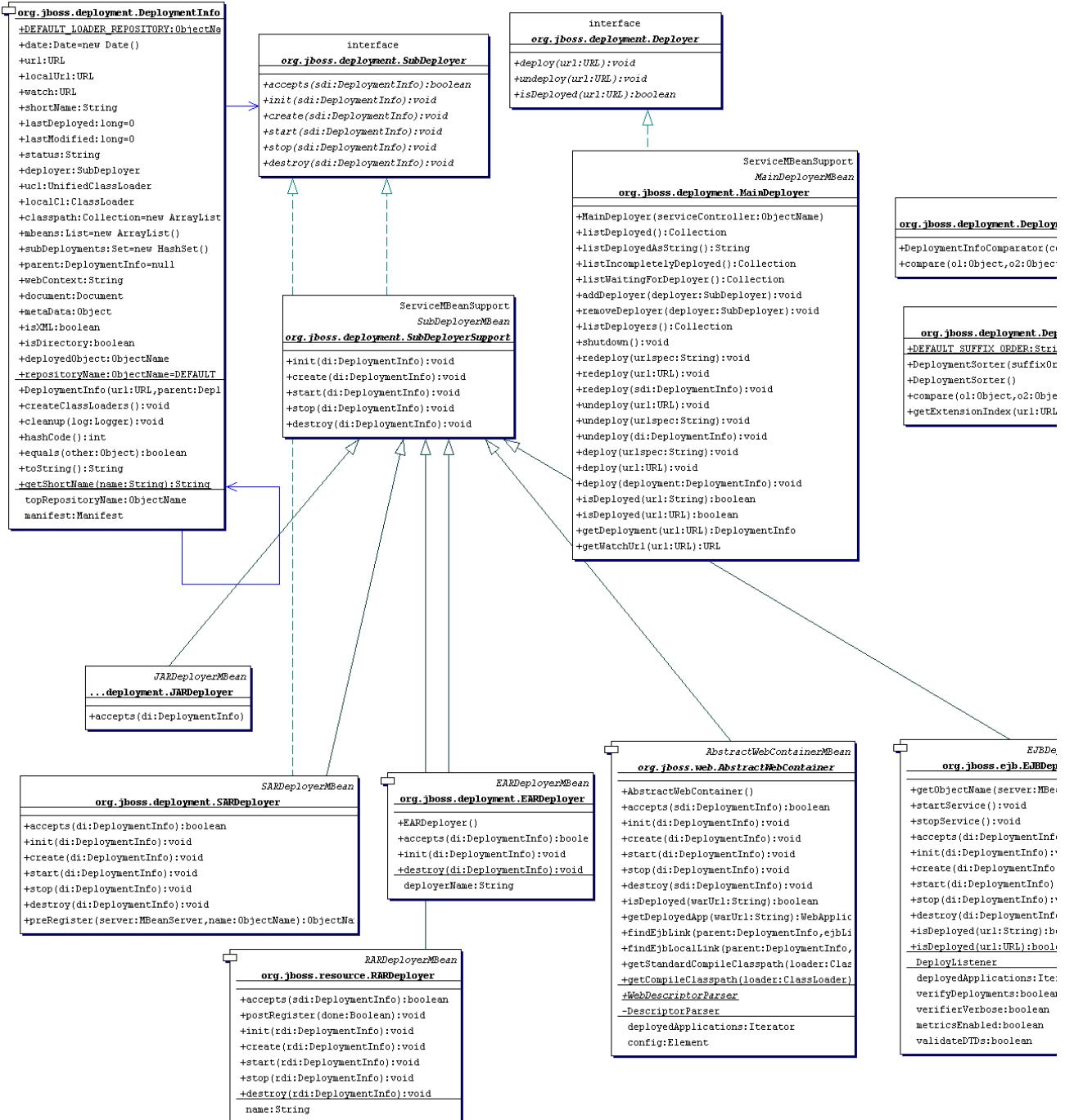


FIGURE 2-11. The deployment layer classes

The `MainDeployer` is the deployment entry point. Requests to deploy a component are sent to the `MainDeployer` and it determines if there is a subdeployer capable of handling the deployment, and if there is, it delegates the deployment to the subdeployer. We saw an example of this when we looked

at how the MainDeployer used the SARDeployer to deploy MBean services. The current deployers included with JBoss are:

- **AbstractWebContainer**: This subdeployer handles web application archives (WARs). It accepts deployment archives and directories whose name ends with a “war” suffix. WARs must have a WEB-INF/web.xml descriptor and may have a WEB-INF/jboss-web.xml descriptor.
- **EARDeployer**: This subdeployer handles enterprise application archives (EARs). It accepts deployment archives and directories whose name ends with an “ear” suffix. EARs must have a META-INF/application.xml descriptor and may have a META-INF/jboss-app.xml descriptor.
- **EJBDeployer**: This subdeployer handles enterprise bean jars. It accepts deployment archives and directories whose name ends with a “jar” suffix. EJB jars must have a META-INF/ejb-jar.xml descriptor and may have a META-INF/jboss.xml descriptor.
- **JARDeployer**: This subdeployer handles library jar archives. The only restriction it places on an archive is that it cannot contain a WEB-INF directory.
- **RARDeployer**: This subdeployer handles JCA resource archives (RARs). It accepts deployment archives and directories whose name ends with a “rar” suffix. RARs must have a META-INF/ra.xml descriptor.
- **SARDeployer**: This subdeployer handles JBoss MBean service archives (SARs). It accepts deployment archives and directories whose name ends with a “sar” suffix, as well as standalone XML files that end with “service.xml”. SARs that are jars must have a META-INF/jboss-service.xml descriptor.

The MainDeployer, JARDeployer and SARDeployer are hard coded deployers in the JBoss server core. The AbstractWebContainer, EARDeployer, EJBDeployer, and RARDeployer are MBean services that register themselves as deployers with the MainDeployer using the addDeployer(SubDeployer) operation. The SubDeployer interface is given in Listing 2-16:

LISTING 2-16. The org.jboss.deployment.SubDeployer interface

```
public interface SubDeployer
{
    /**
     * The <code>accepts</code> method is called by MainDeployer to
     * determine which deployer is suitable for a DeploymentInfo.
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @return a <code>boolean</code> value
     *
     * @jmx:managed-operation
     */
    boolean accepts(DeploymentInfo sdi);

    /**
     * The <code>init</code> method lets the deployer set a few properties
     * of the DeploymentInfo, such as the watch url.
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @throws DeploymentException if an error occurs
     */
}
```

```
* @jmx:managed-operation
*/
void init(DeploymentInfo sdi) throws DeploymentException;

/**
 * Set up the components of the deployment that do not
 * refer to other components
 *
 * @param sdi a <code>DeploymentInfo</code> value
 * @throws DeploymentException Failed to deploy
 *
 * @jmx:managed-operation
*/
void create(DeploymentInfo sdi) throws DeploymentException;

/**
 * The <code>start</code> method sets up relationships with other components.
 *
 * @param sdi a <code>DeploymentInfo</code> value
 * @throws DeploymentException if an error occurs
 *
 * @jmx:managed-operation
*/
void start(DeploymentInfo sdi) throws DeploymentException;

/**
 * The <code>stop</code> method removes relationships between components.
 *
 * @param sdi a <code>DeploymentInfo</code> value
 * @throws DeploymentException if an error occurs
 *
 * @jmx:managed-operation
*/
void stop(DeploymentInfo sdi) throws DeploymentException;

/**
 * The <code>destroy</code> method removes individual components
 *
 * @param sdi a <code>DeploymentInfo</code> value
 * @throws DeploymentException if an error occurs
 *
 * @jmx:managed-operation
*/
void destroy(DeploymentInfo sdi) throws DeploymentException;
}
```

The DeploymentInfo object is basically a data structure that encapsulates the complete state of a deployable component. When the MainDeployer receives a deployment request, it iterates through its registered subdeployers and invokes the accepts(DeploymentInfo) method on the subdeployer. The first subdeployer to return true is chosen and the deployment deployer and the MainDeployer will delegate the init, create, start, stop and destroy deployment life cycle operations to the subdeployer.

Naming on JBoss - The JNDI Naming Service

This chapter discusses the JBoss JNDI based naming service and the role of JNDI in JBoss and J2EE. An introduction to the basic JNDI API and common usage conventions will also be discussed. The JBoss specific configuration of J2EE component naming environments defined by the standard deployment descriptors will also be addressed. The final topic is the configuration and architecture of the JBoss naming service component, Naming/JBoss.

The JBoss naming service is an implementation of the Java Naming and Directory Interface (JNDI). JNDI plays a key role in J2EE because it provides a naming service that allows a user to map a name onto an object. This is a fundamental need in any programming environment because developers and administrators want to be able to refer to objects and services by recognizable names. A good example of a pervasive naming service is the Internet Domain Name System (DNS). The DNS service allows you to refer to hosts using logical names, rather than their numeric Internet addresses. JNDI serves a similar role in J2EE by enabling developers and administrators to create name-to-object bindings for use in J2EE components.

An Overview of JNDI

JNDI is a standard Java API that is bundled with JDK1.3 and higher. JNDI provides a common interface to a variety of existing naming services: DNS, LDAP, Active Directory, RMI registry, COS registry, NIS, and file systems. The JNDI API is divided logically into a client API that is used to access naming services, and a service provider interface (SPI) that allows the user to create JNDI implementations for naming services.

The SPI layer is an abstraction that naming service providers must implement to enable the core JNDI classes to expose the naming service using the common JNDI client interface. An implementation of JNDI for a naming service is referred to as a *JNDI provider*. JBoss naming is an example JNDI imple-

mentation, based on the SPI classes. Note that the JNDI SPI is not needed by J2EE component developers.

For a thorough introduction and tutorial on JNDI, which covers both the client and service provider APIs, see the Sun tutorial at <http://java.sun.com/products/jndi/tutorial/>.

The JNDI API

The main JNDI API package is the javax.naming package. It contains five interfaces, 10 classes, and several exceptions. There is one key class, InitialContext, and two key interfaces, Context and Name.

Names

The notion of a name is of fundamental importance in JNDI. The naming system determines the syntax that the name must follow. The syntax of the naming system allows the user to parse string representations of names into its components. A name is used with a naming system to locate objects. In the simplest sense, a naming system is just a collection of objects with unique names. To locate an object in a naming system you provide a name to the naming system, and the naming system returns the object store under the name.

As an example, consider the Unix file system's naming convention. Each file is named from its path relative to the root of the file system, with each component in the path separated by the forward slash character ("/"). The file's path is ordered from left to right. The pathname, /usr/jboss/readme.txt, for example, names a file readme.txt in the directory jboss, under the directory usr, located in the root of the file system. JBoss naming uses a Unix-style namespace as its naming convention.

The javax.naming.Name interface represents a generic name as an ordered sequence of components. It can be a composite name (one that spans multiple namespaces), or a compound name (one that is used within a single hierarchical naming system). The components of a name are numbered. The indexes of a name with N components range from 0 up to, but not including, N. The most significant component is at index 0. An empty name has no components.

A composite name is a sequence of component names that span multiple namespaces. An example of a composite name would be the hostname+file commonly used with Unix commands like scp. For example, this command copies localfile.txt to the file remotefile.txt in the tmp directory on host ahost.someorg.org:

```
scp localfile.txt ahost.someorg.org:/tmp/remotefile.txt
```

A compound name is derived from a hierarchical namespace. Each component in a compound name is an atomic name, meaning a string that cannot be parsed into smaller components. A file pathname in the Unix file system is an example of a compound name. The ahost.someorg.org:/tmp/remotefile.txt is a composite name that spans the DNS and Unix file system namespaces. The components of the composite name are ahost.someorg.org and /tmp/remotefile.txt. A component is a string name from the namespace of a naming system. If the component comes from a hierarchical namespace, that component can be further parsed into its atomic parts by using the javax.naming.CompoundName class.

The JNDI API provides the `javax.naming.CompositeName` class as the implementation of the Name interface for composite names.

Contexts

The `javax.naming.Context` interface is the primary interface for interacting with a naming service. The Context interface represents a set of name-to-object bindings. Every context has an associated naming convention that determines how the context parses string names into `javax.naming.Name` instances. To create a name to object binding you invoke the bind method of a Context and specify a name and an object as arguments. The object can later be retrieved using its name using the Context lookup method. A Context will typically provide operations for binding a name to an object, unbinding a name, and obtaining a listing of all name-to-object bindings. The object you bind into a Context can itself be of type Context. The Context object that is bound is referred to as a subcontext of the Context on which the bind method was invoked.

As an example, consider a file directory with a pathname /usr, which is a context in the Unix file system. A file directory named relative to another file directory is a subcontext (commonly referred to as a subdirectory). A file directory with a pathname /usr/jboss names a jboss context that is a subcontext of `usr`. In another example, a DNS domain, such as `org`, is a context. A DNS domain named relative to another DNS domain is another example of a subcontext. In the DNS domain `jboss.org`, the DNS domain `jboss` is a subcontext of `org` because DNS names are parsed right to left.

OBTAINING A CONTEXT USING INITIALCONTEXT

All naming service operations are performed on some implementation of the `Context` interface. Therefore, you need a way to obtain a `Context` for the naming service you are interested in using. The `javax.naming.InitialContext` class implements the `Context` interface, and provides the starting point for interacting with a naming service.

When you create an `InitialContext`, it is initialized with properties from the environment. JNDI determines each property's value by merging the values from the following two sources, in order such as:

- The first occurrence of the property from the constructor's environment parameter and (for appropriate properties) the applet parameters and system properties.
- All `jndi.properties` resource files found on the classpath.

For each property found in both of these two sources, the property's value is determined as follows. If the property is one of the standard JNDI properties that specify a list of JNDI factories, all of the values are concatenated into a single, colon-separated list. For other properties, only the first value found is used. The preferred method of specifying the JNDI environment properties is through a `jndi.properties` file. The reason is that this allows your code to externalize the JNDI provider specific information, and changing JNDI providers will not require changes to your code; thus it avoids the need to recompile to be able to see the change.

The `Context` implementation used internally by the `InitialContext` class is determined at runtime. The default policy uses the environment property "java.naming.factory.initial", which contains the class

name of the javax.naming.spi.InitialContextFactory implementation. You obtain the name of the InitialContextFactory class from the naming service provider you are using.

Listing 3-1 gives a sample jndi.properties file a client application would use to connect to a JBossNS service running on the local host at port 1099. The client application would need to have the jndi.properties file available on the application classpath. These are the properties that the JBossNS JNDI implementation requires. Other JNDI providers will have different properties and values.

LISTING 3-1. A sample jndi.properties file

```
### JBossNS properties
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

J2EE and JNDI – The Application Component Environment

JNDI is a fundamental aspect of the J2EE specifications. One key usage is the isolation of J2EE component code from the environment in which the code is deployed. Use of the application component's environment allows the application component to be customized without the need to access or change the application component's source code. The application component environment is sometimes referred to as the enterprise naming context (ENC). It is the responsibility of the application component container to make an ENC available to the container components in the form of JNDI Context. The ENC is utilized by the participants involved in the life cycle of a J2EE component in the following ways:

1. Application component business logic should be coded to access information from its ENC. The component provider uses the standard deployment descriptor for the component to specify the required ENC entries. The entries are declarations of the information and resources the component requires at runtime.
2. The container provides tools that allow a deployer of a component to map the ENC references made by the component developer to the deployment environment entity that satisfies the reference.
3. The component deployer utilizes the container tools to ready a component for final deployment.
4. The component container uses the deployment package information to build the complete component ENC at runtime

The complete specification regarding the use of JNDI in the J2EE platform can be found in Section 5 of the J2EE 1.3 specification. The J2EE specification is available at <http://java.sun.com/j2ee/download.html>.

An application component instance locates the ENC using the JNDI API. An application component instance creates a javax.naming.InitialContext object by using the no argument constructor and then looks up the naming environment under the name `java:comp/env`. The application component's environment entries are stored directly in the ENC, or in its subcontexts. Listing 3-2 illustrates the prototypical lines of code a component uses to access its ENC.

LISTING 3-2. ENC access sample code

```
// Obtain the application component's ENC  
Context iniCtx = new InitialContext();  
Context compEnv = (Context) iniCtx.lookup("java:comp/env");
```

An application component environment is a local environment that is accessible only by the component when the application server container thread of control is interacting with the application component. This means that an EJB Bean1 cannot access the ENC elements of EJB Bean2, and visa-versa. Similarly, Web application Web1 cannot access the ENC elements of Web application Web2 or Bean1 or Bean2 for that matter. Also, arbitrary client code, whether it is executing inside of the application server VM or externally cannot access a component's java:comp JNDI context. The purpose of the ENC is to provide an isolated, read-only namespace that the application component can rely on regardless of the type of environment in which the component is deployed. The ENC must be isolated from other components because each component defines its own ENC content, and components A and B may define the same name to refer to different objects. For example, EJB Bean1 may define an environment entry java:comp/env/red to refer to the hexadecimal value for the RGB color for red, while Web application Web1 may bind the same name to the deployment environment language locale representation of red.

There are three commonly used levels of naming scope in the JBossNS implementation – names under java:comp, names under java:, and any other name. As discussed, the java:comp context and its subcontexts are only available to the application component associated with the java:comp context. Subcontexts and object bindings directly under java: are only visible within the JBoss server virtual machine. Any other context or object binding is available to remote clients, provided the context or object supports serialization. You'll see how the isolation of these naming scopes is achieved in the section titled "The JBossNS Architecture"

An example of where the restricting a binding to the java: context is useful would be a [javax.sql.DataSource](#) connection factory that can only be used inside of the JBoss VM where the associated database pool resides. An example of a globally visible name that should accessible by remote client is an EJB home interface.

ENC Usage Conventions

JNDI is used as the API for externalizing a great deal of information from an application component. The JNDI name that the application component uses to access the information is declared in the standard ejb-jar.xml deployment descriptor for EJB components, and the standard web.xml deployment descriptor for Web components. Several different types of information may be stored in and retrieved from JNDI including:

- Environment entries as declared by the env-entry elements
- EJB references as declared by ejb-ref and ejb-local-ref elements.
- Resource manager connection factory references as declared by the resource-ref elements
- Resource environment references as declared by the resource-env-ref elements

Each type of deployment descriptor element has a JNDI usage convention with regard to the name of the JNDI context under which the information is bound. Also, in addition to the standard deployment descriptor element, there is a JBoss server specific deployment descriptor element that maps the JNDI name as used by the application component to the deployment environment JNDI name.

THE EJB-JAR.XML ENC ELEMENTS

The EJB 2.0 deployment descriptor describes a collection of EJB components and their environment. Each of the three types of EJB components—session, entity, and message-driven—support the specification of an EJB local naming context. The ejb-jar.xml description is a logical view of the environment that the EJB needs to operate. Because the EJB component developer generally cannot know into what environment the EJB will be deployed, the developer describes the component environment in a deployment environment independent manner using logical names. It is the responsibility of a deployment administrator to link the EJB component logical names to the corresponding deployment environment resources.

Figure 3-1 gives a graphical view of the EJB deployment descriptor DTD without the non-ENC elements. Only the session element is shown fully expanded as the ENC elements for entity and message-driven are identical. The full ejb-jar.xml DTD is available from the Sun Web site at the ENC elements in the standard EJB 2.0 ejb-jar.xml deployment descriptor.

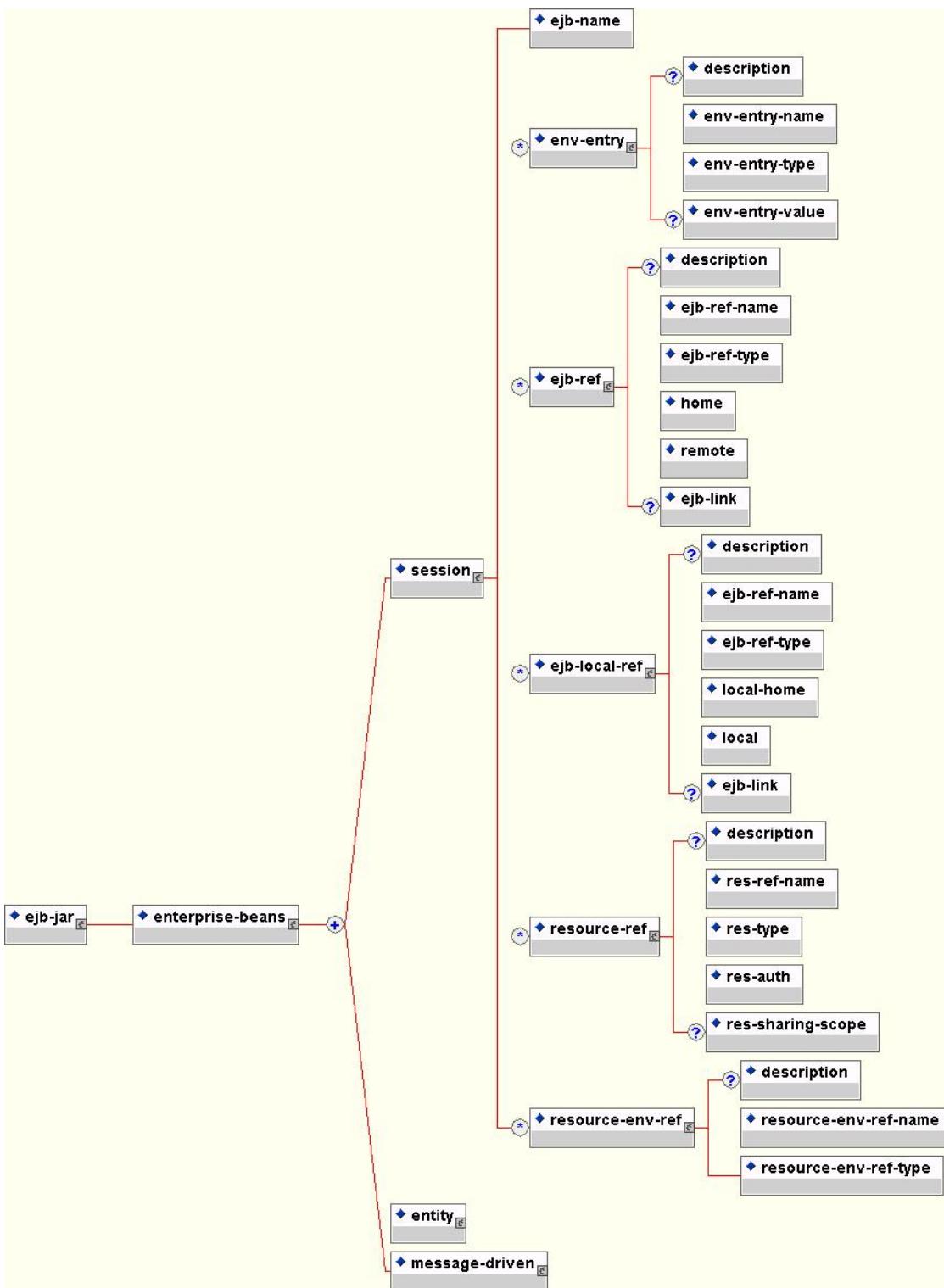


FIGURE 3-1. The ENC elements in the standard ejb-jar.xml 2.0 deployment descriptor.

THE WEB.XML ENC ELEMENTS

The Servlet 2.3 deployment descriptor describes a collection of Web components and their environment. The ENC for a Web application is declared globally for all servlets and JSP pages in the Web application. Because the Web application developer generally cannot know into what environment the Web application will be deployed, the developer describes the component environment in a deployment environment independent manner using logical names. It is the responsibility of a deployment administrator to link the Web component logical names to the corresponding deployment environment resources.

Figure 3-2 gives a graphical view of the Web application deployment descriptor DTD without the non-ENC elements. The full web.xml DTD is available from the Sun Web site at http://java.sun.com/dtd/web-app_2_3.dtd.

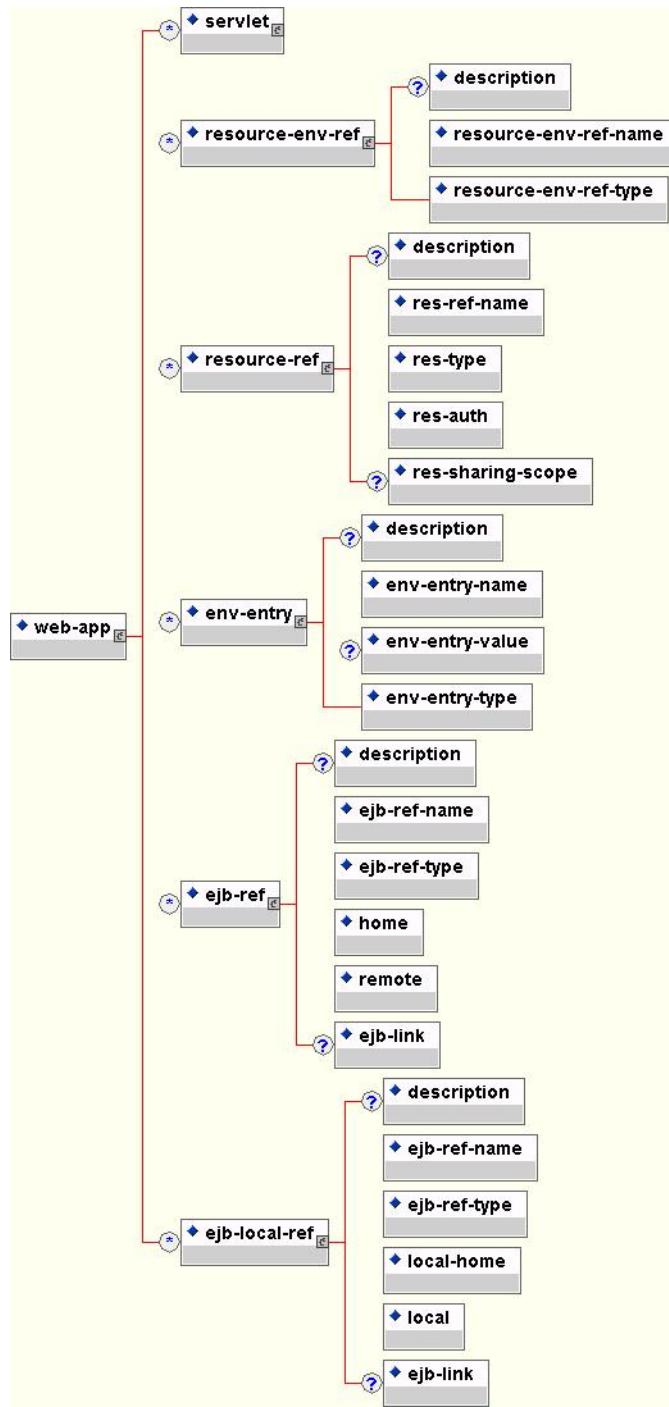


FIGURE 3-2. The ENC elements in the standard servlet 2.3 web.xml deployment descriptor.

THE JBOSS.XML ENC ELEMENTS

The JBoss EJB deployment descriptor provides the mapping from the EJB component ENC JNDI names to the actual deployed JNDI name. It is the responsibility of the application deployer to map

the logical references made by the application component to the corresponding physical resource deployed in a given application server configuration. In JBoss, this is done for the ejb-jar.xml descriptor using the jboss.xml deployment descriptor. Figure 3-3 gives a graphical view of the JBoss EJB deployment descriptor DTD without the non-ENC elements. Only the session element is shown fully expanded as the ENC elements for entity and message-driven are identical.

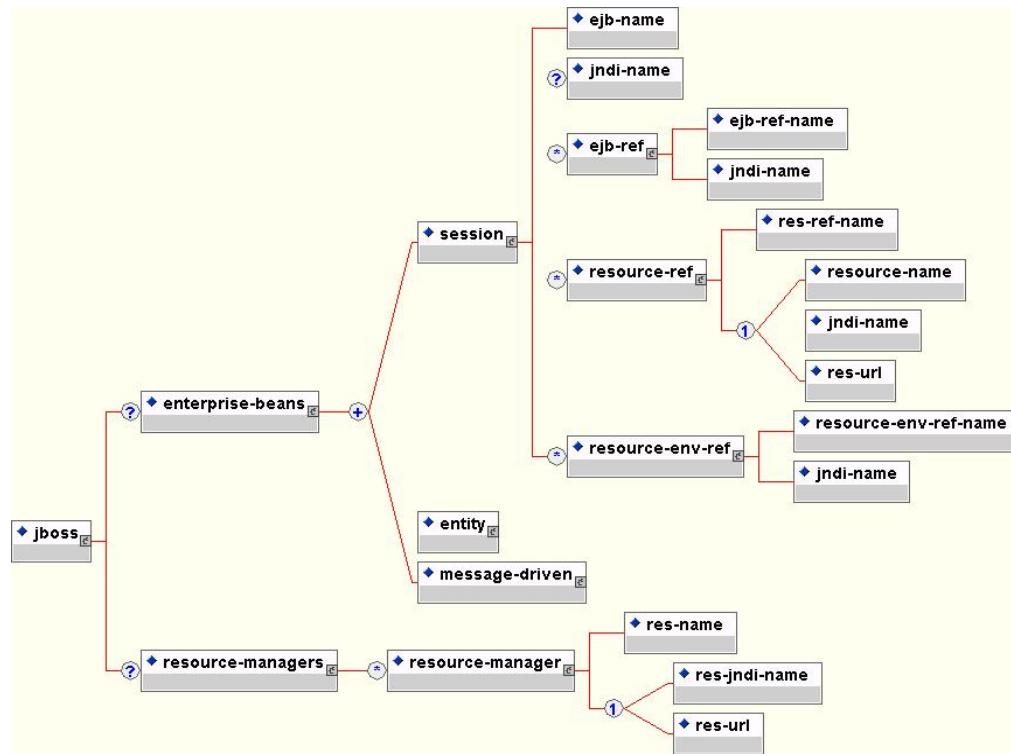


FIGURE 3-3. The ENC elements in the JBoss 3.0 jboss.xml deployment descriptor.

THE JBOSS-WEB.XML ENC ELEMENTS

The JBoss Web deployment descriptor provides the mapping from the Web application ENC JNDI names to the actual deployed JNDI name. It is the responsibility of the application deployer to map the logical references made by the Web application to the corresponding physical resource deployed in a given application server configuration. In JBoss, this is done for the web.xml descriptor using the jboss-web.xml deployment descriptor. Figure 3-4 gives a graphical view of the JBoss Web deployment descriptor DTD without the non-ENC elements. The full jboss-web.xml DTD is available from the JBoss Web site at http://www.jboss.org/j2ee/dtd/jboss_web.dtd.

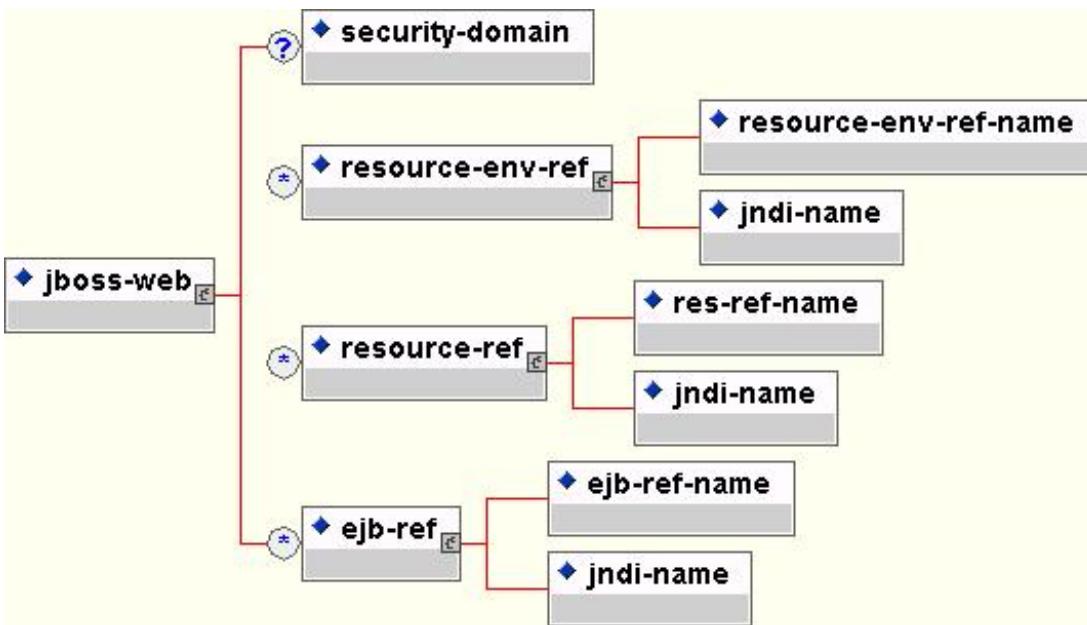


FIGURE 3-4. The ENC elements in the JBoss 3.0 jboss-web.xml deployment descriptor.

ENVIRONMENT ENTRIES

Environment entries are the simplest form of information stored in a component ENC, and are similar to operating system environment variables like those found on Unix or Windows. Environment entries are a name-to-value binding that allows a component to externalize a value and refer to the value using a name.

An environment entry is declared using an env-entry element in the standard deployment descriptors. The env-entry element contains the following child elements:

- An optional description element that provides a description of the entry
- An env-entry-name element giving the name of the entry relative to `java:comp/env`
- An env-entry-type element giving the Java type of the entry value that must be one of:
 - `java.lang.Byte`
 - `java.lang.Boolean`
 - `java.lang.Character`
 - `java.lang.Double`
 - `java.lang.Float`
 - `java.lang.Integer`
 - `java.lang.Long`
 - `java.lang.Short`
 - `java.lang.String`
- An env-entry-value element giving the value of entry as a string

An example of an env-entry fragment from an ejb-jar.xml deployment descriptor is given in Listing 3-3. There is no JBoss specific deployment descriptor element because an env-entry is a complete name and value specification. Listing 3-4 shows a sample code fragment for accessing the maxExemptions and taxRate env-entry values declared in Listing 3-3.

LISTING 3-3. An example ejb-jar.xml env-entry fragment

```
...
<session>
  <ejb-name>ASessionBean</ejb-name>
...
  <env-entry>
    <description>The maximum number of tax exemptions allowed
    </description>
    <env-entry-name>maxExemptions</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>15</env-entry-value>
  </env-entry>

  <env-entry>
    <description>The tax rate
    </description>
    <env-entry-name>taxRate</env-entry-name>
    <env-entry-type>java.lang.Float</env-entry-type>
    <env-entry-value>0.23</env-entry-value>
  </env-entry>
</session>
...
```

LISTING 3-4. ENC env-entry access code fragment

```
InitialContext iniCtx = new InitialContext();
Context envCtx = (Context) iniCtx.lookup("java:comp/env");
Integer maxExemptions = (Integer) envCtx.lookup("maxExemptions");
Float taxRate = (Float) envCtx.lookup("taxRate");
```

EJB REFERENCES

It is common for EJBs and Web components to interact with other EJBs. Because the JNDI name under which an EJB home interface is bound is a deployment time decision, there needs to be a way for a component developer to declare a reference to an EJB that will be linked by the deployer. EJB references satisfy this requirement.

An EJB reference is a link in an application component naming environment that points to a deployed EJB home interface. The name used by the application component is a logical link that isolates the component from the actual name of the EJB home in the deployment environment. The J2EE specification recommends that all references to enterprise beans be organized in the java:comp/env/ejb context of the application component's environment.

An EJB reference is declared using an `ejb-ref` element in the deployment descriptor. Each `ejb-ref` element describes the interface requirements that the referencing application component has for the referenced enterprise bean. The `ejb-ref` element contains the following child elements:

- An optional `description` element that provides the purpose of the reference.
- An `ejb-ref-name` element that specifies the name of the reference relative to the `java:comp/env` context. To place the reference under the recommended `java:comp/env/ejb` context, use an `ejb-link-name` form for the `ejb-ref-name` value.
- An `ejb-ref-type` element that specifies the type of the EJB. This must be either Entity or Session.
- A `home` element that gives the fully qualified class name of the EJB home interface.
- A `remote` element that gives the fully qualified class name of the EJB remote interface.
- An optional `ejb-link` element that links the reference to another enterprise bean in the ejb-jar file or in the same J2EE application unit. The `ejb-link` value is the `ejb-name` of the referenced bean. If there are multiple enterprise beans with the same `ejb-name`, the value uses the path name specifying the location of the ejb-jar file that contains the referenced component. The path name is relative to the referencing ejb-jar file. The Application Assembler appends the `ejb-name` of the referenced bean to the path name separated by `#`. This allows multiple beans with the same name to be uniquely identified.

An EJB reference is scoped to the application component whose declaration contains the `ejb-ref` element. This means that the EJB reference is not accessible from other application components at runtime, and that other application components may define `ejb-ref` elements with the same `ejb-ref-name` without causing a name conflict. Listing 3-5 provides an ejb-jar.xml fragment that illustrates the use of the `ejb-ref` element. A code sample that illustrates accessing the ShoppingCartHome reference declared in Listing 3-5 is given in Listing 3-6.

LISTING 3-5. An example ejb-jar.xml ejb-ref descriptor fragment

```

...
<session>
  <ejb-name>ShoppingCartBean</ejb-name>
  ...
</session>

<session>
  <ejb-name>ProductBeanUser</ejb-name>
  ...
<ejb-ref>
  <description>This is a reference to the store products entity
  </description>
  <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>org.jboss.store.ejb.ProductHome</home>
</ejb-ref>  <remote> org.jboss.store.ejb.Product</remote>
</session>

<session>
<ejb-ref>
  <ejb-name>ShoppingCartUser</ejb-name>

```

```
...
<ejb-ref-name>ejb/ShoppingCartHome</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>org.jboss.store.ejb.ShoppingCartHome</home>
<remote> org.jboss.store.ejb.ShoppingCart</remote>
<ejb-link>ShoppingCartBean</ejb-link>
</ejb-ref>
</session>

<entity>
  <description>The Product entity bean
  </description>
  <ejb-name>ProductBean</ejb-name>
  ...
</entity>
...
```

LISTING 3-6. ENC ejb-ref access code fragment

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ShoppingCartHome home = (ShoppingCartHome) ejbCtx.lookup("ShoppingCartHome");
```

EJB REFERENCES WITH JBOSS.XML AND JBOSS-WEB.XML

The JBoss server jboss.xml EJB deployment descriptor affects EJB references in two ways. First, the jndi-name child element of the session and entity elements allows the user to specify the deployment JNDI name for the EJB home interface. In the absence of a jboss.xml specification of the jndi-name for an EJB, the home interface is bound under the ejb-jar.xml ejb-name value. For example, the session EJB with the ejb-name of ShoppingCartBean in Listing 3.5 would have its home interface bound under the JNDI name ShoppingCartBean in the absence of a jboss.xml jndi-name specification.

The second use of the jboss.xml descriptor with respect to ejb-refs is the setting of the destination to which a component's ENC ejb-ref refers. The ejb-link element cannot be used to refer to EJBs in another enterprise application. If your ejb-ref needs to access an external EJB, you can specify the JNDI name of the deployed EJB home using the jboss.xml ejb-ref/jndi-name element.

The jboss-web.xml descriptor is used only to set the destination to which a Web application ejb-ref refers. The content model for the JBoss ejb-ref is as follows:

- An ejb-ref-name element that corresponds to the ejb-ref-name element in the ejb-jar.xml or web.xml standard descriptor
- A jndi-name element that specifies the JNDI name of the EJB home interface in the deployment environment

Listing 3-7 provides an example jboss.xml descriptor fragment that illustrates the following usage points:

- The ProductBeanUser ejb-ref link destination is set to the deployment name of jboss/store/ProductHome

- The deployment JNDI name of the ProductBean is set to jboss/store/ProductHome

LISTING 3-7. An example jboss.xml ejb-ref fragment

```

...
<session>
<ejb-name>ProductBeanUser</ejb-name>
<ejb-ref>
  <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
  <jndi-name>jboss/store/ProductHome</jndi-name>
</ejb-ref>
</session>

<entity>
  <ejb-name>ProductBean</ejb-name>
  <jndi-name>jboss/store/ProductHome</jndi-name>
  ...
</entity>
...

```

EJB LOCAL REFERENCES

In EJB 2.0 one can specify non-remote interfaces called local interfaces that do not use RMI call by value semantics. These interfaces use a call by reference semantic and therefore do not incur any RMI serialization overhead. An EJB local reference is a link in an application component naming environment that points to a deployed EJB local home interface. The name used by the application component is a logical link that isolates the component from the actual name of the EJB local home in the deployment environment. The J2EE specification recommends that all references to enterprise beans be organized in the java:comp/env/ejb context of the application component's environment.

An EJB local reference is declared using an ejb-local-ref element in the deployment descriptor. Each ejb-local-ref element describes the interface requirements that the referencing application component has for the referenced enterprise bean. The ejb-local-ref element contains the following child elements:

- An optional description element that provides the purpose of the reference.
- An ejb-ref-name element that specifies the name of the reference relative to the java:comp/env context. To place the reference under the recommended java:comp/env/ejb context, use an ejb-link-name form for the ejb-ref-name value.
- An ejb-ref-type element that specifies the type of the EJB. This must be either Entity or Session.
- A local-home element that gives the fully qualified class name of the EJB local home interface.
- A local element that gives the fully qualified class name of the EJB local interface.
- An ejb-link element that links the reference to another enterprise bean in the ejb-jar file or in the same J2EE application unit. The ejb-link value is the ejb-name of the referenced bean. If there are multiple enterprise beans with the same ejb-name, the value uses the path name specifying the location of the ejb-jar file that contains the referenced component. The path name is relative to the referencing ejb-jar file. The Application Assembler appends the ejb-name of the referenced bean to the path name separated by #. This allows multiple beans with the same name to be uniquely

identified. An ejb-link element must be specified in JBoss to match the local reference to the corresponding EJB.

An EJB local reference is scoped to the application component whose declaration contains the ejb-local-ref element. This means that the EJB local reference is not accessible from other application components at runtime, and that other application components may define ejb-local-ref elements with the same ejb-ref-name without causing a name conflict. Listing 3-8 provides an ejb-jar.xml fragment that illustrates the use of the ejb-local-ref element. A code sample that illustrates accessing the ProbeLocalHome reference declared in Listing 3-8 is given in Listing 3-9.

LISTING 3-8. An example ejb-jar.xml ejb-local-ref descriptor fragment

```
...
<session>
    <ejb-name>Probe</ejb-name>
    <home>org.jboss.test.perf.interfaces.ProbeHome</home>
    <remote>org.jboss.test.perf.interfaces.Probe</remote>
    <local-home>org.jboss.test.perf.interfaces.ProbeLocalHome</local-home>
    <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
    <ejb-class>org.jboss.test.perf.ejb.ProbeBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Bean</transaction-type>
</session>

<session>
    <ejb-name>PerfTestSession</ejb-name>
    <home>org.jboss.test.perf.interfaces.PerfTestSessionHome</home>
    <remote>org.jboss.test.perf.interfaces.PerfTestSession</remote>
    <ejb-class>org.jboss.test.perf.ejb.PerfTestSessionBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <ejb-ref>
        <ejb-ref-name>ejb/ProbeHome</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>org.jboss.test.perf.interfaces.SessionHome</home>
        <remote>org.jboss.test.perf.interfaces.Session</remote>
        <ejb-link>Probe</ejb-link>
    </ejb-ref>
    <ejb-local-ref>
        <ejb-ref-name>ejb/ProbeLocalHome</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local-home>org.jboss.test.perf.interfaces.ProbeLocalHome</local-home>
        <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
        <ejb-link>Probe</ejb-link>
    </ejb-local-ref>
</session>
...
```

LISTING 3-9. ENC ejb-local-ref access code fragment

```

InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ProbeLocalHome home = (ProbeLocalHome) ejbCtx.lookup("ProbeLocalHome");

```

RESOURCE MANAGER CONNECTION FACTORY REFERENCES

Resource manager connection factory references allow application component code to refer to resource factories using logical names called resource manager connection factory references. Resource manager connection factory references are defined by the resource-ref elements in the standard deployment descriptors. The Deployer binds the resource manager connection factory references to the actual resource manager connection factories that exist in the target operational environment using the jboss.xml and jboss-web.xml descriptors.

Each resource-ref element describes a single resource manager connection factory reference. The resource-ref element consists of the following child elements:

- An optional description element that provides the purpose of the reference.
- A res-ref-name element that specifies the name of the reference relative to the java:comp/env context. The resource type based naming convention for which subcontext to place the res-ref-name into is discussed in the next paragraph.
- A res-type element that specifies the fully qualified class name of the resource manager connection factory.
- A res-auth element that indicates whether the application component code performs resource signon programmatically, or whether the container signs on to the resource based on the principal mapping information supplied by the Deployer. It must be one of Application or Container.
- An option res-sharing-scope element. This currently is not supported by JBoss.

The J2EE specification recommends that all resource manager connection factory references be organized in the subcontexts of the application component's environment, using a different subcontext for each resource manager type. The recommended resource manager type to subcontext name is as follows:

- JDBC DataSource references should be declared in the java:comp/env/jdbc subcontext.
- JMS connection factories should be declared in the java:comp/env/jms subcontext.
- JavaMail connection factories should be declared in the java:comp/env/mail subcontext.
- URL connection factories should be declared in the java:comp/env/url subcontext.

Listing 3-10 shows an example web.xml descriptor fragment that illustrates the resource-ref element usage. Listing 3-11 provides a code fragment that an application component would use to access the DefaultMail resource defined in Listing 3-10.

LISTING 3-10. A web.xml resource-ref descriptor fragment

```

<web>
...
<servlet>
  <servlet-name>AServlet</servlet-name>

```

```
...
</servlet>
...
<!-- JDBC DataSources ( java:comp/env/jdbc ) -->
<resource-ref>
    <description>The default DS</description>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
<!-- JavaMail Connection Factories ( java:comp/env/mail ) -->
<resource-ref>
    <description>Default Mail</description>
    <res-ref-name>mail/DefaultMail</res-ref-name>
    <res-type>javax.mail.Session</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
<!-- JMS Connection Factories ( java:comp/env/jms ) -->
<resource-ref>
    <description>Default QueueFactory</description>
    <res-ref-name>jms/QueFactory</res-ref-name>
    <res-type>javax.jms.QueueConnectionFactory</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
</web>
```

LISTING 3-11. ENC resource-ref access sample code fragment

```
Context initCtx = new InitialContext();
javax.mail.Session s = (javax.mail.Session)
    initCtx.lookup("java:comp/env/mail/DefaultMail");
```

RESOURCE MANAGER CONNECTION FACTORY REFERENCES WITH JBOSS.XML AND JBOSS-WEB.XML

The purpose of the JBoss jboss.xml EJB deployment descriptor and jboss-web.xml Web application deployment descriptor is to provide the link from the logical name defined by the res-ref-name element to the JNDI name of the resource factory as deployed in JBoss. This is accomplished by providing a resource-ref element in the jboss.xml or jboss-web.xml descriptor. The JBoss resource-ref element consists of the following child elements:

- A res-ref-name element that must match the res-ref-name of a corresponding resource-ref element from the ejb-jar.xml or web.xml standard descriptors
- An optional res-type element that specifies the fully qualified class name of the resource manager connection factory
- A jndi-name element that specifies the JNDI name of the resource factory as deployed in JBoss

Listing 3-12 provides a sample jboss-web.xml descriptor fragment that shows sample mappings of the resource-ref elements given in Listing 3-10.

LISTING 3-12. A sample jboss-web.xml resource-ref descriptor fragment

```

<jboss-web>
...
<resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <jndi-name>java:/DefaultDS</jndi-name>
</resource-ref>
<resource-ref>
    <res-ref-name>mail/DefaultMail</res-ref-name>
    <res-type>javax.mail.Session</res-type>
    <jndi-name>java:/Mail</jndi-name>
</resource-ref>
<resource-ref>
    <res-ref-name>jms/QueFactory</res-ref-name>
    <res-type>javax.jms.QueueConnectionFactory</res-type>
    <jndi-name>QueueConnectionFactory</jndi-name>
</resource-ref>
...
</jboss-web>

```

RESOURCE ENVIRONMENT REFERENCES

Resource environment references are elements that refer to administered objects that are associated with a resource (for example, JMS destinations) by using logical names. Resource environment references are defined by the resource-env-ref elements in the standard deployment descriptors. The Deployer binds the resource environment references to the actual administered objects location in the target operational environment using the jboss.xml and jboss-web.xml descriptors.

Each resource-env-ref element describes the requirements that the referencing application component has for the referenced administered object. The resource-env-ref element consists of the following child elements:

- An optional description element that provides the purpose of the reference.
- A resource-env-ref-name element that specifies the name of the reference relative to the java:comp/env context. Convention places the name in a subcontext that corresponds to the associated resource factory type. For example, a JMS queue reference named MyQueue should have a resource-env-ref-name of jms/MyQueue.
- A resource-env-ref-type element that specifies the fully qualified class name of the referenced object. For example, in the case of a JMS queue, the value would be javax.jms.Queue.

Listing 3-13 provides an example resource-ref-env element declaration by a session bean. Listing 3-14 gives a code fragment that illustrates

LISTING 3-13. An example ejb-jar.xml resource-env-ref fragment

```

<session>
<ejb-name>MyBean</ejb-name>

```

```
...
<resource-env-ref>
  <description>This is a reference to a JMS queue used in the
  processing of Stock info
  </description>
  <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
...
</session>
```

LISTING 3-14. ENC resource-env-ref access code fragment

```
InitialContext iniCtx = new InitialContext();
javax.jms.Queue q = (javax.jms.Queue)
    envCtx.lookup("java:comp/env/jms/StockInfo");
```

RESOURCE ENVIRONMENT REFERENCES AND JBOSS.XML, JBOSS-WEB.XML

The purpose of the JBoss jboss.xml EJB deployment descriptor and jboss-web.xml Web application deployment descriptor is to provide the link from the logical name defined by the resource-env-ref-name element to the JNDI name of the administered object deployed in JBoss. This is accomplished by providing a resource-env-ref element in the jboss.xml or jboss-web.xml descriptor. The JBoss resource-env-ref element consists of the following child elements:

- A resource-env-ref-name element that must match the resource-env-ref-name of a corresponding resource-env-ref element from the ejb-jar.xml or web.xml standard descriptors
- A jndi-name element that specifies the JNDI name of the resource as deployed in JBoss

Listing 3-15 provides a sample jboss.xml descriptor fragment that shows a sample mapping for the resource-env-ref element given in Listing 3-13.

LISTING 3-15. A sample jboss.xml resource-env-ref descriptor fragment

```
<session>
  <ejb-name>MyBean</ejb-name>
  ...
  <resource-env-ref>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <jndi-name>queue/StockInfoQue</jndi-name>
  </resource-env-ref>
  ...
</session>
```

The JBossNS Architecture

The JBossNS architecture is a Java socket/RMI based implementation of the `javax.naming.Context` interface. It is a client/server implementation that can be accessed remotely. The implementation is optimized so that access from within the same VM in which the JBossNS server is running does not involve sockets. Same VM access occurs through an object reference available as a global singleton. Figure 3-5 illustrates some of the key classes in the JBossNS implementation and their relationships.

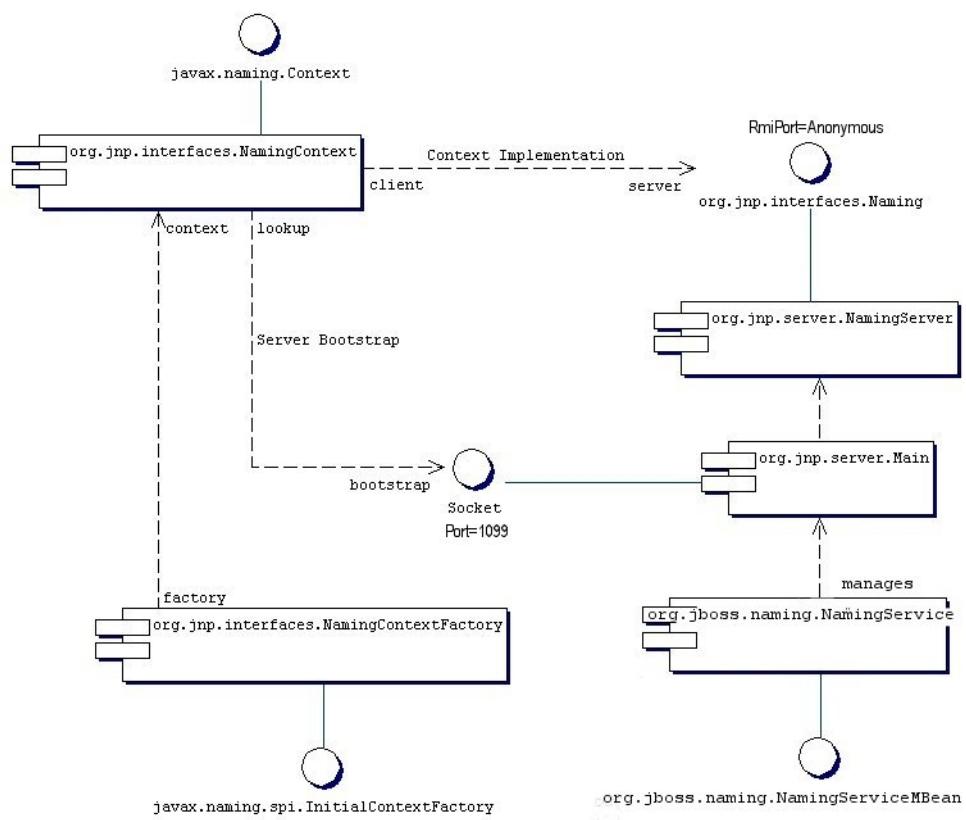


FIGURE 3-5. Key components in the JBossNS architecture.

We will start with the `NamingService` MBean. The `NamingService` MBean provides the JNDI naming service. This is a key service used pervasively by the J2EE technology components. The configurable attributes for the `NamingService` are as follows:

- **Port:** The jnp protocol listening port for the `NamingService`. If not specified default is 1099, the same as the RMI registry default port.
- **RmiPort:** The RMI port on which the RMI `Naming` implementation will be exported. If not specified the default is 0 which means use any available port.

- **BindAddress**: the specific address the NamingService listens on. This can be used on a multi-homed host for a java.net.ServerSocket that will only accept connect requests on one of its addresses.
- **Backlog**: The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter. If a connection indication arrives when the queue is full, the connection is refused.
- **ClientSocketFactory**: An optional custom java.rmi.server.RMIClientSocketFactory implementation class name. If not specified the default RMIClientSocketFactory is used.
- **ServerSocketFactory**: An optional custom java.rmi.server.RMIServerSocketFactory implementation class name. If not specified the default RMIServerSocketFactory is used.
- **JNPSocketFactory**, An optional custom javax.net.ServerSocketFactory implementation class name. This is the factory for the ServerSocket used to bootstrap the download of the JBossNS Naming interface. If not specified the javax.net.ServerSocketFactory.getDefault() method value is used.

The NamingService also creates the java:comp context such that access to this context is isolated based on the context ClassLoader of the thread that accesses the java:comp context. This provides the application component private ENC that is required by the J2EE specs. The segregation of java:comp by ClassLoader is accomplished by binding a javax.naming.Reference to a Context that uses the org.jboss.naming.ENCFactory as its javax.naming.ObjectFactory. When a client performs a lookup of java:comp, or any subcontext, the ENCFactory checks the thread context ClassLoader, and performs a lookup into a map using the ClassLoader as the key. If a Context instance does not exist for the ClassLoader instance, one is created and associated with the ClassLoader in the ENCFactory map. Thus, correct isolation of an application component's ENC relies on each component receiving a unique ClassLoader that is associated with the component threads of execution. The NamingService delegates its functionality to an org.jnp.server.Main MBean. The reason for the duplicate MBeans is because JBossNS started out as a stand-alone JNDI implementation, and can still be run as such. The NamingService MBean embeds the Main instance into the JBoss server so that usage of JNDI with the same VM as the JBoss server does not incur any socket overhead. The configurable attributes of the NamingService are really the configurable attributes of the JBossNS Main MBean. The setting of any attributes on the NamingService MBean simply set the corresponding attributes on the Main MBean the NamingService contains. When the NamingService is started, it starts the contained Main MBean to activate the JNDI naming service. In addition, the NamingService exposes the Naming interface operations through a JMX detyped invoke operation. This allows the naming service to be accessed via arbitrary protocol to JMX adaptors. We will look at an example of how HTTP can be used to access the naming service using the invoke operation later in this chapter.

The details of threads and the thread context class loader won't be explored here, but the JNDI tutorial provides a concise discussion that is applicable. See <http://java.sun.com/products/jndi/tutorial/beyond/misc/classloader.html> for the details.

When the Main MBean is started, it performs the following tasks:

- Instantiates an org.jnp.naming.NamingService instance and sets this as the local VM server instance. This is used by any org.jnp.interfaces.NamingContext instances that are created within the JBoss server VM to avoid RMI calls over TCP/IP.

- Exports the NamingServer instance's org.jnp.naming.interfaces.Naming RMI interface using the configured RmiPort, ClientSocketFactory, ServerSocketFactory attributes.
- Creates a socket that listens on the interface given by the BindAddress and Port attributes.
- Spawns a thread to accept connections on the socket.

The Naming InitialContext Factories

The JBoss JNDI provider currently supports three different InitialContext factory implementations. The most commonly used factory is the org.jnp.interfaces.NamingContextFactory implementation. Its properties include:

- **java.naming.factory.initial (or Context.INITIAL_CONTEXT_FACTORY)**, The name of the environment property for specifying the initial context factory to use. The value of the property should be the fully qualified class name of the factory class that will create an initial context. If it is not specified, a javax.naming.NoInitialContextException will be thrown when an InitialContext object is created.
- **java.naming.provider.url (or Context.PROVIDER_URL)**, The name of the environment property for specifying the location of the JBoss JNDI service provider the client will use. The NamingContextFactory class uses this information to know which JBossNS server to connect to. The value of the property should be a URL string. For JBossNS the URL format is jnp://host:port/[jndi_path]. The jnp: portion of the URL is the protocol and refers to the socket/RMI based protocol used by JBoss. The jndi_path portion of the URL is an option JNDI name relative to the root context, for example, "apps" or "apps/tmp". Everything but the host component is optional. The following examples are equivalent because the default port value is 1099:
 - jnp://www.jboss.org:1099/
 - www.jboss.org:1099
 - www.jboss.org
- **java.naming.factory.url.pkgs (or Context.URL_PKG_PREFIXES)**, The name of the environment property for specifying the list of package prefixes to use when loading in URL context factories. The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory. For the JBoss JNDI provider this must be org.jboss.naming:org.jnp.interfaces. This property is essential for locating the jnp: and java: URL context factories of the JBoss JNDI provider.
- **jnp.socketFactory**, The fully qualified class name of the javax.net.SocketFactory implementation to use to create the bootstrap socket. The default value is org.jnp.interfaces.TimedSocketFactory. The TimedSocketFactory is a simple SocketFactory implementation that supports the specification of a connection and read timeout. These two properties are specified by:
 - **jnp.timeout**, The connection timeout in milliseconds. The default value is 0 which means the connection will block until the VM TCP/IP layer times out.
 - **jnp.sotimeout**, The connected socket read timeout in milliseconds. The default value is 0 which means reads will block. This is the value passed to the Socket.setSoTimeout on the newly connected socket.

When a client creates an InitialContext with these JBossNS properties available, the org.jnp.interfaces.NamingContextFactory object is used to create the Context instance that will be used in subse-

quent operations. The NamingContextFactory is the JBossNS implementation of the javax.naming.spi.InitialContextFactory interface. When the NamingContextFactory class is asked to create a Context, it creates an org.jnp.interfaces.NamingContext instance with the InitialContext environment and name of the context in the global JNDI namespace. It is the NamingContext instance that actually performs the task of connecting to the JBossNS server, and implements the Context interface. The Context.PROVIDER_URL information from the environment indicates from which server to obtain a NamingServer RMI reference.

The association of the NamingContext instance to a NamingServer instance is done in a lazy fashion on the first Context operation that is performed. When a Context operation is performed and the NamingContext has no NamingServer associated with it, it looks to see if its environment properties define a Context.PROVIDER_URL. A Context.PROVIDER_URL defines the host and port of the JBossNS server the Context is to use.. If there is a provider URL, the NamingContext first checks to see if a Naming instance keyed by the host and port pair has already been created by checking a NamingContext class static map. It simply uses the existing Naming instance if one for the host port pair has already been obtained. If no Naming instance has been created for the given host and port, the NamingContext connects to the host and port using a java.net.Socket, and retrieves a Naming RMI stub from the server by reading a java.rmi.MarshalledObject from the socket and invoking its get method. The newly obtained Naming instance is cached in the NamingContext server map under the host and port pair. If no provider URL was specified in the JNDI environment associated with the context, the NamingContext simply uses the in VM Naming instance set by the Main MBean.

The NamingContext implementation of the Context interface delegates all operations to the Naming instance associated with the NamingContext. The NamingServer class that implements the Naming interface uses a java.util.Hashtable as the Context store. There is one unique NamingServer instance for each distinct JNDI Name for a given JBossNS server. There are zero or more transient NamingContext instances active at any given moment that refers to a NamingServer instance. The purpose of the NamingContext is to act as a Context to the Naming interface adaptor that manages translation of the JNDI names passed to the NamingContext. Because a JNDI name can be relative or a URL, it needs to be converted into an absolute name in the context of the JBossNS server to which it refers. This translation is a key function of the NamingContext.

The HTTP InitialContext Factory Implementation

As of JBoss-3.0.2, support exists for accessing the JNDI naming service over HTTP. From a JNDI client's perspective this is a transparent change as they continue to use the JNDI Context interface. Operations through the Context interface are translated into HTTP posts to a servlet that passes the request to the NamingService using its JMX invoke operation. Advantages of using HTTP as the access protocol include better access through firewalls and proxies setup to allow HTTP, as well as the ability to secure access to the JNDI service using standard servlet role based security.

To access JNDI over HTTP you use the org.jboss.naming.HttpNamingContextFactory as the factory implementation. The complete set of support InitialContext environment properties for this factory are:

- **java.naming.factory.initial (or Context.INITIAL CONTEXT FACTORY)**, The name of the environment property for specifying the initial context factory, which must be org.jboss.naming.HttpNamingContextFactory.

- **java.naming.provider.url (or Context.PROVIDER_URL)**, This must be set to the http URL of the JMX invoker servlet. It depends on the configuration of the http-invoker.sar and its contained war, but the default setup places the JMX invoker servlet under /invoker/JMXInvokerServlet. The full http URL would be the public URL of the JBoss servlet container + “/invoker/JMXInvokerServlet“. Examples include
 - http://www.jboss.org:8080/invoker/JMXInvokerServlet
 - http://www.jboss.org/invoker/JMXInvokerServlet
 - https://www.jboss.org/invoker/JMXInvokerServlet

where the first accesses the servlet using the port 8080, the second uses the standard HTTP port 80, and the third uses an SSL encrypted connection to the standard HTTPS port 443.

- **java.naming.factory.url.pkgs (or Context.URL_PKG_PREFIXES)**, For all JBoss JNDI provider this must be org.jboss.naming:org.jnp.interfaces. This property is essential for locating the jnp: and java: URL context factories of the JBoss JNDI provider.

The JNDI Context implementation returned by the HttpNamingContextFactory is a proxy that delegates invocations made on it to a bridge servlet which forwards the invocation to the NamingService through the JMX bus, and marshalls the reply back over HTTP. The proxy needs to know what the URL of the bridge servlet is in order to operate. This value may have been bound on the server side if the JBoss web server has a well known public interface. If the JBoss web server is sitting behind one or more firewalls or proxies, the proxy cannot know what URL is required. In this case, the proxy will be associated with a system property value that must be set in the client VM. For more information on the operation of JNDI over HTTP See “Accessing JNDI over HTTP” on page 120.

The Login InitialContext Factory Implementation

Historically JBoss has not supported providing login information via the InitialContext factory environment. The reason being that JAAS provides a much more flexible framework. For simplicity and migration from other application server environment that do make use of this mechanism, JBoss-3.0.3 adds a new InitialContext factory implementation that allows this. JAAS is still used under in the implementation, but there is no manifest use of the JAAS interfaces in the client application.

The factory class that provides this capability is the org.jboss.security.jndi.LoginInitialContextFactory. The complete set of support InitialContext environment properties for this factory are:

- **java.naming.factory.initial (or Context.INITIAL_CONTEXT_FACTORY)**, The name of the environment property for specifying the initial context factory, which must be org.jboss.security.jndi.LoginInitialContextFactory.
- **java.naming.provider.url (or Context.PROVIDER_URL)**, This must be set to a NamingContextFactory provider URL. The LoginInitialContext is really just a wrapper around the NamingContextFactory that adds a JAAS login to the existing NamingContextFactory behavior.
- **java.naming.factory.url.pkgs (or Context.URL_PKG_PREFIXES)**, For all JBoss JNDI provider this must be org.jboss.naming:org.jnp.interfaces. This property is essential for locating the jnp: and java: URL context factories of the JBoss JNDI provider.
- **java.naming.security.principal (or Context.SECURITY_PRINCIPAL)**, The principal to authenticate. This may be either a java.security.Principal implementation or a string representing the name of a principal.

Context SECURITY CREDENTIALS

- **java.naming.security.credentials** (or **Context.SECURITY_CREDENTIALS**), The credentials that should be used to authenticate the principal, e.g., password, session key, etc.
- **java.naming.security.protocol** (or **Context.SECURITY_PROTOCOL**), This gives the name of the JAAS login module to use for the authentication of the principal and credentials.

Accessing JNDI over HTTP

In addition to the legacy RMI/JRMP with a socket bootstrap protocol, JBoss-3.0.3 provides support for accessing its JNDI naming service using HTTP. This capability is provided by the http-invoker.sar deployment and its contained services and servlets. The structure of the http-invoker.sar is:

```
http-invoker.sar
+- META-INF/jboss-service.xml
+- invoker.war
|   +- WEB-INF/jboss-web.xml
|   +- WEB-INF/classes/org/jboss/invocation/http/servlet/InvokerServlet.class
|   +- WEB-INF/classes/org/jboss/invocation/http/servlet/NamingAccessFilter.class
|   +- WEB-INF/classes/org/jboss/invocation/http/servlet/NamingFactoryServlet.class
|   +- WEB-INF/classes/org/jboss/invocation/http/servlet/ReadOnlyAccessFilter.class
|   +- WEB-INF/classes/roles.properties
|   +- WEB-INF/classes/users.properties
|   +- WEB-INF/web.xml
|   +- META-INF/MANIFEST.MF
+- META-INF/MANIFEST.MF
```

The http-invoker.sar jboss-service.xml descriptor defines the HttpInvoker and HttpInvokerHA MBeans. These services handle the routing of methods invocations that are sent via HTTP to the appropriate target MBean on the JMX bus.

The http-invoker.war web application contains servlets that handle the details of the HTTP transport. The NamingFactoryServlet handles creation requests for the JBoss JNDI naming service javax.naming.Context implementation. The InvokerServlet handles invocations made through RMI/HTTP clients. The ReadOnlyAccessFilter allows one to secure the JNDI naming service while making a single JNDI context available for read-only access by unauthenticated clients.

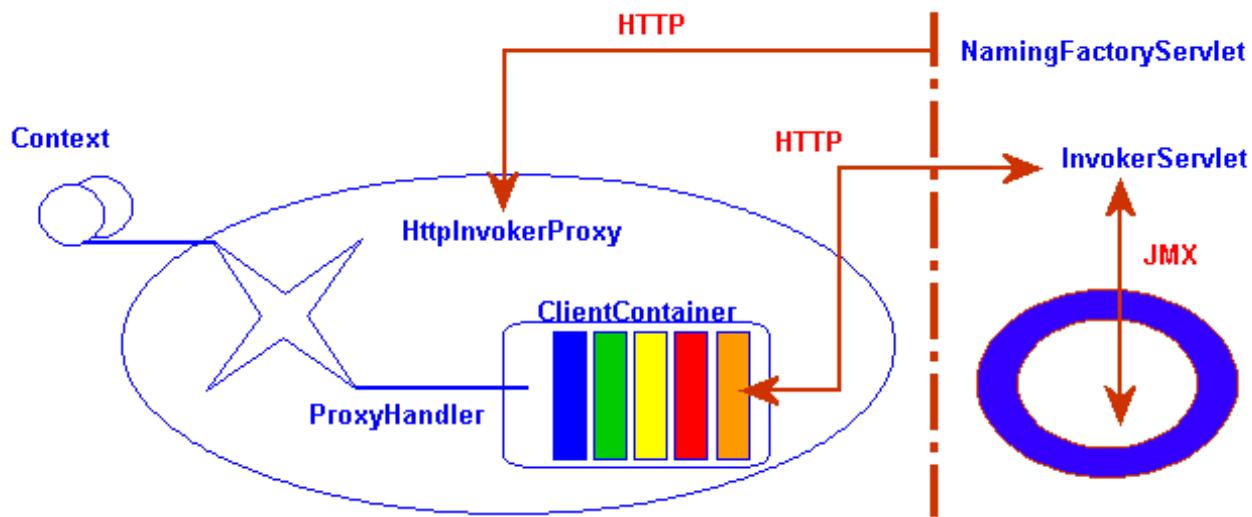


FIGURE 3-6. The HTTP invoker proxy/server structure for a JNDI Context

Before looking at the configurations let's overview the operation of the http-invoker services. Figure 3-6 shows a logical view of the structure of a JNDI Context proxy and its relationship to the JBoss server side components of the http-invoker. The proxy is obtained from the NamingFactoryServlet using an InitialContext with the org.jboss.naming.HttpNamingContextFactory class as the Context.INITIAL_CONTEXT_FACTORY value. The proxy is an instance of org.jboss.invocation.http.interfaces.HttpInvokerProxy, and implements the javax.naming.Context interface. Internally the HttpInvokerProxy contains an invoker that marshalls the Context interface via HTTP posts to the InvokerServlet. The InvokerServlet translates these posts into JMX invocations, and returns the invocation response back to the proxy in the HTTP post reponse.

There are several configuration values that need to be set to tie all of these components together.

The http URL used as the Context.PROVIDER_URL in the InitialContext environment when obtaining the HttpInvokerProxy from the NamingFactoryServlet. The settings that affect this URL include:

- The servlet mapping of the NamingFactoryServlet in the invoker.war web.xml descriptor. The default setting for the unsecured mapping is "/JNDIFactory/*".
- The interface and port on which the JBoss servlet container is running, or equivalently, the address and port of any proxy used to access the JBoss server.. The default setting is to bind on all interfaces on port 8080.
- Thus, the default URL would be "http://myhost:8080/invoker/JNDIFactory", where myhost is any name or IP address of the host you are running JBoss on. The "invoker" portion of the URL path is due to the default context into which the invoker.war is bound. Unless overridden by a jboss-web.xml context-root element setting, it defaults to the name of the war minus the ".war" suffix.

The http URL to the InvokerServlet used by the invoker contained in the HttpInvokerProxy. The settings that affect this URL include:

- The servlet mapping of the InvokerServlet in the http-invoker.war web.xml descriptor. The default setting for the unsecured mapping is “/JMXInvokerServlet/*”.
- The externalURL init parameter of the NamingFactoryServlet in the invoker.war web.xml descriptor. This can either be the http URL of the InvokerServlet, or the name of a system property that will be resolved in the client VM. The default http URL would be “http://myhost:8080/invoker/JMXInvokerServlet” based on the default webserver settings and servlet mapping for the InvokerServlet. The purpose of allowing the name of a system property is to allow for situations where the server cannot know how the client is connecting to the server as may be the case when a firewall or proxy sits between the client and server. In this case, the client application would set the system property to the hostname and port needed to access the proxy/firewall and include the “/invoker/JMXInvokerServlet“ servlet path component as well.
- The interface and port on which the JBoss servlet container is running, or equivalently, the address and port of any proxy used to access the JBoss server.

The JMX ObjectName of the NamingService MBean passed in by the HttpInvokerProxy for use by the InvokerServlet to route the invocation. The settings that affect this name include:

- The name attribute of the NamingService defined in the conf/jboss-service.xml descriptor. The standard setting used in the JBoss distributions is “jboss:service=Naming”
- The invokerName init parameter of the NamingFactoryServlet in the invoker.war web.xml descriptor. This parameter must match the name attribute of the NamingService

Securing Access to JNDI over HTTP

One benefit to accessing JNDI over HTTP is that it is easy to secure access to the JNDI InitialContext factory as well as the naming operations using standard web declarative security. This is possible because the server side handling of the JNDI/HTTP transport is implemented with two servlets. These servlets are included in the http-invoker.sar/invoker.war directory found in the default and all configuration deploy directories as shown previously. To enable secured access to JNDI you need to edit the invoker.war/WEB-INF/web.xml descriptor and remove all unsecured servlet mappings. For example, the web.xml descriptor shown in Listing 3-16 only allows access to the invoker.war servlets if the user has been authenticated and has a role of HttpInvoker.

LISTING 3-16. An example web.xml descriptor for secured access to the JNDI servlets

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ### Servlets -->
  <servlet>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <servlet-class>org.jboss.invocation.http.servlet.InvokerServlet</servlet-
class>
    <init-param>
```

```

        <param-name>invokerName</param-name>
        <param-value>jboss:service=invoker,type=http</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
    <servlet-name>JNDIFactory</servlet-name>
    <servlet-class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-class>
    <init-param>
        <param-name>invokerName</param-name>
        <param-value>jboss:service=Naming</param-value>
    </init-param>
    <init-param>
        <param-name>externalURL</param-name>
        <param-value>http://myhost:8080/invoker/JMXInvokerServlet</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!-- ### Servlet Mappings -->
<servlet-mapping>
    <servlet-name>JNDIFactory</servlet-name>
    <url-pattern>/restricted/JNDIFactory/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <url-pattern>/restricted/JMXInvokerServlet/*</url-pattern>
</servlet-mapping>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>HttpInvokers</web-resource-name>
        <description>An example security config that only allows users with the
            role HttpInvoker to access the HTTP invoker servlets
        </description>
        <url-pattern>/restricted/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>HttpInvoker</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>JBoss HTTP Invoker</realm-name>
</login-config>

<security-role>
    <role-name>HttpInvoker</role-name>
</security-role>
</web-app>

```

The web.xml descriptor only defines which sevlets are secured, and which roles are allowed to access the secured servlets. You must additionally define the security domain that will handle the authentication and authorization for the war. This is done through the jboss-web.xml descriptor, and an example that uses the “http-invoker” security domain is given in .

An example jboss-web.xml descriptor to assign the

```
<jboss-web>
    <security-domain>java:/jaas/http-invoker</security-domain>
</jboss-web>
```

The security-domain element defines the name of the security domain that will be used for the JAAS login module configuration used for authentication and authorization. See the security chapter of the JBoss Administration and Development book for additional details on the meaning and configuration of the security domain name.

Securing Access to JNDI with a Read-Only Unsecured Context

Another feature available for the JNDI/HTTP naming service is the ability to define a context that can be accessed by unauthenticated users in read-only mode. This can be important for services used by the authentication layer. For example, the SRPLoginModule needs to lookup the SRP server interface used to perform authentication. To enable this, some additional web.xml descriptor settings are needed. Listing 3-17 shows the necessary elements.

LISTING 3-17. The additional web.xml descriptor elements needed for read-only access

```
<web-app>
    <filter>
        <filter-name>ReadOnlyAccessFilter</filter-name>
        <filter-class>org.jboss.invocation.http.servlet.ReadOnlyAccessFilter</
filter-class>
        <init-param>
            <param-name>readOnlyContext</param-name>
            <param-value>readonly-context</param-value>
        </init-param>
        <init-param>
            <param-name>invokerName</param-name>
            <param-value>jboss:service=Naming</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>ReadOnlyAccessFilter</filter-name>
        <url-pattern>/readonly/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>ReadOnlyJNDIFactory</servlet-name>
```

```
<servlet-class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-class>
<init-param>
    <param-name>invokerName</param-name>
    <param-value>jboss:service=Naming</param-value>
</init-param>
<init-param>
    <param-name>externalURL</param-name>
    <param-value>http://localhost:8080/invoker/readonly/JMXInvokerServlet</param-value>
</init-param>
<load-on-startup>2</load-on-startup>
</servlet>

<!-- A mapping for the JMXInvokerServlet that only allows invocations
of lookups under a read-only context. This is enforced by the
ReadOnlyAccessFilter
-->
<servlet-mapping>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <url-pattern>/readonly/JMXInvokerServlet/*</url-pattern>
</servlet-mapping>
```

With these settings, one may perform Context.lookup operations on the “readonly-context” or its sub-contexts, but no other operations on this context. Also, no operations of any kind may be performed on other contexts. Here is a code fragment for a lookup of the “readonly-context/data” binding:

```
Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
    "http://localhost:8080/invoker/ReadOnlyJNDIFactory");
Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly-context/data");
```

Additional Naming MBeans

In addition to the NamingService MBean that configures an embedded JBossNS server within JBoss, there are three additional MBean services related to naming that ship with JBoss. They are the ExternalContext, NamingAlias, and JNDIView.

org.jboss.naming.ExternalContext MBean

The ExternalContext MBean allows you to federate external JNDI contexts into the JBoss server JNDI namespace. The term external refers to any naming service external to the JBossNS naming service running inside of the JBoss server VM. You can incorporate LDAP servers, file systems, DNS servers, and so on, even if the JNDI provider root context is not serializable. The federation can be made available to remote clients if the naming service supports remote access.

To incorporate an external JNDI naming service, you have to add a configuration of the ExternalContext MBean service to the jboss.jcml configuration file. The configurable attributes of the ExternalContext service are as follows:

- **JndiName**—The JNDI name under which the external context is to be bound.
- **RemoteAccess**—A boolean flag indicating if the external InitialContext should be bound using a Serializable form that allows a remote client to create the external InitialContext. When a remote client looks up the external context via the JBoss JNDI InitialContext, they effectively create an instance of the external InitialContext using the same env properties passed to the ExternalContext MBean. This will only work if the client could do a 'new InitialContext(env)' remotely. This requires that the Context.PROVIDER_URL value of env is resolvable in the remote VM that is accessing the context. This should work for the LDAP example. For the file system example this most likely won't work unless the file system path refers to a common network path. If this property is not given it defaults to false.
- **CacheContext**—The cacheContext flag. When set to true, the external Context is only created when the MBean is started and then stored as an in memory object until the MBean is stopped. If cacheContext is set to false, the external Context is created on each lookup using the MBean properties and InitialContext class. When the uncached Context is looked up by a client, the client should invoke close() on the Context to prevent resource leaks.
- **InitialContext**—The fully qualified class name of the InitialContext implementation to use. Must be one of: javax.naming.InitialContext, javax.naming.directory.InitialDirContext or javax.naming.ldap.InitialLdapContext. In the case of the InitialLdapContext, a null Controls array is used. The default is javax.naming.InitialContext.
- **Properties**—Set the jndi.properties information for the external InitialContext. This is either a URL string or a classpath resource name. Examples are as follows:
 - file:///config/myldap.properties
 - http://config.mycompany.com/myldap.properties
 - /conf/myldap.properties
 - myldap.properties

The jboss.jcml fragment shown in Listing 3-18 shows two configurations—one for an LDAP server, and the other for a local file system directory.

LISTING 3-18. ExternalContext MBean configurations

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"
       name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/jboss">
    <attribute name="JndiName">external/ldap/jboss</attribute>
    <attribute name="Properties">jboss.ldap</attribute>
    <attribute name="InitialContext">
        javax.naming.ldap.InitialLdapContext
    </attribute>
    <attribute name="RemoteAccess">true</attribute>
</mbean>
<!-- Bind the /usr/local file system directory -->
<mbean code="org.jboss.naming.ExternalContext"
```

```
name="jboss.jndi:service=ExternalContext,jndiName=external/fs/usr/local" >
<attribute name="JndiName">external/fs/usr/local</attribute>
<attribute name="Properties">local.props</attribute>
<attribute name="InitialContext">javax.naming.InitialContext</attribute>
</mbean>
```

The first configuration describes binding an external LDAP context into the JBoss JNDI namespace under the name “external/ldap/jboss”. An example jboss.ldap properties file is as follows:

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://ldaphost.jboss.org:389/o=jboss.org
java.naming.security.principal=cn=Directory Manager
java.naming.security.authentication=simple
java.naming.security.credentials=secret
```

With this configuration, you can access the external LDAP context located at ldap://ldaphost.jboss.org:389/o=jboss.org from within the JBoss VM using the following code fragment:

```
InitialContext iniCtx = new InitialContext();
LdapContext ldapCtx = iniCtx.lookup("external/ldap/jboss");
```

Using the same code fragment outside of the JBoss server VM will work in this case because the RemoteAccess property was set to true. If it were set to false, it would not work because the remote client would receive a Reference object with an ObjectFactory that would not be able to recreate the external InitialContext.

The second configuration describes binding a local file system directory /usr/local into the JBoss JNDI namespace under the name “external/fs/usr/local”. An example local.props properties file is:

```
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
java.naming.provider.url=file:///usr/local
```

With this configuration, you can access the external file system context located at file:///usr/local from within the JBoss VM using the following code fragment:

```
InitialContext iniCtx = new InitialContext();
Context ldapCtx = iniCtx.lookup("external/fs/usr/local");
```

The org.jboss.naming.NamingAlias MBean

The NamingAlias MBean is a simple utility service that allows you to create an alias in the form of a JNDI javax.naming.LinkRef from one JNDI name to another. This is similar to a symbolic link in the Unix file system. To an alias you add a configuration of the NamingAlias MBean to the jboss.jcml configuration file. The configurable attributes of the NamingAlias service are as follows:

- **FromName**, The location where the LinkRef is bound under JNDI.
- **ToName**, The to name of the alias. This is the target name to which the LinkRef refers. The name is a URL, or a name to be resolved relative to the InitialContext, or if the first character of the name is „, the name is relative to the context in which the link is bound.

An example that provides a mapping of the JNDI name “QueueConnectionFactory” to the name “ConnectionFactory” file is as follows:

```
<mbean code="org.jboss.naming.NamingAlias"
name="jboss.mq:service=NamingAlias,fromName=QueueConnectionFactory">
<attribute name="ToName">ConnectionFactory</attribute>
<attribute name="FromName">QueueConnectionFactory</attribute>
</mbean>
```

The org.jboss.naming.JNDIView MBean

The JNDIView MBean allows the user to view the JNDI namespace tree as it exists in the JBoss server using the JMX agent view interface. All that is required to use the JNDIView service is to add a configuration to jboss.jcml file. The JNDIView service has no configurable attributes, and so a suitable configuration is:

```
<mbean code="org.jboss.naming.JNDIView" name="jboss:service=JNDIView"/>
```

To view the JBoss JNDI namespace using the JNDIView MBean, you connect to the JMX Agent View using the http interface. The default settings put this at <http://localhost:8080/jmx-console/>. On this page you will see a section that lists the registered MBeans by domain. It should look something like that shown in Figure 3-7, The HTTP JMX agent view of the configured JBoss MBeans., where the JNDIView MBean is under the mouse cursor.

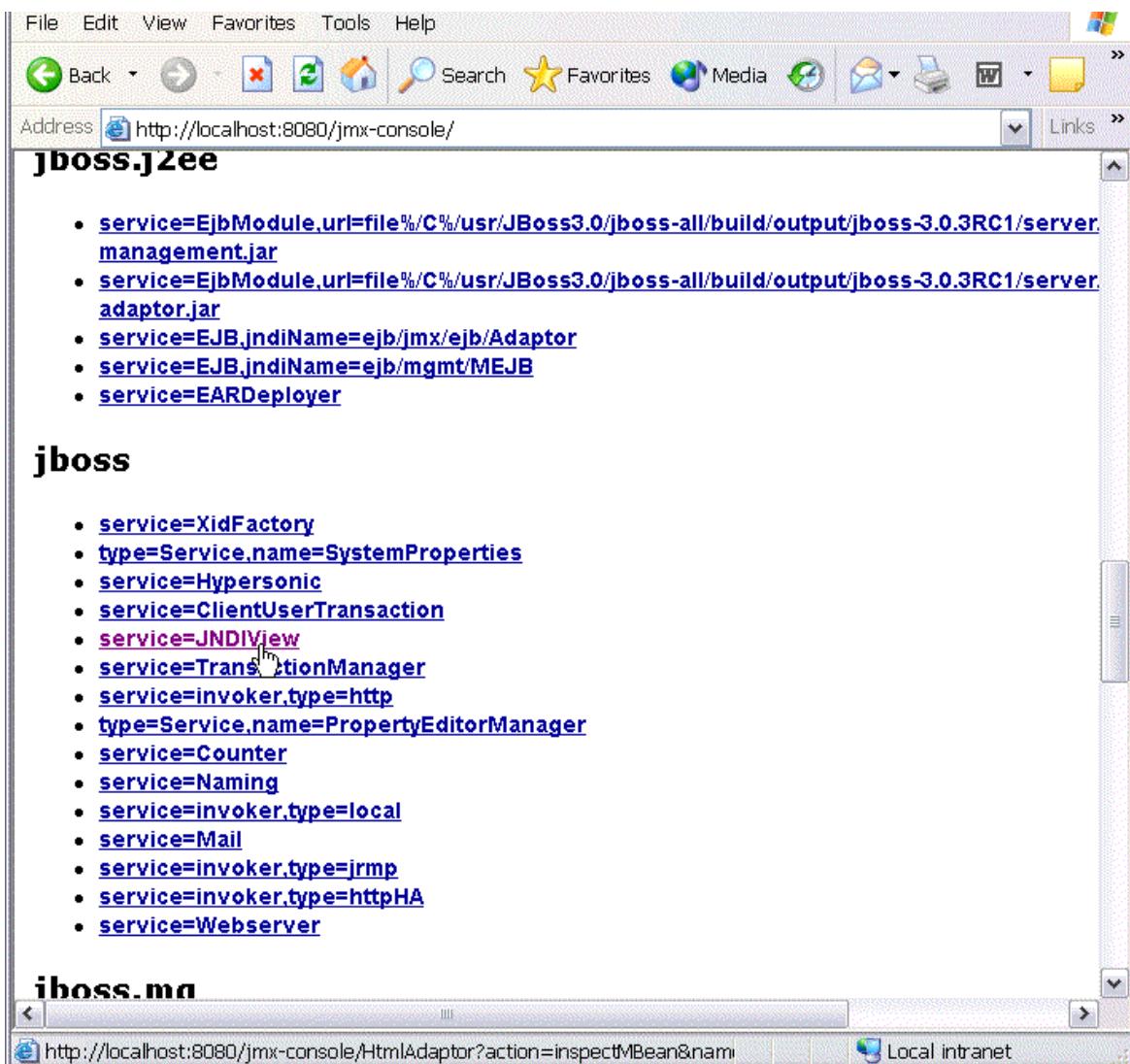


FIGURE 3-7. The HTTP JMX agent view of the configured JBoss MBeans.

Selecting the [JNDIView](#) link takes you to the [JNDIView](#) MBean view, which will have a list of the [JNDIView](#) MBean operations. This view should look similar to that shown in Figure 3-8, The HTTP JMX MBean view of the JNDIView MBean..

The screenshot shows a web browser window with the address bar containing `http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss%3Aservice`. The main content area displays the 'MBean description:' section, which includes a table of attributes:

Name	Type	Access	Value
Name	java.lang.String	R	JNDIView
State	int	R	3
StateString	java.lang.String	R	Started

Below this is the 'List of MBean operations:' section, which lists a single operation:

`java.lang.String list()`

Underneath the operation name is a table for parameters:

Param	ParamType	ParamValue
arg0	boolean	<input checked="" type="radio"/> True <input type="radio"/> False

A yellow box highlights the 'Invoke' button at the bottom left of the parameter table.

FIGURE 3-8. The HTTP JMX MBean view of the JNDIView MBean.

The list operation dumps out the JBoss server JNDI namespace as an html page using a simple text view. As an example, invoking the list operation for the default JBoss-3.0.1 distribution server produced the view shown in Figure 3-9 .

The screenshot shows a web browser window displaying the JBoss JMX console at <http://localhost:8080/jmx-console/HtmlAdaptor>. The main title is "java: Namespace". Below it, a tree view lists various JNDI entries:

- +-- DefaultDS (class: org.jboss.resource.adapter.jdbc.local.LocalDataSource)
- +-- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
- +-- SecurityProxyFactory (class: org.jboss.security.SubjectSecurityProxyFactory)
- +-- DefaultJMSProvider (class: org.jboss.jms.jndi.JBossMQProvider)
- +-- CounterService (class: org.jboss.varia.counter.CounterService)
- +-- comp (class: javax.naming.Context)
- +-- JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
- +-- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
- +-- jaas (class: javax.naming.Context)
 - | +- JmsXAResult (class: org.jboss.security.plugins.SecurityDomainContext)
 - | +- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
 - | +- http-invoker (class: org.jboss.security.plugins.SecurityDomainContext)
- +-- timedCacheFactory (class: javax.naming.Context)

Failed to lookup: timedCacheFactory, errormsg=org.jboss.util.TimedCachePolicy

- +-- TransactionPropagationContextExporter (class: org.jboss.tm.TransactionPropagationContextExporter)
- +-- Mail (class: javax.mail.Session)
- +-- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)
- +-- TransactionPropagationContextImporter (class: org.jboss.tm.TransactionPropagationContextImporter)
- +-- TransactionManager (class: org.jboss.tm.TxManager)

Global JNDI Namespace

- +-- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
- +-- UserTransactionSessionFactory (class: org.jboss.tm.usertx.server.UserTransactionSessionFactory)
- +-- RMIXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
- +-- topic (class: org.jnp.interfaces.NamingContext)
 - | +- testDurableTopic (class: org.jboss.mq.SpyTopic)
 - | +- testTopic (class: org.jboss.mq.SpyTopic)
 - | +- securedTopic (class: org.jboss.mq.SpyTopic)

FIGURE 3-9. The HTTP JMX view of the JNDIView list operation output.

Transactions on JBoss - The JTA Transaction Service

This chapter discusses transaction management in JBoss and the JBossTX architecture. The JBossTX architecture allows for any Java Transaction API (JTA) transaction manager implementation to be used. JBossTX includes a fast in-VM implementation of a JTA compatible transaction manager that is used as the default transaction manager. We will first provide an overview of the key transaction concepts and notions in the JTA to provide sufficient background for the JBossTX architecture discussion. We will then discuss the interfaces that make up the JBossTX architecture and conclude with a discussion of the MBeans available for integration of alternate transaction managers.

Transaction/JTA Overview

For the purpose of this discussion, we can define a transaction as a unit of work containing one or more operations involving one or more shared resources having ACID properties. ACID is an acronym for Atomicity, Consistency, Isolation and Durability, the four important properties of transactions. The meanings of these terms is:

- **Atomicity:** A transaction must be atomic. This means that either all the work done in the transaction must be performed, or none of it must be performed. Doing part of a transaction is not allowed.
- **Consistency:** When a transaction is completed, the system must be in a stable and consistent condition.
- **Isolation:** Different transactions must be isolated from each other. This means that the partial work done in one transaction is not visible to other transactions until the transaction is committed, and that each process in a multi-user system can be programmed as if it was the only process accessing the system.
- **Durability:** The changes made during a transaction are made persistent when it is committed. When a transaction is committed, its changes will not be lost, even if the server crashes afterwards.

To illustrate these concepts, consider a simple banking account application. The banking application has a database with a number of accounts. The sum of the amounts of all accounts must always be 0. An amount of money M is moved from account A to account B by subtracting M from account A and adding M to account B. This operation must be done in a transaction, and all four ACID properties are important.

The atomicity property means that both the withdrawal and deposit is performed as an indivisible unit. If, for some reason, both cannot be done nothing will be done.

The consistency property means that after the transaction, the sum of the amounts of all accounts must still be 0.

The isolation property is important when more than one bank clerk uses the system at the same time. A withdrawal or deposit could be implemented as a three-step process: First the amount of the account is read from the database; then something is subtracted from or added to the amount read from the database; and at last the new amount is written to the database. Without transaction isolation several bad things could happen. For example, if two processes read the amount of account A at the same time, and each independently added or subtracted something before writing the new amount to the database, the first change would be incorrectly overwritten by the last.

The durability property is also important. If a money transfer transaction is committed, the bank must trust that some subsequent failure cannot undo the money transfer.

Pessimistic and optimistic locking

Transactional isolation is usually implemented by locking whatever is accessed in a transaction. There are two different approaches to transactional locking: Pessimistic locking and optimistic locking.

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time. If most transactions simply look at the resource and never change it, an exclusive lock may be overkill as it may cause lock contention, and optimistic locking may be a better approach. With pessimistic locking, locks are applied in a fail-safe way. In the banking application example, an account is locked as soon as it is accessed in a transaction. Attempts to use the account in other transactions while it is locked will either result in the other process being delayed until the account lock is released, or that the process transaction will be rolled back. The lock exists until the transaction has either been committed or rolled back.

With optimistic locking, a resource is not actually locked when it is first accessed by a transaction. Instead, the state of the resource at the time when it would have been locked with the pessimistic locking approach is saved. Other transactions are able to concurrently access to the resource and the possibility of conflicting changes is possible. At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back.

In the banking application example, the amount of an account is saved when the account is first accessed in a transaction. If the transaction changes the account amount, the amount is read from the store again just before the amount is about to be updated. If the amount has changed since the transaction began, the transaction will fail itself, otherwise the new amount is written to persistent storage.

The components of a distributed transaction

There are a number of participants in a distributed transaction. These include:

- Transaction Manager: This component is distributed across the transactional system. It manages and coordinates the work involved in the transaction. The transaction manager is exposed by the javax.transaction.TransactionManager interface in JTA.
- Transaction Context: A transaction context identifies a particular transaction. In JTA the corresponding interface is javax.transaction.Transaction.
- Transactional Client: A transactional client can invoke operations on one or more transactional objects in a single transaction. The transactional client that started the transaction is called the transaction originator. A transaction client is either an explicit or implicit user of JTA interfaces and has no interface representation in the JTA.
- Transactional Object: A transactional object is an object whose behavior is affected by operations performed on it within a transactional context. A transactional object can also be a transactional client. Most Enterprise Java Beans are transactional objects.
- Recoverable Resource: A recoverable resource is a transactional object whose state is saved to stable storage if the transaction is committed, and whose state can be reset to what it was at the beginning of the transaction if the transaction is rolled back. At commit time, the transaction manager uses the two-phase XA protocol when communicating with the recoverable resource to ensure transactional integrity when more than one recoverable resource is involved in the transaction being committed. Transactional databases and message brokers like JBossMQ are examples of recoverable resources. A recoverable resource is represented using the javax.transaction.xa.XAResource interface in JTA.

The two-phase XA protocol

When a transaction is about to be committed, it is the responsibility of the transaction manager to ensure that either all of it is committed, or that all of it is rolled back. If only a single recoverable resource is involved in the transaction, the task of the transaction manager is simple: It just has to tell the resource to commit the changes to stable storage.

When more than one recoverable resource is involved in the transaction, management of the commit gets more complicated. Simply asking each of the recoverable resources to commit changes to stable storage is not enough to maintain the atomic property of the transaction. The reason for this is that if one recoverable resource has committed and another fails to commit, part of the transaction would be committed and the other part rolled back.

To get around this problem, the two-phase XA protocol is used. The XA protocol involves an extra prepare phase before the actual commit phase. Before asking any of the recoverable resources to

commit the changes, the transaction manager asks all the recoverable resources to prepare to commit. When a recoverable resource indicates it is prepared to commit the transaction, it has ensured that it can commit the transaction. The resource is still able to rollback the transaction if necessary as well.

So the first phase consists of the transaction manager asking all the recoverable resources to prepare to commit. If any of the recoverable resources fails to prepare, the transaction will be rolled back. But if all recoverable resources indicate they were able to prepare to commit, the second phase of the XA protocol begins. This consists of the transaction manager asking all the recoverable resources to commit the transaction. Because all the recoverable resources have indicated they are prepared, this step cannot fail.

Heuristic exceptions

In a distributed environment communications failures can happen. If communication between the transaction manager and a recoverable resource is not possible for an extended period of time, the recoverable resource may decide to unilaterally commit or rollback changes done in the context of a transaction. Such a decision is called a heuristic decision. It is one of the worst errors that may happen in a transaction system, as it can lead to parts of the transaction being committed while other parts are rolled back, thus violating the atomicity property of transaction and possibly leading to data integrity corruption.

Because of the dangers of heuristic exceptions, a recoverable resource that makes a heuristic decision is required to maintain all information about the decision in stable storage until the transaction manager tells it to forget about the heuristic decision. The actual data about the heuristic decision that is saved in stable storage depends on the type of recoverable resource and is not standardized. The idea is that a system manager can look at the data, and possibly edit the resource to correct any data integrity problems.

There are several different kinds of heuristic exceptions defined by the JTA. The [javax.transaction.HeuristicCommitException](#) is thrown when a recoverable resource is asked to rollback to report that a heuristic decision was made and that all relevant updates have been committed. On the opposite end is the [javax.transaction.HeuristicRollbackException](#), which is thrown by a recoverable resource when it is asked to commit to indicate that a heuristic decision was made and that all relevant updates have been rolled back.

The [javax.transaction.HeuristicMixedException](#) is the worst heuristic exception. It is thrown to indicate that parts of the transaction were committed, while other parts were rolled back. The transaction manager throws this exception when some recoverable resources did a heuristic commit, while other recoverable resources did a heuristic rollback.

Transaction IDs and branches

In JTA, the identity of transactions is encapsulated in objects implementing the [javax.transaction.xa.Xid](#) interface. The transaction ID is an aggregate of three parts:

- The format identifier indicates the transaction family and tells how the other two parts should be interpreted.
- The global transaction id identified the global transaction within the transaction family.
- The branch qualifier denotes a particular branch of the global transaction.

Transaction branches are used to identify different parts of the same global transaction. Whenever the transaction manager involves a new recoverable resource in a transaction it creates a new transaction branch.

JBoss Transaction Internals

The JBoss application server is written to be independent of the actual transaction manager used. JBoss uses the JTA [javax.transaction.TransactionManager](#) interface as its view of the server transaction manager. Thus, JBoss may use any transaction manager which implements the JTA Transaction-Manager interface. Whenever a transaction manager is used it is obtained from the well-known JNDI location "java:/TransactionManager". This is the globally available access point for the server transaction manager.

If transaction contexts are to be propagated with RMI/JRMP calls, the transaction manager must also implement two simple interfaces for the import and export of transaction propagation contexts (TPCs). The interfaces are [org.jboss.tm.TransactionPropagationContextImporter](#), and [org.jboss.tm.TransactionPropagationContextFactory](#).

Being independent of the actual transaction manager used also means that JBoss does not specify the format or type of the transaction propagation contexts used. In JBoss, a TPC is of type [Object](#), and the only requirement is that the TPC must implement the [java.io.Serializable](#) interface.

When using the RMI/JRMP protocol for remote calls, the TPC is carried as a field in the [org.jboss.ejb.plugins.jrmp.client.RemoteMethodInvocation](#) class that is used to forward remote method invocation requests.

Adapting a Transaction Manager to JBoss

A transaction manager has to implement the Java Transaction API to be easily integrated with JBoss. As almost everything in JBoss, the transaction manager is managed as an MBean. Like all JBoss services it should implement [org.jboss.system.ServiceMBean](#) to ensure proper life-cycle management.

The primary requirement of the transaction manager service on startup is that it binds its implementation of the three required interfaces into JNDI. These interfaces and their JNDI locations are:

- The [javax.transaction.TransactionManager](#) interface. This interface is used by the application server to manage transactions on behalf of the transactional objects that use container managed transactions. It must be bound under the JNDI name “java:/TransactionManager”.
- The transaction propagation context factory interface [org.jboss.tm.TransactionPropagationContextFactory](#). It is called by JBoss whenever a transaction propagation context is needed for transporting a transaction with a remote method call. It must be bound under the JNDI name "java:/TransactionPropagationContextImporter".
- The transaction propagation context importer interface [org.jboss.tm.TransactionPropagationContextImporter](#). This interface is called by JBoss whenever a transaction propagation context from an incoming remote method invocation has to be converted to a transaction that can be used within the receiving JBoss server VM.

Establishing these JNDI bindings is all the transaction manager service needs to do to install its implementation as the JBoss server transaction manager.

The Default Transaction Manager

JBoss is by default configured to use the fast in-VM transaction manager. This transaction manager is very fast, but does have two limitations:

- It does not do transactional logging, and is thus incapable of automated recovery after a server crash.
- While it does support propagating transaction contexts with remote calls, it does not support propagating transaction contexts to other virtual machines, so all transactional work must be done in the same virtual machine as the JBoss server.

The corresponding default transaction manager MBean service is the [org.jboss.tm.TransactionManagerService](#) MBean. It has two configurable attributes:

- **TransactionTimeout:** The default transaction timeout in seconds. The default value is 300 seconds or 5 minutes.
- **XidFactory:** The JMX [ObjectName](#) of the MBean service that provides the [org.jboss.tm.XidFactoryMBean](#) implementation. The [XidFactoryMBean](#) interface is used to create [javax.transaction.xa.Xid](#) instances. This is a workaround for XA JDBC drivers that only work with their own Xid implementation. Examples of such drivers are the older Oracle XA drivers. If not specified a JBoss implementation of the [Xid](#) interface is used.

[org.jboss.tm.XidFactory](#)

The [XidFactory](#) MBean is a factory for [javax.transaction.xa.Xid](#) instances in the form of [org.jboss.tm.XidImpl](#). The [XidFactory](#) allows for customization of the [XidImpl](#) that it constructs through the following attributes:

- **BaseGlobalId:** This is used for building globally unique transaction identifiers. This must be set individually if multiple jboss instances are running on the same machine. The default value is the host name of the JBoss server, followed by a slash.
- **GlobalIdNumber:** A long value used as initial transaction id. The default is 0.

- **Pad:** The pad value determines whether the byte[] returned by the Xid getGlobalTransactionId and getBranchQualifier methods should be equal to maximum 64 byte length or a variable value <= 64. Some resource managers(Oracle for example) require ids that are max length in size.

UserTransaction Support

The JTA javax.transaction.UserTransaction interface allows applications to explicitly control transactions. For enterprise session beans that manage transaction themselves (BMT), a UserTransaction can be obtained by calling the getUserTransaction method on the bean context object, javax.ejb.SessionContext.

Note: For BMT beans, do not obtain the UserTransaction interface using a JNDI lookup. Doing this violates the EJB specification, and the returned UserTransaction object does not have the hooks the EJB container needs to make important checks.

To use the UserTransaction interface in other places, the org.jboss.tm.usertx.server.ClientUserTransactionService MBean must be configured and started. This MBean publishes a UserTransaction implementation under the JNDI name "UserTransaction". This MBean is configured by default in the standard JBoss distributions and has no configurable attributes.

When the UserTransaction is obtained with a JNDI lookup from a stand-alone client (ie. a client operating in a virtual machine than the server's), a very simple UserTransaction suitable for thin clients is returned. This UserTransaction implementation only controls the transactions on the server the UserTransaction object was obtained from. Local transactional work done in the client is not done within the transactions started by this UserTransaction object.

When a UserTransaction object is obtained by looking up JNDI name "UserTransaction" in the same virtual machine as JBoss, a simple interface to the JTA TransactionManager is returned. This is suitable for web components running in web containers embedded in JBoss. When components are deployed in an embedded web server, the deployer will make a JNDI link from the standard "java:comp/UserTransaction" ENC name to the global "UserTransaction" binding so that the web components can lookup the UserTransaction instance under JNDI name as specified by the J2EE.

EJBs on JBoss - The EJB Container Configuration and Architecture

The JBoss 3.0 EJB container architecture is a fourth generation design that emphasizes a modular plug-in approach. All key aspects of the EJB container may be replaced by custom versions of a plug-in by a developer. This approach allows for fine tuned customization of the EJB container behavior to optimally suite your needs. Most of the EJB container behavior is configurable through the EJB jar META-INF/jboss.xml descriptor and the default server-wide equivalent standardjboss.xml descriptor. We will look at various configuration capabilities throughout this chapter as we explore the container architecture.

The EJB Client Side View

We will begin our tour of the EJB container by looking at the client view of an EJB through the home and remote proxies. It is the responsibility of the container provider to generate the [javax.ejb.EJBHome](#) and [javax.ejb.EJBObject](#) for an EJB implementation. A client never references an EJB bean instance directly, but rather references the [EJBHome](#) which implements the bean home interface, and the [EJBObject](#) which implements the bean remote interface.

Figure 5-1 shows the composition of an EJB home proxy and its relation to the EJB deployment.

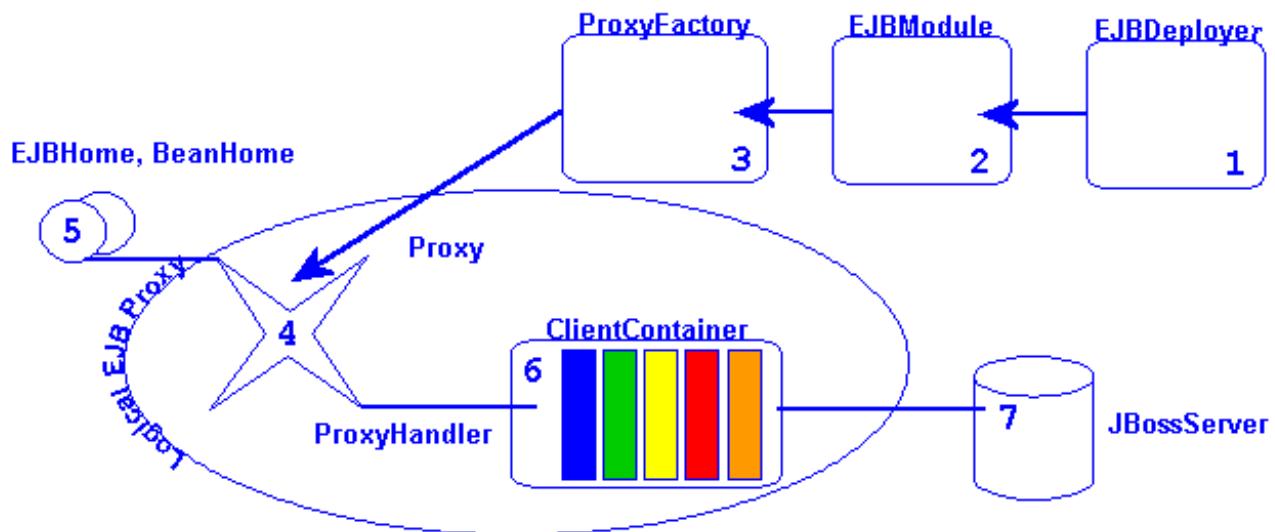


FIGURE 5-1. The composition of an EJBHome proxy in JBoss.

The numbered items in the figure are:

1. The EJBDeployer (`org.jboss.ejb.EJBDeployer`) is invoked to deploy an EJB jar. For each EJB defined in the deployment an EJBModule (`org.jboss.ejb.EJBModule`) is created to encapsulate the deployment metadata.
2. The create phase of the EJBModule life cycle creates a ProxyFactory (`org.jboss.proxy.ejb.ProxyFactory`) that manages the creation of EJB home and remote interface proxies based on the EJBModule metadata.
3. The ProxyFactory constructs the logical proxies and binds the homes into JNDI. A logical proxy is composed of a dynamic Proxy (`java.lang.reflect.Proxy`), the home interfaces of the EJB that the Proxy exposes and the ProxyHandler (`java.lang.reflect.InvocationHandler`) implementation in the form of the ClientContainer.
4. The Proxy created by the ProxyFactory is a JDK 1.3+ dynamic proxy. It is a serializable object that proxies the EJB home interfaces as defined in the EJBModule metadata. The Proxy translates requests made through the strongly typed home interfaces into a detyped invocation using the ClientContainer handler associated with the Proxy. It is the dynamic Proxy instance that is bound into JNDI as the EJB home interface that clients lookup. When a client does a lookup of an EJB home, the home Proxy is transported into the client VM along with the ProxyHandler. The use of dynamic proxies avoids the EJB specific compilation step required by many other EJB containers.
5. The EJB home interface is declared in the ejb-jar.xml descriptor and available from the EJBModule metadata. A key property of dynamic proxies is that they are seen to implement the interfaces they expose. This is true in the sense of Java's strongly typed system. A proxy can be cast to any of the home interfaces and reflection on the proxy provides the full details of the interfaces it proxies.
6. The Proxy delegates calls made through any of its interfaces to its ProxyHandler. The single method required of the handler is:

```
public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
```

The ProxyFactory creates a ClientContainer ([org.jboss.proxy.ClientContainer](#)) and assigns this as the ProxyHandler. The ClientContainer's state consists of an [InvocationContext](#) ([org.jboss.invocation.InvocationContext](#)) and a chain of interceptors ([org.jboss.proxy.Interceptor](#)). The InvocationContext contains:

- the JMX ObjectName of the EJB container MBean the Proxy is associated with
- the [javax.ejb.EJBMetaData](#) for the EJB
- the JNDI name of the EJB home interface
- the transport specific invoker ([org.jboss.invocation.Invoker](#))
- The interceptor chain consists of the functional units that make up the EJB home or remote interface behavior. This is a configurable aspect of an EJB as we will see when we discuss the jboss.xml descriptor, and the interceptor makeup is contained in the EJBModule metadata. Interceptors ([org.jboss.proxy.Interceptor](#)) handle the different EJB types, security, transactions and transport. You can add your own interceptors as well.

The configuration of the client side interceptors is done using the jboss.xml client-interceptors element. Figure 5-2 shows the subset of the jboss.xml DTD for the client interceptors. When the ClientContainer invoke method is called it creates an untyped Invocation ([org.jboss.invocation.Invocation](#)) to encapsulate request. This is then passed through the interceptor chain. The last interceptor in the chain will be the transport handler that knows how to get the request to the server and obtain the reply, taking care of the transport specific details.

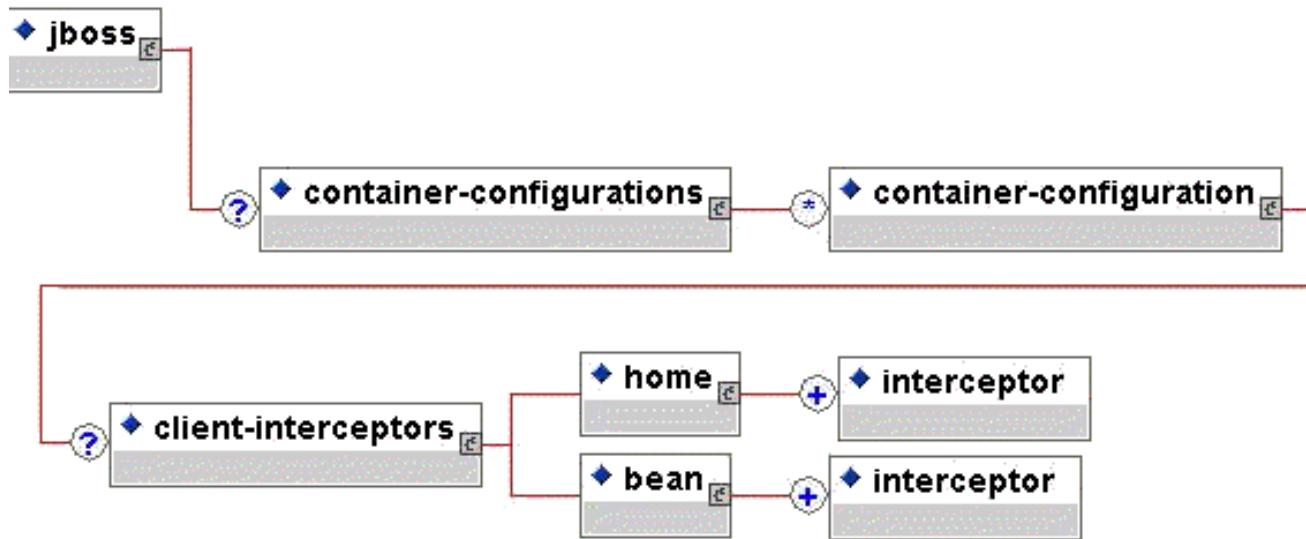


FIGURE 5-2. The jboss.xml descriptor client side interceptor configuration elements.

As an example of the client interceptor configuration usage, consider the default stateless session bean configuration found in the server/default/standardjboss.xml descriptor. Listing 5-1 shows the client-interceptors configuration for the “Standard Stateless SessionBean”.

LISTING 5-1. The client-interceptors from the “Standard Stateless SessionBean” configuration.

```
<container-configuration>
  <container-name>Standard Stateless SessionBean</container-name>
  <call-logging>false</call-logging>
  <container-invoker>org.jboss.proxy.ejb.ProxyFactory</container-invoker>
  <container-interceptors>
    ...
  </container-interceptors>
  <client-interceptors>
    <home>
      <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </home>
    <bean>
      <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </bean>
  </client-interceptors>
  ...
</container-configuration>
```

This is the client interceptor configuration for stateless session beans that is used in the absence of an EJB jar META-INF/jboss.xml configuration that overrides these settings. The functionality provided by each interceptor is:

- org.jboss.proxy.ejb.HomeInterceptor, this handles the `toString`, `equals`, `hashCode`, `getHomeHandle`, `getEJBMetaData`, and `remove` methods of the `EJBHome` interface locally in the client VM. Any other methods are propagated to the next interceptor.
- org.jboss.proxy.ejb.StatelessSessionInterceptor, this handles the `toString`, `equals`, `hashCode`, `getHandle`, `getEJBHome` and `isIdentical` methods of the `EJBObject` interface locally in the client VM. Any other methods are propagated to the next interceptor.
- org.jboss.proxy.SecurityInterceptor, this associates the current SecurityAssociation context with the method invocation for use by other interceptors or the server.
- org.jboss.proxy.TransactionInterceptor, this associates any active transaction with the invocation method `invocation` for use by other interceptors.
- org.jboss.invocation.InvokerInterceptor, this interceptor encapsulates the dispatch of the method invocation to the transport specific invoker. It knows if the client is executing in the same VM as the server and will optimally route the invocation to a by reference invoker in this situation. When the client is external to the server VM this interceptor delegates the invocation to the transport invoker associated with the invocation context.

The EJB Server Side View

Every EJB invocation must end up at a JBoss server hosted EJB container. In this section we will look at how invocations are transported to the JBoss server VM and find their way to the EJB container via the JMX bus.

Detached Invokers - The Transport Middlemen

We have seen that the last point of contact with the client architecture is an Invoker instance which is an implementation of the [org.jboss.invocation.Invoker](#) interface. This is a trivial interface, the signature of which is given in Listing 5-2.

LISTING 5-2. The org.jboss.invocation.Invoker interface

```
package org.jboss.invocation;

import java.rmi.Remote;
import org.jboss.proxy.Interceptor;

public interface Invoker extends Remote
{
    /** A free form String identifier for this delegate invoker, can be clustered
     * or target node. This should evolve in a more advanced meta-inf object
     */
    public String getServerHostName() throws Exception;

    /** The invoke with an Invocation Object the delegate can handle network
     * protocols
     * on behalf of proxies (proxies delegate to these puppies)
     *
     * @param invocation A pointer to the invocation object
     * @return the value of method invocation.
     *
     * @throws Exception any invoke method exception.
     */
    public Object invoke(Invocation invocation) throws Exception;
}
```

Although the interface is an RMI [Remote](#) interface, this has no implications with regard to requiring the use of RMI or any specific RMI based protocol. An [Invoker](#) simply represents a detyped transport handler that is compatible with RMI. JBoss 3.0 provides invokers for in VM call-by-reference, RMI/JRMP, and RMI/IOP transports. The generic view of the invoker architecture is presented in Figure 5-3.

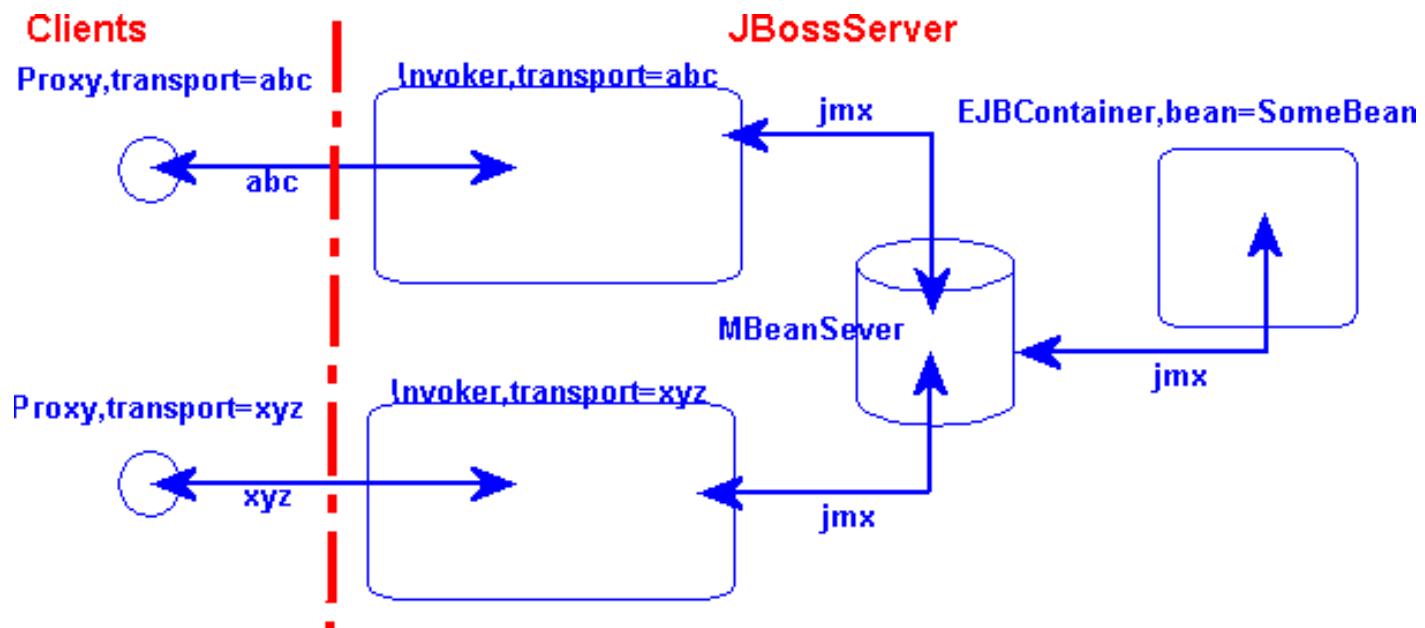


FIGURE 5-3. The transport invoker server side architecture

For each type of home proxy there is a binding to an invoker and its associated transport protocol. In 3.2+ a container will be able to have multiple invocation protocols active simultaneously. In 3.0.x a container can only have a single protocol associated with it, but this can be any protocol. The jboss.xml DTD configuration fragment for the container invoker configuration is given in Figure 5-4. The bean-invoker and home-invoker values are the JMX ObjectNames of the MBean that provides the Invoker implementation to use with the remote and home client interceptor chains respectively. These should be set to the same value as it makes little sense in general to have separate invokers for the remote and home interface methods.

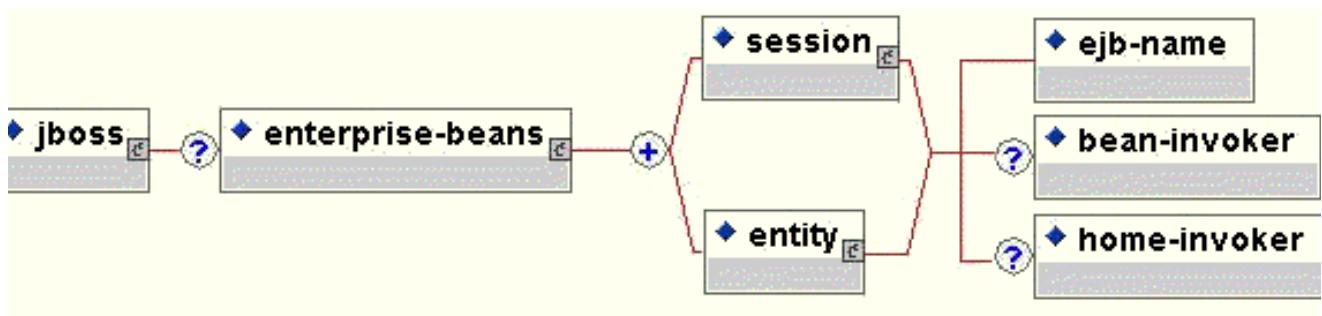


FIGURE 5-4. The jboss.xml descriptor container invoker configuration elements.

The LocalInvoker - In VM transport

org.jboss.invocation.local.LocalInvoker is an MBean service that provides an in VM call-by-reference implementation of the Invoker interface that has no configurable attributes. When it receives an

invocation it simply looks up the address of the target EJB container in the form of its JMX Object-Name and then sends the invocation to the EJB container via the MBeanServer.

The JRMPInvoker - RMI/JRMP Transport

The org.jboss.invocation.jrmp.server.JRMPInvoker class is an MBean service that provides the RMI/JRMP implementation of the Invoker interface. The JRMPInvoker exports itself as an RMI server so that when it is used as the Invoker in a remote client the JRMPInvoker stub is sent to the client instead and invocations use the RMI/JRMP protocol.

The JRMPInvoker MBean supports a number of attribute to configure the RMI/JRMP transport layer. Its configurable attributes are:

- **RMIOBJECTPORT**: sets the RMI server socket listening port number. This is the port RMI clients will connect to when communicating through the EJB home interface. The default setting in the jboss-service.xml descriptor is 4444, and if not specified, the attribute defaults to 0 to indicate an anonymous port should be used.
- **RMIClientSocketFactory**: specifies a fully qualified class name for the java.rmi.server.RMIClientSocketFactory interface to use during export of the EJB home interface.
- **RMIServerSocketFactory**: specifies a fully qualified class name for the java.rmi.server.RMIServerSocketFactory interface to use during export of the EJB home interface.
- **ServerAddress**: specifies the interface address that will be used for the RMI server socket listening port. This can be either a DNS hostname or a dot-decimal Internet address. Since the RMIServerSocketFactory does not support a method that accepts an InetAddress object, this value is passed to the RMIServerSocketFactory implementation class using reflection. A check for the existence of a:

```
public void setBindAddress(java.net.InetAddress addr)
```

method is made, and if one exists, the RMIServerSocketAddr value is passed to the RMIServerSocketFactory implementation. If the RMIServerSocketFactory implementation does not support such a method, the ServerAddress value will be ignored.

- **SecurityDomain**: specifies the JNDI name of an org.jboss.security.SecurityDomain interface implementation to associate with the RMIServerSocketFactory implementation. The value will be passed to the RMIServerSocketFactory using reflection to locate a method with a signature of:

```
public void setSecurityDomain(org.jboss.security.SecurityDomain d)
```

If no such method exists the SecurityDomain value will be ignored.

An example of using a custom JRMPInvoker MBean can be found in the org.jboss.test.jrmp package of the testsuite. Listing 5-3 illustrates the custom JRMPInvoker configuration and its mapping to a stateless session bean.

LISTING 5-3. A custom JRMPInvoker example that enables compressed sockets for session bean.

```
// The custom JRMPInvoker jboss-service.xml descriptor
<server>
  <mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
```

```
        name="jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory">
<attribute name="RMIOBJECTPORT">4445</attribute>
<attribute name="RMIClientSocketFactory">
    org.jboss.test.jrmp.ejb.CompressionClientSocketFactory
</attribute>
<attribute name="RMIServerSocketFactory">
    org.jboss.test.jrmp.ejb.CompressionServerSocketFactory
</attribute>
</mbean>
</server>
// The jboss.xml descriptor using the custom invoker
<jboss>
<enterprise-beans>
<session>
<ejb-name>StatelessSession</ejb-name>
<configuration-name>Standard Stateless SessionBean</configuration-name>
<home-invoker>
    jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory
</home-invoker>
<bean-invoker>
    jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory
</bean-invoker>
</session>
</enterprise-beans>
</jboss>
```

Here the default JRMPInvoker has been customized to bind to port 4445 and to use custom socket factories that enable compression at the transport level.

The HttpInvoker - RMI/HTTP Transport

The org.jboss.invocation.http.server.HttpInvoker MBean service provides the support for making invocations into the JMX bus over HTTP. Unlike the JRMPInvoker, the HttpInvoker is not an implementation of Invoker, but it does implement the Invoker.invoke method. The HttpInvoker is accessed indirectly by issuing an HTTP POST against the org.jboss.invocation.http.servlet.InvokerServlet. The HttpInvoker exports a client side proxy in the form of the org.jboss.invocation.http.interfaces.HttpInvokerProxy class, which is an implementation of Invoker, and is serializable. The HttpInvoker is a drop in replacement for the JRMPInvoker as the target of the bean-inovker and home-invoker EJB configuration elements. The HttpInvoker and InvokerServlet are deployed as the http-inovker.sar discussed previously in the JNDI chapter in the section entitled “Accessing JNDI over HTTP” on page 120.

The HttpInvoker supports a single attribute:

- **InvokerURL:** This is either the http URL to the InvokerServlet mapping, or the name of a system property that will be resolved inside the client VM to obtain the http URL to the InvokerServlet. This value can itself be a reference to a system property resolved in the server if the value is of the form \${x} where x is the name of the system property. This allows the URL or client side system property to be set in one place and reused in the HttpInvoker config as well as the InvokerServlet config.

An example of using the [HttpInvoker](#) to configure a stateless session bean to use the RMI/HTTP protocol can be found in the org.jboss.test.hello testsuite package. Listing 5-4 illustrates the custom settings.

LISTING 5-4. A sample jboss.xml descriptor for enabling RMI/HTTP for a stateless session bean.

```
<jboss>
  <!-- A custom container configuration for RMI/HTTP -->
  <container-configurations>
    <container-configuration extends="Standard Stateless SessionBean">
      <container-name>HTTP Stateless SessionBean</container-name>
      <home-invoker>jboss:service=invoker,type=http</home-invoker>
      <bean-invoker>jboss:service=invoker,type=http</bean-invoker>
    </container-configuration>
  </container-configurations>

  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldViaHTTP</ejb-name>
      <configuration-name>HTTP Stateless SessionBean</configuration-name>
      <jndi-name>helloworld/HelloHTTP</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

Here an extension of the standardjboss.xml “Standard Stateless SessionBean” container configuration is defined with the name “HTTP Stateless SessionBean”. It overrides the home-invoker and bean-invoker settings to use the [HttpInvoker](#) MBean. The “jboss:service=invoker,type=http” name is the default name of the [HttpInvoker](#) MBean as found in the http-inovker.sar/META-INF/jboss-service.xml descriptor.

The HA JRMPInvoker - Clustered RMI/JRMP Transport

The [org.jboss.invocation.jrmp.server.JRMPInvokerHA](#) service is an extension of the [JRMPInovker](#) that is a cluster aware invoker. As of JBoss-3.0.3, the [JRMPInvokerHA](#) fully supports all of the attributes of the [JRMPInovker](#). This means that customized bindings of the port, interface and socket transport are available to clustered RMI/JRMP as well. For additional information on the clustering architecture and the implementation of the HA RMI proxies see the JBoss Clustering docs.

The HA HttpInvoker - Clustered RMI/HTTP Transport

The RMI/HTTP layer added in JBoss-3.0.2 has been extended to allow for software load balancing of the invocations in a clustered environment in JBoss-3.0.3. An HA capable extension of the HTTP invoker has been added that borrows much of its functionality from the HA-RMI/JRMP clustering.

To enable HA-RMI/HTTP you need to configure the invokers for the EJB container. This is done through either a jboss.xml descriptor, or the standardjboss.xml descriptor. Listing 5-5 shows is an example of a stateless session configuration taken from the org.jboss.test.hello testsuite package.

LISTING 5-5. A jboss.xml stateless session configuration for HA-RMI/HTTP

```
<jboss>
  <container-configurations>
    <!-- A custom container configuration for HA-RMI/HTTP -->
    <container-configuration extends="Clustered Stateless SessionBean">
      <container-name>HA HTTP Stateless SessionBean</container-name>
      <home-invoker>jboss:service=invoker,type=httpHA</home-invoker>
      <bean-invoker>jboss:service=invoker,type=httpHA</bean-invoker>
    </container-configuration>
  </container-configurations>

  <session>
    <ejb-name>HelloWorldViaClusteredHTTP</ejb-name>
    <configuration-name>HA HTTP Stateless SessionBean</configuration-name>
    <jndi-name>helloworld/HelloHA-HTTP</jndi-name>
  </session>
</enterprise-beans>
</jboss>
```

The referenced “jboss:service=invoker,type=httpHA” invoker service is configured in the http-invoker.sar. Its default configuration from the sar descriptor is:

```
<mbean code="org.jboss.invocation.http.server.HttpInvokerHA"
       name="jboss:service=invoker,type=httpHA">
  <attribute name="InvokerURL">
    http://localhost:8080/invoker/JMXInvokerHAServlet
  </attribute>
</mbean>
```

The InvokerURL must be set to the http URL of the JMXInvokerHAServlet mapping as deployed on the cluster node. The HttpInvokerHA instances across the cluster form a collection of candidate http URLs that are made available to the client side proxy for failover and/or load balancing.

The EJB Container

An EJB container is the component that manages a particular class of EJB. In JBoss there is one instance of the org.jboss.ejb.Container created for each unique configuration of an EJB that is deployed. The actual object that is instantiated is a subclass of Container and the creation of the container instance is managed by the EJBDeployer MBean.

EJBDeployer MBean

The `org.jboss.ejb.EJBDeployer` MBean is responsible for the creation of EJB containers. Given an EJB-jar that is ready for deployment, the EJBDeployer will create and initialize the necessary EJB containers, one for each type of EJB. The configurable attributes of the EJBDeployer are:

- **VerifyDeployments**: a boolean flag indicating if the EJB verifier should be run. This validates that the EJBs in a deployment unit conform to the EJB 2.0 specification. Setting this to true is useful for ensuring your deployments are valid.
- **ValidateDTDs**: a boolean flag that indicates if the ejb-jar.xml and jboss.xml descriptors should be validated against their declared DTDs. Setting this to true is useful for ensuring your deployment descriptors are valid.
- **MetricsEnabled**: a boolean flag that controls whether container interceptors marked with an metricsEnabled=true attribute should be included in the configuration. This allows one to define a container interceptor configuration that includes metrics type interceptors that can be toggled on and off.

The deployer contains two central methods: `deploy` and `undeploy`. The `deploy` method takes a URL, which either points to an EJB-jar, or to a directory whose structure is the same as a valid EJB-jar (which is convenient for development purposes). Once a deployment has been made, it can be undeployed by calling `undeploy` on the same URL. A call to `deploy` with an already deployed URL will cause an `undeploy`, followed by deployment of the URL, such as a re-deploy. JBoss has support for full re-deployment of both implementation and interface classes, and will reload any changed classes. This will allow you to develop and update EJBs without ever stopping a running server.

During the deployment of the EJB jar the `EJBDeployer` and its associated classes perform three main functions, verify the EJBs, create a container for each unique EJB, initialize the container with the deployment configuration information. We will talk about each function in the following sections.

Verifying EJB deployments

When the **VerifyDeployments** attribute of the `EJBDeployer` is true, the deployer performs a verification of EJBs in the deployment. The verification checks that an EJB meets EJB specification compliance. This entails validating that the EJB deployment unit contains the required home and remote, local-home and local interfaces, and that the objects appearing in these interfaces are of the proper types, and that the required methods are present in the implementation class. This is a useful behavior that is enabled by default since there are a number of steps that an EJB developer and deployer must perform correctly to construct a proper EJB jar and, it is easy to make a mistake. The verification stage attempts to catch any errors and fail the deployment with an error that indicates what needs to be corrected.

Probably the most problematic aspect of writing EJBs is the fact that there is a disconnection between the bean implementation and its remote and home interfaces, as well as its deployment descriptor configuration. It is easy to have these separate elements get out of sync. One tool that helps eliminate this problem is XDoclet, an extension of the standard JavaDoc Doclet engine. It works off of custom JavaDoc tags in the EJB bean implementation class and creates the remote and home interfaces as well as the

deployment descriptors. See the XDoclet home page here <http://sourceforge.net/projects/xdoclet> for additional details.

Deploying EJBs Into Containers

The most important role performed by the EJBDeployer is the creation of an EJB container and the deployment of the EJB into the container. The deployment phase consists of iterating over EJBs in an EJB jar, and extracting the bean classes and their metadata as described by the ejb-jar.xml and jboss.xml deployment descriptors. For each EJB in the EJB jar, the following steps are performed:

1. Create subclass of org.jboss.ejb.Container depending on the type of the EJB, Stateless, Stateful, BMP Entity, CMP Entity, or MessageDriven. The container is assigned a unique ClassLoader from which it can load local resources. The uniqueness of the ClassLoader is also used to isolate the standard “java:comp” JNDI namespace from other J2EE components.
2. Set all container configurable attributes from a merge of the jboss.xml and standardjboss.xml descriptors.
3. Create and add the container interceptors as configured for the container.
4. Associate the container with an application object. This application object represents a J2EE enterprise application and may contain multiple EJBs and web contexts.

If all EJBs are successfully deployed, the application is started which in turn starts all containers and makes the EJBs available to clients. If any EJB fails to deploy, a deployment exception is thrown and the deployment module is failed.

Container configuration information

JBoss externalizes most if not all of the setup of the EJB containers using an XML file that conforms to the jboss_3_0.dtd. The section of the jboss_3_0 DTD that relates to container configuration information is shown in Figure 5-5 .

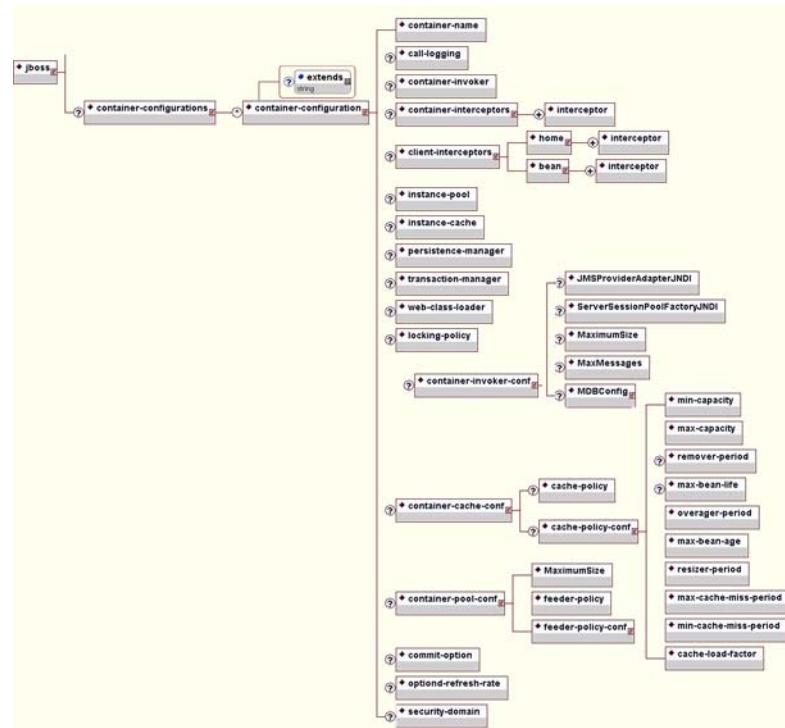


FIGURE 5-5. The `jboss_3_0` DTD elements related to container configuration.

The `container-configurations` element and its subelements specify container configuration settings for a type of container as given by the `container-name` element. Each configuration specifies information such as container invoker type, the container interceptor makeup, instance caches/pools and their sizes, persistence manager, security, and so on. Because this is a large amount of information that requires a detailed understanding of the JBoss container architecture, JBoss ships with a standard configuration for the four types of EJBs. This configuration file is called `standardjboss.xml` and it is located in the `conf` directory of any configuration file set that uses EJBs. Listing 5-6 gives a sample of a configuration from the `standardjboss.xml`.

LISTING 5-6. An example of a complex `container-configuration` element from the `server/default/conf/standardjboss.xml` file.

```

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE jboss PUBLIC
  "-//JBoss//DTD JBOSS 3.0//EN"
  "http://www.jboss.org/j2ee/dtd/jboss_3_0.dtd">
<jboss>
  ...
  <container-configurations>
    <container-configuration>
      <container-name>Standard CMP 2.x EntityBean</container-name>
      <call-logging>false</call-logging>
      <container-invoker>org.jboss.proxy.ejb.ProxyFactory</container-invoker>
      ...
    </container-configuration>
  </container-configurations>
</jboss>
  
```

```
<container-interceptors>
    <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
    <interceptor metricsEnabled =
"true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</
interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</
interceptor>

<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</
interceptor>
    <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</
interceptor>
        <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</
interceptor>
</container-interceptors>
<client-interceptors>
    <home>
        <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
        <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
        <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </home>
    <bean>
        <interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
        <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </bean>
    <list-entity>
        <interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
        <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </list-entity>
</client-interceptors>
<instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
<instance-cache>org.jboss.ejb.plugins.EntityInstanceCache</instance-cache>
<persistence-manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</
persistence-manager>
    <transaction-manager>org.jboss.tm.TxManager</transaction-manager>
    <locking-policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLOCK</
locking-policy>
    <container-cache-conf>
        <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</
cache-policy>
            <cache-policy-conf>
                <min-capacity>50</min-capacity>
                <max-capacity>1000000</max-capacity>
                <overager-period>300</overager-period>
                <max-bean-age>600</max-bean-age>
                <resizer-period>400</resizer-period>
```

```

<max-cache-miss-period>60</max-cache-miss-period>
<min-cache-miss-period>1</min-cache-miss-period>
<cache-load-factor>0.75</cache-load-factor>
</cache-policy-conf>
</container-cache-conf>
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
</container-pool-conf>
<commit-option>B</commit-option>
</container-configuration>
...
</container-configurations>
</jboss>

```

Figure 5-5and Listing 5-6 demonstrate how extensive the container configuration options are. The container configuration information can be specified at two levels. The first is in the standardjboss.xml file contained in the configuration set directory. The second is at the ejb-jar level. By placing a jboss.xml file in the ejb-jar META-INF directory, you can specify either overrides for container configurations in the standardjboss.xml file, or entirely new named container configurations. This provides great flexibility in the configuration of containers. As you have seen, all container configuration attributes have been externalized and as such are easily modifiable. Knowledgeable developers can even implement specialized container components, such as instance pools or caches, and easily integrate them with the standard container configurations to optimize behavior for a particular application or environment.

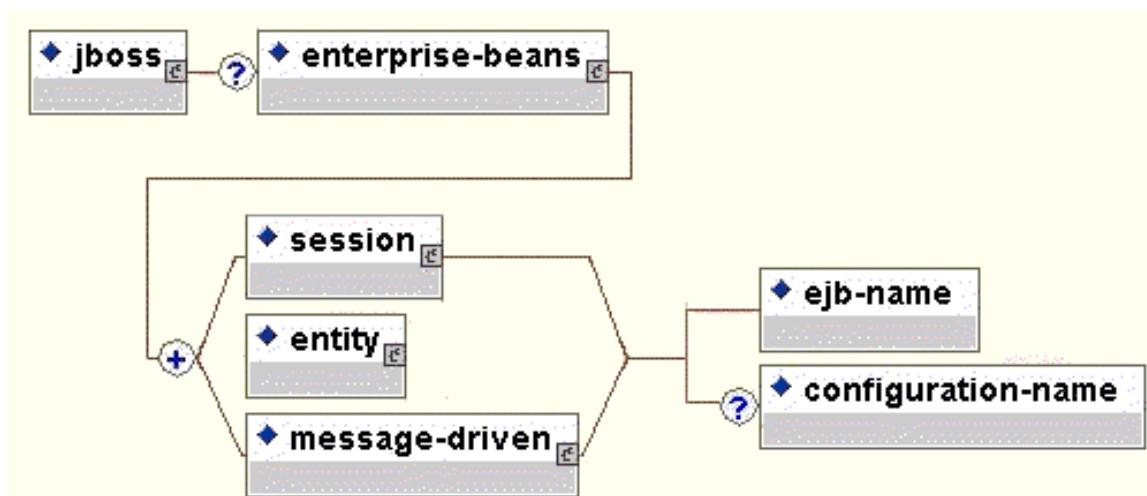


FIGURE 5-6. The jboss.xml descriptor EJB to container configuration mapping elements

How an EJB deployment .chooses its container configuration is based on the explicit or implicit jboss/enterprise-beans/<type>/configuration-name element. Figure 5-6 shows the jboss.xml DTD fragment that shows how an EJB can declare which container configuration it should use.

The configuration-name element is a link to a container-configurations/container-configuration element in Figure 5-5. It specifies which container configuration to use for the referring EJB. The link is from a configuration-name element to a container-name element. You are able to specify container configurations per class of EJB by including a container-configuration element in the EJB definition. Typically one does not define completely new container configurations, although this is supported. The typical usage of a jboss.xml level container-configuration is to override one or more aspects of a container-configuration coming from the standardjboss.xml descriptor. This is done by specifying conatiner-configuration that references the name of an existing standardjboss.xml container-configuration/container-name as the value for the container-configuration/extends attribute. Listing 5-7 shows an example of defining a new “Secured Stateless SessionBean” configuration that is an extension of the standard stateless session configuration whose name is “Standard Stateless SessionBean”.

LISTING 5-7. An example of overriding the standardjboss.xml container stateless session beans configuration to enable secured access.

```
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <configuration-name>Secured Stateless SessionBean</configuration-name>
      ...
    </session>
  </enterprise-beans>

  <container-configurations>
    <container-configuration extends="Standard Stateless SessionBean">
      <container-name>Secured Stateless SessionBean</container-name>
      <!-- Override the container socket factories -->
      <security-domain>java:/jaas/my-security-domain</security-domain>
    </container-configuration>
  </container-configurations>
</jboss>
```

If an EJB does not provide a container configuration specification in the deployment unit ejb-jar, the container factory chooses a container configuration from the standardjboss.xml descriptor based on the type of the EJB. So, in reality there is an implicit configuration-name element for every type of EJB, and the mappings from the EJB type to default container configuration name are as follows:

- container-managed persistence entity version 2.0 = Standard CMP 2.x EntityBean
- container-managed persistence entity version 1.1 = Standard CMP EntityBean
- bean-managed persistence entity = Standard BMP EntityBean
- stateless session = Standard Stateless SessionBean
- stateful session = Standard Stateful SessionBean
- message driven = Standard Message Driven Bean

So, it is not necessary to indicate which container configuration an EJB is using if you want to use the default based on the bean type. It probably provides for a more self-contained descriptor to include the configuration-name element, but this is a matter of style.

Now that you know how to specify which container configuration an EJB is using, and that you can define a deployment unit level override, the question is what are all of those container-configuration child elements? This question will be addressed element by element in the following sections. A number of the elements specify interface class implementations whose configuration is affected by other elements, so before starting in on the configuration elements you need to understand the org.jboss.metadata.XmlLoadable interface.

The XmlLoadable interface is a simple interface that consists of a single method. The interface definition is:

```
import org.w3c.dom.Element;
public interface XmlLoadable
{
    public void importXml(Element element) throws Exception;
}
```

Classes implement this interface to allow their configuration to be specified via an XML document fragment. The root element of the document fragment is what would be passed to the importXml method. You will see a few examples of this as the container configuration elements are described in the following sections.

THE CONTAINER-NAME ELEMENT

The container-name element specifies a unique name for a given configuration. EJBs link to a particular container configuration by setting their configuration-name element to the value of the container-name for the container configuration.

THE CALL-LOGGING ELEMENT

The call-logging element expects a boolean (true or false) as its value to indicate whether or not the LogInterceptor should log method calls to a container. This is somewhat obsolete with the change to log4j, which provides a fine-grained logging API.

THE CONTAINER-INVOKER AND CONTAINER-INVOKER-CONF ELEMENTS

The container-invoker element specifies the class name of the factory that sets up the binding between the home proxy and the client interceptor invoker. Examples of available choices include org.jboss.proxy.ejb.ProxyFactory for RMI/JRMP, org.jboss.proxy.ejb.ProxyFactoryHA, org.jboss.ejb.plugins.jms.JMSContainerInvoker for JMS access to message driven beans, org.jboss.ejb.plugins.iiop.server.IIOPContainerInvoker for RMI/IIOP.

The container-invoker class is configured using the container-invoker-conf element, provided that the class that implements the org.jboss.ejb.ContainerInvoker interface also implements the XmlLoadable interface. If it does, it is simply passed the XML document container-invoker-conf element to use as it chooses. As of 3.0.1 only the JMSContainerInvoker makes use of the invoker-conf, and the following child elements are meaningful:

- **JMSProviderAdapterJNDI**, specifies the JNDI name of the org.jboss.jms.jndi.JMSProviderAdapter implementation to use to setup the JMS layer.

- **ServerSessionPoolFactoryJNDI**, specifies the JNDI name of the org.jboss.jms.asf.ServerSessionPoolFactory implementation to use for creating the javax.jms.ServerSessionPool which will be used to manage the concurrency of the MDBs.
- **MaximumSize**, specifies the upper limit to the number of concurrent MDBs that will be allowed for the JMS destination associated with a given MDB deployment. This defaults to 15 if not specified.
- **MaxMessages**, specifies the maxMessages parameter value for the createConnectionConsumer method of javax.jms.QueueConnection and javax.jms.TopicConnection interfaces, as well as the maxMessages parameter value for the createDurableConnectionConsumer method of javax.jms.TopicConnection. It is the maximum number of messages that can be assigned to a server session at one time. This defaults to 1 if not specified. This value should not be modified from the default unless your JMS provider indicates this is supported.

In JBoss 3.2 the invoker architecture will be further extended to support multiple active invokers for a given EJB deployment.

THE CONTAINER-INTERCEPTORS ELEMENT

The container-interceptors element specifies one or more interceptor elements that are to be configured as the method interceptor chain for the container. The value of the interceptor element is a fully qualified class name of an org.jboss.ejb.Interceptor interface implementation. The container interceptors form a linked-list like structure through which EJB method invocations pass. The first interceptor in the chain is invoked when ContainerInvoker passes a method invocation to the container. The last interceptor invokes the business method on the bean. We will discuss the Interceptor interface latter in this chapter when we talk about the container plugin framework. Generally, care must be taken when changing an existing standard EJB interceptor configuration as the EJB contract regarding security, transactions, persistence, and thread safety derive from the interceptors.

THE INSTANCE-POOL AND CONTAINER-POOL-CONF ELEMENTS

The instance-pool element specifies the fully qualified class name of an org.jboss.ejb.InstancePool interface implementation to use as the container InstancePool. We will discuss the InstancePool interface in detail latter in this chapter when we talk about the container plugin framework.

The container-pool-conf is passed to the InstancePool implementation class given by the instance-pool element if it implements XmlLoadable interface. All current JBoss InstancePool implementations derive from the org.jboss.ejb.plugins.AbstractInstancePool class and it provides support for the MinimumSize and MaximumSize container-pool-conf child elements. The MinimumSize element gives the minimum number of instances to keep in the pool, while the MaximumSize specifies the maximum number of pool instances that are allowed. The default use of MaximumSize may not be what you expect. The pool MaximumSize is the maximum number of EJB instances that are kept available, but additional instances can be created if the number of concurrent requests exceeds the MaximumSize value. If you want to limit the maximum concurrency of an EJB to the pool MaximumSize, you need to set the strictMaximumSize element to true. When strictMaximumSize is true, only MaximumSize EJB instances may be active. When there are MaximumSize active instances, any subsequent requests will be blocked until an instance is freed back to the pool. The default value for strictMaximumSize is false. How long a request blocks waiting for an instance pool object is con-

trolled by the `strictTimeout`: element. It defines the time in milliseconds to wait for an instance to be returned to the pool when there are `MaximumSize` active instances. A value less than or equal to 0 will mean not to wait at all. When a request times out waiting for an instance a `java.rmi.ServerException` is generated and the call aborted. This is parsed as an Integer so that max wait time is 2147483647 or just under 25 days, and this is the default value.

The `Synchronized` child element is a true/false flag used by the specialty `org.jboss.ejb.plugins.SingletonStatelessSessionInstancePool` class that supports a single stateless session instance or a singleton pattern. If `Synchronized` is true only one method invocation thread at a time is allowed to access the singleton session bean. If `Synchronized` is false then the singleton may have multiple method invocation threads active at any given moment and the session bean would have to be coded in a thread-safe manner.

THE INSTANCE-CACHE AND CONTAINER-CACHE-CONF ELEMENTS

The instance-cache element specifies the fully qualified class name of the `org.jboss.ejb.InstanceCache` interface implementation. This element is only meaningful for entity and stateful session beans as these are the only EJB types that have an associated identity. We will discuss the `InstanceCache` interface in detail latter in this chapter when we talk about the container plugin framework.

The container-cache-conf element is passed to the `InstanceCache` implementation if it supports the `XmlLoadable` interface. All current JBoss `InstanceCache` implementations derive from the `org.jboss.ejb.plugins.AbstractInstanceCache` class and it provides support for the `XmlLoadable` interface and uses the cache-policy child element as the fully qualified class name of an `org.jboss.util.CachePolicy` implementation that acts as the instance cache store. The `cache-policy-conf` child element is passed to the `CachePolicy` implementation if it supports the `XmlLoadable` interface. If it does not, the `cache-policy-conf` will silently be ignored.

There are two JBoss implementations of `CachePolicy` used by the standard jboss.xml configuration that support the current array of `cache-policy-conf` child elements. The classes are `org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy` and `org.jboss.ejb.plugins.LRUSTatefulContextCachePolicy`. The `LRUEnterpriseContextCachePolicy` is used by entity bean containers while the `LRUSTatefulContextCachePolicy` is used by stateful session bean containers. Both cache policies support the following `cache-policy-conf` child elements:

- **min-capacity**, specifies the minimum capacity of this cache
- **max-capacity**, specifies the maximum capacity of the cache, which cannot be less than **min-capacity**.
- **overager-period**, specifies the period in seconds between runs of the overager task. The purpose of the overager task is to see if the cache contains beans with an age greater than the **max-bean-age** element value. Any beans meeting this criterion will be passivated.
- **max-bean-age**, specifies the maximum period of inactivity in seconds a bean can have before it will be passivated by the overager process.
- **resizer-period**, specifies the period in seconds between runs of the resizer task. The purpose of the resizer task is to contract or expand the cache capacity based on the remaining three element values in the following way. When the resizer task executes it checks the current period between cache misses, and if the period is less than the **min-cache-miss-period** value the cache is

expanded up to the **max-capacity** value using the **cache-load-factor**. If instead the period between cache misses is greater than the **max-cache-miss-period** value the cache is contracted using the **cache-load-factor**.

- **max-cache-miss-period**, specifies the time period in seconds in which a cache miss should signal that the cache capacity be contracted. It is equivalent to the minimum miss rate that will be tolerated before the cache is contracted.
- **min-cache-miss-period**, specifies the time period in seconds in which a cache miss should signal that the cache capacity be expanded. It is equivalent to the maximum miss rate that will be tolerated before the cache is expanded.
- **cache-load-factor**, specifies the factor by which the cache capacity is contracted and expanded. The factor should be less than 1. When the cache is contracted the capacity is reduced so that the current ratio of beans to cache capacity is equal to the **cache-load-factor** value. When the cache is expanded the new capacity is determined as $\text{current-capacity} * \frac{1}{\text{cache-load-factor}}$. The actual expansion factor may be as high as 2 based on an internal algorithm based on the number of cache misses. The higher the cache miss rate the closer the true expansion factor will be to 2.

The LRUStatefulContextCachePolicy also supports the remaining child elements:

- **remover-period**, specifies the period in seconds between runs of the remover task. The remover task removes passivated beans that have not been accessed in more than **max-bean-life** seconds. This task prevents stateful session beans that were not removed by users from filling up the passivation store.
- **max-bean-life**, specifies the maximum period of inactivity in seconds that a bean can exist before being removed from the passivation store.

An alternative cache policy implementation is the org.jboss.ejb.plugins.NoPassivationCachePolicy class, which simply never passivates instances. It uses an in-memory HashMap implementation that never discards instances unless they are explicitly removed. This class does not support any of the cache-policy-conf configuration elements.

THE PERSISTENCE-MANAGER ELEMENT

The persistence-manager element value specifies the fully qualified class name of the persistence manager implementation. The type of the implementation depends on the type of EJB. For stateful session beans it must be an implementation of the org.jboss.ejb.StatefulSessionPersistenceManager interface. For BMP entity beans it must be an implementation of the org.jboss.ejb.EntityPersistenceManager interface, while for CMP entity beans it must be an implementation of the org.jboss.ejb.EntityPersistenceStore interface.

THE TRANSACTION-MANAGER ELEMENT

The transaction-manager element is now obsolete and no longer used as the JTA implementation class is obtained from the well-known JNDI location “java:/TransactionManager”. See “Transactions on JBoss - The JTA Transaction Service” on page 133. for information on integrating an alternate JTA transaction manager.

THE LOCKING-POLICY ELEMENT

The locking-policy element gives the fully qualified class name of the EJB lock implementation to use. This class must implement the org.jboss.ejb.BeanLock interface. The current JBoss versions include:

- org.jboss.ejb.plugins.lock.MethodOnlyEJBLock, an implementation that does not perform any pessimistic transactional locking. It does provide locking for single-threaded non-reentrant beans.
- org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock, an implementation that holds threads awaiting the transactional lock to be freed in a fair FIFO queue. Non-transactional threads are also put into this wait queue as well. Unlike the SimplePessimisticEJBLock that notifies all threads on transaction completion, this class pops the next waiting transaction from the queue and notifies only those threads waiting associated with that transaction. This class should perform better than SimplePessimisticEJBLock when contention is high. This implementation is the current default used by the standard configurations.
- org.jboss.ejb.plugins.lock.SimplePessimisticEJBLock, an implementation that is similar to QueuedPessimisticEJBLock, but threads simply are blocked by waiting on the lock and are notified using the notifyAll broadcast.

We will talk in more detail about the locking policy usage in section “Entity Bean Locking and Deadlock Detection” on page 173.

THE COMMIT-OPTION AND OPTIOND-REFRESH-RATE ELEMENT

The commit-option value specifies the EJB entity bean persistent storage commit option. It must be one of A, B, C or D. The meaning of the option values is:

- A, the container caches the beans state between transactions. This option assumes that the container is the only user accessing the persistent store. This assumption allows the container to synchronize the in-memory state from the persistent storage only when absolutely necessary. This occurs before the first business method executes on a found bean or after the bean is passivated and reactivated to serve another business method. This behavior is independent of whether the business method executes inside a transaction context.
- B, the container caches the bean state between transactions. However, unlike option A the container does not assume exclusive access to the persistent store. Therefore, the container will synchronize the in-memory state at the beginning of each transaction. Thus, business methods executing in a transaction context don't see much benefit from the container caching the bean, whereas business methods executing outside a transaction context (transaction attributes Never, NotSupported or Supports) access the cached (and potentially invalid) state of the bean.
- C, the container does not cache bean instances. The in-memory state must be synchronized on every transaction start. For business methods executing outside a transaction the synchronization is still performed, but the ejbLoad executes in the same transaction context as that of the caller.
- D, is a JBoss specific feature which is not described in the EJB specification. It is a lazy read scheme where bean state is cached between transactions as with option A, but the state is periodically resynchronized with that of the persistent store. The default time between reloads is 30 seconds, but may be configured using the optiond-refresh-rate element.

THE SECURITY-DOMAIN ELEMENT

Inside the EJB org.jboss.ejb.Container class The security-domain element specifies the JNDI name of the object that implements the org.jboss.security.AuthenticationManager and org.jboss.security.RealmMapping interfaces. Usually one specifies the security-domain globally under the jboss root element so that all EJBs in a given deployment are secured. The details of the security manager interfaces and configuring the security layer are discussed in the “” chapter.

The JBoss EJB container uses a framework pattern that allows one to change implementations of various aspects of the container behavior. The container itself does not perform any significant work other than connecting the various behavioral components together. Implementations of the behavioral components are referred to as plugins, because you can plug in a new implementation by changing a container configuration. Examples of plug-in behavior you may want to change include persistence management, object pooling, object caching, container invokers and interceptors. There are four subclasses of the [org.jboss.ejb.Container](#) class, each one implementing a particular bean type:

- [org.jboss.ejb.EntityContainer](#) handles [javax.ejb.EntityBean](#) types
- [org.jboss.ejb.StatelessSessionContainer](#) handles Stateless [javax.ejb.SessionBean](#) types
- [org.jboss.ejb.StatefulSessionContainer](#) handles Stateful [javax.ejb.SessionBean](#) types
- [org.jboss.ejb.MessageDrivenContainer](#) handles [javax.ejb.MessageDrivenBean](#) types

Container Plug-in Framework

The EJB container delegates much of its behavior to components known as container plug-ins. The interfaces that make up the container plugin points include the following:

[org.jboss.ejb.ContainerPlugin](#)

[org.jboss.ejb.ContainerInvoker](#)

[org.jboss.ejb.Interceptor](#)

[org.jboss.ejb.InstancePool](#)

[org.jboss.ejb.InstanceCache](#)

[org.jboss.ejb.EntityPersistanceManager](#)

[org.jboss.ejb.EntityPersistanceStore](#)

[org.jboss.ejb.StatefulSessionPersistenceManager](#)

The container's main responsibility is to manage its plug-ins. This means ensuring that the plug-ins have all the information they need to implement their functionality.

[org.jboss.ejb.ContainerPlugin](#)

The [ContainerPlugin](#) interface is the parent interface of all container plug-in interfaces. It provides a callback that allows a container to provide each of its plug-ins a pointer to the container the plug-in is working on behalf of. The [ContainerPlugin](#) interface is given in Listing 5-8.

LISTING 5-8. The org.jboss.ejb.ContainerPlugin interface

```
public interface ContainerPlugin extends org.jboss.system.Service
{
    /**
     * This callback is set by the container so that the plugin
     * may access its container
     *
     * @param con  the container which owns the plugin
     */
    public void setContainer(Container con);
}
```

org.jboss.ejb.Interceptor

The Interceptor interface enables one to build a chain of method interceptors through which each EJB method invocation must pass. The Interceptor interface is given in Listing 5-9.

LISTING 5-9. The org.jboss.ejb.Interceptor interface

```
import org.jboss.invocation.Invocation;

public interface Interceptor extends ContainerPlugin
{
    public void setNext(Interceptor interceptor);
    public Interceptor getNext();
    public Object invokeHome(Invocation mi) throws Exception;
    public Object invoke(Invocation mi) throws Exception;
}
```

All interceptors defined in the container configuration are created and added to the container interceptor chain by the EJBDeployer. The last interceptor is not added by the deployer but rather by the container itself because this is the interceptor that interacts with the EJB bean implementation.

The order of the interceptor in the chain is important. The idea behind ordering is that interceptors that are not tied to a particular EnterpriseContext instance are positioned before interceptors that interact with caches and pools.

Implementers of the Interceptor interface form a linked-list like structure through which the Invocation object is passed. The first interceptor in the chain is invoked when an invoker passes a Invocation to the container via the JMX bus. The last interceptor invokes the business method on the bean. There are usually on the order of five interceptors in a chain depending on the bean type and container configuration. Interceptor semantic complexity ranges from simple to complex. An example of a simple interceptor would be LoggingInterceptor, while a complex example is EntitySynchronizationInterceptor.

One of the main advantages of an interceptor pattern is flexibility in the arrangement of interceptors. Another advantage is the clear functional distinction between different interceptors. For example,

logic for transaction and security is cleanly separated between the [TXInterceptor](#) and [SecurityInterceptor](#) respectively.

If any of the interceptors fail, the call is terminated at that point. This is a fail-quickly type of semantic. For example, if a secured EJB is accessed without proper permissions, the call will fail as the [SecurityInterceptor](#) before any transactions are started or instances caches are updated.

org.jboss.ejb.InstancePool

An [InstancePool](#) is used to manage the EJB instances that are not associated with any identity. The pools actually manage subclasses of the [org.jboss.ejb.EnterpriseContext](#) objects that aggregate unassociated bean instances and related data. Listing 5-10 gives the [InstancePool](#) interface.

LISTING 5-10. The org.jboss.ejb.InstancePool interface

```
public interface InstancePool extends ContainerPlugin
{
    /**
     * Get an instance without identity. Can be used
     * by finders and create-methods, or stateless beans
     *
     * @return      Context /w instance
     * @exception   RemoteException
     */
    public EnterpriseContext get() throws Exception;

    /**
     * Return an anonymous instance after invocation.
     *
     * @param      ctx
     */
    public void free(EnterpriseContext ctx);

    /**
     * Discard an anonymous instance after invocation.
     * This is called if the instance should not be reused,
     * perhaps due to some exception being thrown from it.
     *
     * @param      ctx
     */
    public void discard(EnterpriseContext ctx);

    /**
     * Return the size of the pool.
     *
     * @return the size of the pool.
     */
    public int getCurrentSize();

    /**
     * Get the maximum size of the pool.
     *
     * @return the size of the pool.
     */
    public int getMaxSize();
```

```
}
```

Depending on the configuration, a container may choose to have a certain size of the pool contain recycled instances, or it may choose to instantiate and initialize an instance on demand.

The pool is used by the InstanceCache implementation to acquire free instances for activation, and it is used by interceptors to acquire instances to be used for Home interface methods (create and finder calls).

org.jboss.ejb.InstanceCache

The container InstanceCache implementation handles all EJB-instances that are in an active state, meaning bean instances that have an identity attached to them. Only entity and stateful session beans are cached, as these are the only bean types that have state between method invocations. The cache key of an entity bean is the bean primary key. The cache key for a stateful session bean is the session id. Listing 5-11 gives the InstanceCache interface.

LISTING 5-11. The org.jboss.ejb.InstanceCache interface

```
public interface InstanceCache extends ContainerPlugin
{
    /**
     * Gets a bean instance from this cache given the identity.
     * This method may involve activation if the instance is not
     * in the cache.
     * Implementation should have O(1) complexity.
     * This method is never called for stateless session beans.
     *
     * @param id the primary key of the bean
     * @return the EnterpriseContext related to the given id
     * @exception RemoteException in case of illegal calls
     *          (concurrent / reentrant), NoSuchObjectException if
     * the bean cannot be found.
     * @see #release
     */
    public EnterpriseContext get(Object id)
        throws RemoteException, NoSuchObjectException;

    /**
     * Inserts an active bean instance after creation or activation.
     * Implementation should guarantee proper locking and O(1) complexity.
     *
     * @param ctx the EnterpriseContext to insert in the cache
     * @see #remove
     */
    public void insert(EnterpriseContext ctx);

    /**
     * Releases the given bean instance from this cache.
     * This method may passivate the bean to get it out of the cache.
     * Implementation should return almost immediately leaving the
}
```

```
*   passivation to be executed by another thread.  
*  
* @param ctx the EnterpriseContext to release  
* @see #get  
*/  
public void release(EnterpriseContext ctx);  
  
/** Removes a bean instance from this cache given the identity.  
 * Implementation should have O(1) complexity and guarantee  
* proper locking.  
*  
* @param id the primary key of the bean  
* @see #insert  
*/  
public void remove(Object id);  
  
/** Checks whether an instance corresponding to a particular  
 * id is active  
*  
* @param id the primary key of the bean  
* @see #insert  
*/  
public boolean isActive(Object id);  
}
```

In addition to managing the list of active instances, the InstanceCache is also responsible for activating and passivating instances. If an instance with a given identity is requested, and it is not currently active, the InstanceCache must use the InstancePool to acquire a free instance, followed by the persistence manager to activate the instance. Similarly, if the InstanceCache decides to passivate an active instance, it must call the persistence manager to passivate it and release the instance to the InstancePool.

org.jboss.ejb.EntityPersistenceManager

The EntityPersistenceManager is responsible for the persistence of EntityBeans. This includes the following:

- Creating an EJB instance in a storage
- Loading the state of a given primary key into an EJB instance
- Storing the state of a given EJB instance
- Removing an EJB instance from storage
- Activating the state of an EJB instance
- Passivating the state of an EJB instance

Listing 5-12 gives the EntityPersistenceManager interface.

LISTING 5-12. The org.jboss.ejb.EntityPersistenceManager interface

```
public interface EntityPersistenceManager extends ContainerPlugin
```

```

{
    /**
     * Returns a new instance of the bean class or a subclass of the bean class.
     *
     * @return the new instance
     */
    Object createBeanClassInstance() throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbCreate method
     * on the instance and to handle the results properly wrt the persistent
     * store.
     *
     * @param m           the create method in the home interface that was
     *                   called
     * @param args        any create parameters
     * @param instance    the instance being used for this create call
     */
    void createEntity(Method m,
                      Object[] args,
                      EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbPostCreate method
     * on the instance and to handle the results properly wrt the persistent
     * store.
     *
     * @param m           the create method in the home interface that was
     *                   called
     * @param args        any create parameters
     * @param instance    the instance being used for this create call
     */
    void postCreateEntity(Method m,
                          Object[] args,
                          EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when single entities are to be found. The
     * persistence manager must find out whether the wanted instance is
     * available in the persistence store, and if so it shall use the
     * ContainerInvoker plugin to create an EJBObject to the instance, which
     * is to be returned as result.
     *
     * @param finderMethod   the find method in the home interface that was
     *                       called
     * @param args           any finder parameters
     * @param instance       the instance to use for the finder call
     * @return               an EJBObject representing the found entity
     */
    Object findEntity(Method finderMethod,
                      Object[] args,

```

```
        EntityEnterpriseContext instance)
throws Exception;

/**
 * This method is called when collections of entities are to be found. The
 * persistence manager must find out whether the wanted instances are
 * available in the persistence store, and if so it shall use the
 * ContainerInvoker plugin to create EJBObjects to the instances, which are
 * to be returned as result.
 *
 * @param finderMethod      the find method in the home interface that was
 *                          called
 * @param args               any finder parameters
 * @param instance           the instance to use for the finder call
 * @return                  an EJBObject collection representing the found
 *                         entities
 */
Collection findEntities(Method finderMethod,
                        Object[] args,
                        EntityEnterpriseContext instance)
throws Exception;

/**
 * This method is called when an entity shall be activated. The persistence
 * manager must call the ejbActivate method on the instance.
 *
 * @param instance          the instance to use for the activation
 *
 * @throws RemoteException  thrown if some system exception occurs
 */
void activateEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is called whenever an entity shall be load from the
 * underlying storage. The persistence manager must load the state from
 * the underlying storage and then call ejbLoad on the supplied instance.
 *
 * @param instance          the instance to synchronize
 *
 * @throws RemoteException  thrown if some system exception occurs
 */
void loadEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is used to determine if an entity should be stored.
 *
 * @param instance          the instance to check
 * @return true, if the entity has been modified
 * @throws Exception         thrown if some system exception occurs
 */
boolean isModified(EntityEnterpriseContext instance) throws Exception;

/**
```

```
* This method is called whenever an entity shall be stored to the
* underlying storage. The persistence manager must call ejbStore on the
* supplied instance and then store the state to the underlying storage.
*
* @param instance      the instance to synchronize
*
* @throws RemoteException      thrown if some system exception occurs
*/
void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be passivate. The persistence
 * manager must call the ejbPassivate method on the instance.
*
* @param instance      the instance to passivate
*
* @throws RemoteException      thrown if some system exception occurs
*/
void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be removed from the
 * underlying storage. The persistence manager must call ejbRemove on the
 * instance and then remove its state from the underlying storage.
*
* @param instance      the instance to remove
*
* @throws RemoteException      thrown if some system exception occurs
* @throws RemoveException      thrown if the instance could not be removed
*/
void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}
```

As per the EJB 2.0 specification, JBoss supports two entity bean persistence semantics: Container Managed Persistence (CMP) and Bean Managed Persistence (BMP). The CMP implementation uses an implementation of the [org.jboss.ejb.EntityPersistanceStore](#) interface. By default this is the [org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager](#) which is the entry point for the CMP2 persistence engine. Listing 5-13 gives the [EntityPersistanceStore](#) interface.

LISTING 5-13. The org.jboss.ejb.EntityPersistanceStore interface

```
public interface EntityPersistenceStore extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the bean class.
     *
     * @return      the new instance
     *
     * @throws Exception
     */

```

```
Object createBeanClassInstance() throws Exception;

/**
 * Initializes the instance context.
 *
 * <p>This method is called before createEntity, and should
 *     reset the value of all cmpFields to 0 or null.
 *
 * @param ctx
 *
 * @throws RemoteException
 */
void initEntity(EntityEnterpriseContext ctx);

/**
 * This method is called whenever an entity is to be created.
 * The persistence manager is responsible for handling the results properly
 * wrt the persistent store.
 *
 * @param m           the create method in the home interface that was
 *                   called
 * @param args        any create parameters
 * @param instance    the instance being used for this create call
 * @return            The primary key computed by CMP PM or null for BMP
 *
 * @throws Exception
 */
Object createEntity(Method m,
                    Object[] args,
                    EntityEnterpriseContext instance)
throws Exception;

/**
 * This method is called when single entities are to be found. The
 * persistence manager must find out whether the wanted instance is
 * available in the persistence store, if so it returns the primary key of
 * the object.
 *
 * @param finderMethod   the find method in the home interface that was
 *                       called
 * @param args           any finder parameters
 * @param instance       the instance to use for the finder call
 * @return               a primary key representing the found entity
 *
 * @throws RemoteException   thrown if some system exception occurs
 * @throws FinderException   thrown if some heuristic problem occurs
 */
Object findEntity(Method finderMethod,
                  Object[] args,
                  EntityEnterpriseContext instance)
throws Exception;

/**
 * This method is called when collections of entities are to be found. The
 * persistence manager must find out whether the wanted instances are
```

```
* available in the persistence store, and if so it must return a
* collection of primaryKeys.
*
* @param finderMethod      the find method in the home interface that was
*                           called
* @param args               any finder parameters
* @param instance           the instance to use for the finder call
* @return                  an primary key collection representing the found
*                           entities
*
* @throws RemoteException   thrown if some system exception occurs
* @throws FinderException    thrown if some heuristic problem occurs
*/
Collection findEntities(Method finderMethod,
                        Object[] args,
                        EntityEnterpriseContext instance)
throws Exception;

/**
 * This method is called when an entity shall be activated.
*
* <p>With the PersistenceManager factorization most EJB calls should not
* exists However this calls permits us to introduce optimizations in
* the persistence store. Particularly the context has a
* "PersistenceContext" that a PersistenceStore can use (JAWS does for
* smart updates) and this is as good a callback as any other to set it
* up.
* @param instance      the instance to use for the activation
* @throws RemoteException   thrown if some system exception occurs
*/
void activateEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is called whenever an entity shall be load from the
* underlying storage. The persistence manager must load the state from
* the underlying storage and then call ejbLoad on the supplied instance.
*
* @param instance      the instance to synchronize
*
* @throws RemoteException   thrown if some system exception occurs
*/
void loadEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is used to determine if an entity should be stored.
*
* @param instance      the instance to check
* @return true, if the entity has been modified
* @throws Exception    thrown if some system exception occurs
*/
boolean isModified(EntityEnterpriseContext instance) throws Exception;
```

```
/***
 * This method is called whenever an entity shall be stored to the
 * underlying storage. The persistence manager must call ejbStore on the
 * supplied instance and then store the state to the underlying storage.
 *
 * @param instance      the instance to synchronize
 *
 * @throws RemoteException      thrown if some system exception occurs
 */
void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/***
 * This method is called when an entity shall be passivate. The persistence
 * manager must call the ejbPassivate method on the instance.
 *
 * <p>See the activate discussion for the reason for exposing EJB callback
 * calls to the store.
 *
 * @param instance      the instance to passivate
 *
 * @throws RemoteException      thrown if some system exception occurs
 */
void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/***
 * This method is called when an entity shall be removed from the
 * underlying storage. The persistence manager must call ejbRemove on the
 * instance and then remove its state from the underlying storage.
 *
 * @param instance      the instance to remove
 *
 * @throws RemoteException      thrown if some system exception occurs
 * @throws RemoveException      thrown if the instance could not be removed
 */
void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}
```

The default BMP implementation of the EntityPersistenceManager interface is org.jboss.ejb.plugins.BMPPersistenceManager. The BMP persistence manager is fairly simple since all persistence logic is in the entity bean itself. The only duty of the persistence manager is to perform container callbacks.

org.jboss.ejb.StatefulSessionPersistenceManager

The StatefulSessionPersistenceManager is responsible for the persistence of stateful SessionBeans. This includes the following:

- Creating stateful sessions in a storage
- Activating stateful sessions from a storage
- Passivating stateful sessions to a storage

- Removing stateful sessions from a storage

Listing 5-14 gives the StatefulSessionPersistenceManager interface.

LISTING 5-14. The org.jboss.ejb.StatefulSessionPersistenceManager interface

```
public interface StatefulSessionPersistenceManager extends
    ContainerPlugin
{
    public void createSession(Method m, Object[] args,
        StatefulSessionEnterpriseContext ctx)
        throws Exception;

    public void activateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void passivateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void removeSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException, RemoveException;

    public void removePassivated(Object key);
}
```

The default implementation of the StatefulSessionPersistenceManager interface is org.jboss.plugins.StatefulSessionFilePersistenceManager. As its name implies, StatefulSessionFilePersistenceManager utilizes the file system to persist stateful session beans. More specifically, the persistence manager serializes beans in a flat file whose name is composed of the bean name and session id with a .ser extension. The persistence manager restores a bean's state during activation and respectively stores its state during passivation from the bean's .ser file.

Entity Bean Locking and Deadlock Detection

Bill Burke

This section provides information on what entity bean locking is and how entity beans are accessed and locked within JBoss 3.0. It also describes the problems you may encounter as you use Entity Beans within your system and how to combat these issues. Deadlocking is formally defined and examined. And, finally, we walk you through how to fine tune your system in terms of Entity Bean locking.

Why JBoss Needs Locking

Locking is all about protecting the integrity of your data. Sometimes you need to be sure that only one user can update critical data at one time. Sometimes, access to sensitive objects in your system need to be serialized so that data is not corrupted by concurrent reads and writes. Databases traditionally provide this sort of functionality with transactional scopes and table and row locking facilities.

Entity Beans are a great way to provide an object-oriented interface to relational data. Beyond that, they can improve performance by taking the load off of the database through caching and delaying updates until absolutely needed so that the database efficiency can be maximized. But, with caching, data integrity is a problem, so some form of application server level locking is needed for Entity Beans to provide the transaction isolation properties that you are used to with traditional databases.

Entity Bean Lifecycle

With the default configuration of JBoss there is only one active instance of a given Entity Bean in memory at one time. This applies for every cache configuration and every type of commit-option. The lifecycle for this instance is different for every commit-option though.

- For commit-option ‘A’, this instance is cached and used between transactions.
- For commit-optoin ‘B’, this instance is cached and used between transactions, but is marked as ‘dirty’ at the end of a transaction. This means that at the start of a new transaction ejbLoad must be called.
- For commit-option ‘C’, this instance is marked as ‘dirty’, released from the cache, and marked for passivation at the end of a transaction.
- For commit-option ‘D’, a background refresh thread periodically calls ejbLoad on stale beans within the cache. Otherwise, this option works in the same way as ‘A’.

Marked for passivation means that the bean is placed within a queue. Each Entity Bean container has a Passivation Thread that periodically passivates beans that have been placed in the queue. A bean is pulled out of the passivation queue and reused if the application requests access to a bean of the same primary key.

On an exception or transaction rollback, the entity bean instance is thrown out of cache entirely and is not even put in the passivation queue and is not reused by an instance pool. Except for the passivation queue, there is no Entity Bean instance pooling. This created too many problems and bugs in the system so it was removed.

Default Locking Behavior

Since JBoss 2.4.1, Entity Bean locking has been totally decoupled from the Entity Bean instance. The logic for locking is totally isolated and managed in a separate lock object. This allows for funkier implementations of the Entity Bean lifecycle that is described later in this section.

Because there is only one allowed instance of a given Entity Bean active at one time, JBoss employs two types of locks to ensure data integrity and to conform to the EJB spec.

Method Lock

The method lock ensures that only one thread of execution at a time can invoke on a given Entity Bean. This is required by the EJB spec. But, this single-threadedness, can be overridden by marking the bean reentrant in its deployment descriptor.

Transaction Lock

A transaction lock ensures that only one transaction at a time has access to a give Entity Bean. This ensures the ACID properties of transactions at the application server level. Since, by default, there is only one active instance of any given Entity Bean at one time, JBoss must protect this instance from *dirty reads* and *dirty writes*. So, the default Entity Bean locking behavior will lock an Entity Bean within a transaction until it completes. This means that if any method at all is invoked on an Entity Bean within a transaction, ***no other transaction can have access to this bean until the holding transaction commits or is rolled back.***

Pluggable Interceptors and Locking Policy

We saw that the basic Entity Bean lifecycle and behavior is defined by the container configuration defined in *standardjboss.xml* descriptor. Listing 5-15 shows the container-interceptors defintion for the “Standard CMP 2.x EntityBean” configuration.

LISTING 5-15. The “Standard CMP 2.x EntityBean” interceptor definition

```
<container-configuration>
  <container-name>Standard CMP 2.x EntityBean</container-name>
...
  <container-interceptors>
    <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
    <interceptor>org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>

    <interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
      <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
    </interceptor>
  </container-interceptors>
```

The interceptors shown above define most of the behavior of the Entity Bean. Below is an explanation of the interceptors that are relevant to this section.

EntityLockInterceptor. This interceptor's role is to schedule any locks that must be acquired before the invocation is allowed to proceed. This interceptor is very lightweight and delegates all locking behavior to a pluggable locking policy.

EntityInstanceInterceptor. The job of this interceptor is to find the Entity Bean within the cache or create a new one. This interceptor also ensures that there is only one active instance of a bean in memory at one time.

EntitySynchronizationInterceptor. The role of this interceptor is to synchronize the state of the cache with the underlying storage. It does this with the ejbLoad and ejbStore semantics of the EJB specification. In the presence of a transaction this is triggered by transaction demarcation. It registers a callback with the underlying transaction monitor through the JTA interfaces. If there is no transaction the policy is to store state upon returning from invocation. The synchronization policies A,B,C of the specification are taken care of here as well as the JBoss specific commit-option 'D'.

Deadlock

Finding deadlock problems and resolving them is the topic of this section. We will describe what deadlocking means, how you can detect it within your application, and how you can resolve deadlocks. Deadlock can occur when two or more threads have locks on shared resources. For example Figure 5-7 illustrates a simple deadlock scenario. Here, Thread 1 has the lock for Bean A, and Thread 2 has the lock for Bean B. At a later time, Thread 1 tries to lock Bean B and blocks because Thread 2 has it. Likewise, as Thread 2 tries to lock A it also blocks because Thread 1 has the lock. At this point both threads are deadlocked waiting for access to the resource already locked by the other thread.

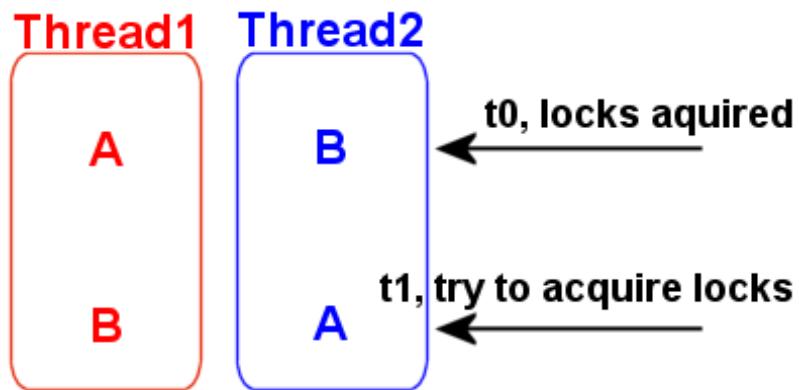


FIGURE 5-7. Deadlock definition example

The default locking policy of JBoss is to lock an Entity bean when an invocation occurs in the context of a transaction until the transaction completes. Because of this, it is very easy to encounter deadlock if you have long running transactions that access many Entity Beans, or if you are not careful about

ordering the access to them. Various techniques and advanced configurations can be used to avoid deadlocking problems. They are discussed later in this section.

Dedlock Detection

Fortunately, with JBoss 2.4.5 and greater, a deadlock detection algorithm has been introduced in the code base. JBoss holds a global internal graph of waiting transactions and what transactions they are blocking on. Whenever a thread determines that it cannot acquire an Entity Bean lock, it figures out what transaction currently holds the lock on the bean and add itself to the blocked transaction graph. An example of what the graph may look like is given in Table 5-1.

TABLE 5-1. An example blocked transaction table

Blocking TX	Tx that holds needed lock
Tx1	Tx2
Tx3	Tx4
Tx4	Tx1

Before the thread actually blocks it tries to detect whether there is deadlock problem. It does this by traversing the block transaction graph. As it traverses the graph, it keeps track of what transactions are blocked. If it sees a blocked node more than once in the graph, then it knows there is deadlock and will throw an ApplicationDeadlockException. This exception will cause a transaction rollback which will cause all locks that transaction holds to be released. The algorithm for the deadlock detection is found in the BeanLockSupport deadlockDetection method. The code for this method is shown in Listing 5-16.

LISTING 5-16. The org.jboss.ejb.plugins.lock.BeanLockSupport deadlockDetection method

```
// This following is for deadlock detection
protected static HashMap waiting = new HashMap();

public void deadlockDetection(Transaction miTx) throws Exception
{
    HashSet set = new HashSet();
    set.add(miTx);

    Object checkTx = this.tx;
    synchronized(waiting)
    {
        while (checkTx != null)
        {
            Object waitingFor = waiting.get(checkTx);
            if (waitingFor != null)
            {
                if (set.contains(waitingFor))
                {
```

```
        log.error("Application deadlock detected: " + miTx + " has deadlock
conditions");
        throw new ApplicationDeadlockException("application deadlock
detected");
    }
    set.add(waitingFor);
}
checkTx = waitingFor;
}
}
}
```

CATCHING APPLICATIONDEADLOCKEXCEPTION

Since JBoss can detect application deadlock, you should write your application so that it can retry a transaction if the invocation fails because of the ApplicationDeadlockException. Unfortunately, this exception can be deeply embedded within a RemoteException, so you have to search for it in your catch block. For example:

```
try
{
...
}
catch (RemoteException ex)
{
    Throwable cause = null;
    RemoteException rex = ex;
    while (rex.detail != null)
    {
        cause = rex.detail;
        if (cause instanceof ApplicationDeadlockException)
        {
            // ... We have deadlock, force a retry of the transaction.
            break;
        }
        if (cause instanceof RemoteException)
        {
            dex = (RemoteException)cause;
        }
    }
}
```

Advanced Configurations and Optimizations

The default locking behavior of Entity Beans can cause deadlock. Since access to an Entity Bean locks the bean into the transaction, this also can present a huge performance/throughput problem for your application. This section walks through various techniques and configurations that you can use to optimize performance and reduce the possibility of deadlock.

Short-lived Transactions

Make your transactions as short-lived and fine-grained as possible. The shorter the transaction you have, the less likelihood you will have concurrent access collisions and your application throughput will go up.

Ordered Access

Ordering the access to your entity beans can help lessen the likelihood of deadlock. This means making sure that the entity beans in your system are always accessed in the same exact order. In most cases, user applications are just too complicated to use this approach and more advanced configurations are needed.

Read-Only Beans

In JBoss 3.0, Entity Beans can be marked as read-only. When a bean is marked as read-only, it never takes part in a transaction. This means that it is never transactionally locked. Using commit-option ‘D’ with this option is sometimes very useful when your read-only bean’s data is sometimes updated by an external source.

To mark a bean as read-only, use the <read-only> flag in the *jboss.xml* deployment descriptor

LISTING 5-17. Marking an entity bean read-only using *jboss.xml*

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>MyEntityBean</ejb-name>
      <jndi-name>MyEntityHomeRemote</jndi-name>
      <read-only>True</read-only>
    </entity>
  </enterprise-beans>
</jboss>
```

Explicitly Defining Read-Only Methods

After reading and understanding the default locking behavior of Entity Beans, you’re probably wondering, “Why lock the bean if its not modifying the data?” JBoss 3.0 allows you to define what methods on your Entity Bean are read-only so that it will not lock the bean within the transaction if only these types of methods are called. You can define these read-only methods within a *jboss.xml* deployment descriptor. Wildcards are allowed for method names. Listing 5-18 shows an example of declaring all getter methods and the anotherReadOnlyMethod as read-only.

LISTING 5-18. Defining entity bean methods as read-only

```
<jboss>
  <enterprise-beans>
    <entity>
```

```
<ejb-name>nextgen.EnterpriseEntity</ejb-name>
<jndi-name>nextgen.EnterpriseEntity</jndi-name>
<method-attributes>
  <method>
    <method-name>get*</method-name>
    <read-only>true</read-only>
  </method>
  <method>
    <method-name>anotherReadOnlyMethod</method-name>
    <read-only>true</read-only>
  </method>
</method-attributes>
</entity>
</enterprise-beans>
</jboss>
```

Instance Per Transaction Policy

The *Instance Per Transaction* policy is an advanced configuration that can totally wipe away deadlock and throughput problems caused by JBoss's default locking policy. The default Entity Bean locking policy is to only allow one active instance of a bean. The *Instance Per Transaction* policy breaks this requirement by allocating a new instance of a bean per transaction and dropping this instance at the end of the transaction. Because each transaction has its own copy of the bean, there is no need for transaction based locking.

This option does sound great but does have some drawbacks right now. First, the transactional isolation behavior of this option is equivalent to READ_COMMITTED. This can create *repeatable reads* when they are not desired. In other words, a transaction could have a copy of a stale bean. Second, this configuration option currently requires commit-option 'B' or 'C' which can be a performance drain since an ejbLoad must happen at the beginning of the transaction. But, if your application currently requires commit-option 'B' or 'C' anyways, then this is the way to go. The JBoss developers are currently exploring ways to allow commit-option 'A' as well (which would allow the use of caching for this option).

JBoss 3.0.1 and higher has container configurations named "Instance Per Transaction CMP 2.x EntityBean" and "Instance Per Transaction BMP EntityBean" defined in the *standardjboss.xml* that implement this locking policy. To use this configuration, you just have to reference the name of the container configuration to use with your bean in the *jboss.xml* deployment descriptor as show in Listing 5-19.

LISTING 5-19. An example of using the Instance Per Transaction policy available in JBoss 3.0.1+.

```
<jboss>
<enterprise-beans>
<entity>
  <ejb-name>MyCMP2Bean</ejb-name>
  <jndi-name>MyCMP2</jndi-name>
  <configuration-name>Instance Per Transaction CMP 2.x EntityBean</
configuration-name>
</entity>
```

```
<entity>
    <ejb-name>MyBMPBean</ejb-name>
    <jndi-name>MyBMP</jndi-name>
    <configuration-name>Instance Per Transaction BMP EntityBean</configuration-
name>
</entity>
</enterprise-beans>
</jboss>
```

If you're using JBoss 3.0.0, you'll have to setup the configuration yourself within the *jboss.xml* descriptor. The highlighted code shown in Listing 5-20 is what is new in the configuration.

LISTING 5-20. The Instance Per Transaction configuration

```
<jboss>
<container-configurations>
    <container-configuration>
        <container-name>Instance Per Transaction CMP 2.x EntityBean</container-name>
        <call-logging>false</call-logging>
        <container-invoker>org.jboss.proxy.ejb.ProxyFactory</container-invoker>
        <container-interceptors>
            <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
            <interceptor metricsEnabled =
"true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityMultiInstanceInterceptor</
interceptor>
            <interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</
interceptor>

            <interceptor>org.jboss.ejb.plugins.EntityMultiInstanceSynchronizationInterceptor
</interceptor>
            <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</
interceptor>
        </container-interceptors>
        <client-interceptors>
            <home>
                <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
                <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
                <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
            </home>
            <bean>
                <interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
                <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
                <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
            </bean>
            <list-entity>
                <interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
```

```
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
</list-entity>
</client-interceptors>
<instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
<instance-cache>org.jboss.ejb.plugins.EntityInstanceCache</instance-cache>
<persistence-manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-manager>
<transaction-manager>org.jboss.tm.TxManager</transaction-manager>
<b><locking-policy>org.jboss.ejb.plugins.lock.MethodOnlyEJBLock</locking-policy></b>
<container-cache-conf>
  <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-policy>
    <cache-policy-conf>
      <min-capacity>50</min-capacity>
      <max-capacity>1000000</max-capacity>
      <overager-period>300</overager-period>
      <max-bean-age>600</max-bean-age>
      <resizer-period>400</resizer-period>
      <max-cache-miss-period>60</max-cache-miss-period>
      <min-cache-miss-period>1</min-cache-miss-period>
      <cache-load-factor>0.75</cache-load-factor>
    </cache-policy-conf>
  </container-cache-conf>
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
</container-pool-conf>
<b><commit-option>B</commit-option></b>
<container-configuration>
```

Running Within a Cluster

Currently there is no distributed locking capability for Entity Beans within the cluster. This functionality has been delegated to the database and must be supported by the application developer. For clustered Entity Beans, it is suggested to use commit-option ‘B’ or ‘C’ in combination with a row locking mechanism. For CMP, there is a `<row-locking>` configuration option. This option will use “SELECT ... FOR UPDATE” when the bean is loaded from the database. With commit-option ‘B’ or ‘C’, this implements a transactional lock that can be used across the cluster. For BMP, you must explicitly implement the “SELECT ... FOR UPDATE” invocation within the BMP’s ejbLoad method.

Troubleshooting

This section will describe some common locking problems and their solution.

Locking Behavior Not Working

There are many emails on the JBoss User email list which sometimes state that the locking is not working and they are having concurrent access to their beans, and thus *dirty reads*. Here are some common reasons for this:

- If you are upgrading from JBoss 2.4.0 or lower and you have custom <container-configurations>, make sure you have updated these configurations because 2.4.0 and lower does not have the EntityLockInterceptor.
- Make absolutely sure that you have implemented equals and hashCode correctly from custom/complex primary key classes.
- Make absolutely sure that your custom/complex primary key classes serialize correctly. One common mistake is assuming that member variable initializations will be executed when a primary key is unmarshalled.

IllegalStateException

Sometimes, people get:

```
java.lang.IllegalStateException: removing bean lock and it has tx set!
```

This usually means that you have not implemented equals and/or hashCode correctly for your custom/complex primary key class, or that your primary key class is not implemented correctly for serialization.

Hangs and Transaction Timeouts

One long outstanding bug of JBoss is that on a transaction timeout, that transaction is only “marked” for a rollback and not actually rolled back. This responsibility is delegated to the invocation thread. This can cause major problems if the invocation thread hangs indefinitely since things like Entity Bean locks will never be released. The solution to this problem is not a good one. You really just need to avoid doing stuff within a transaction that could hang indefinitely. One common no-no is making connections across the internet or running a web-crawler within a transaction.

Messaging on JBoss - JMS Configuration and Architecture

The JMS API stands for Java™ Message Service Application Programming Interface, and it is used by applications to send asynchronous “business quality” messages to other applications. In the JMS world, messages are not sent directly to other applications. Instead, messages are sent to destinations, also known as “queues” or “topics”. Applications sending messages do not need to worry if the receiving applications are up and running, and conversely, receiving applications do not need to worry about the sending application’s status. Both senders, and receivers only interact with the destinations.

The JMS API is the standardized interface to a JMS provider, sometimes called a Message Oriented Middleware (MOM) system. JBoss comes with a JMS 1.0.2b compliant JMS provider called JBoss Messaging or JBossMQ. When you use the JMS API with JBoss, you are using the JBoss Messaging engine transparently. JBoss Messaging fully implements the JMS specification. Therefore, the best JBoss Messaging user guide is the JMS specification! For more information about the JMS API please visit the [JMS Tutorial](#) or [JMS Downloads & Specifications](#).

This chapter focuses on the JBoss specific aspects of using JMS and message driven beans as well as the JBoss Messaging configuration and MBeans.

JMS Examples

Scott Stark

In this section we discuss the basics needed to use the JBoss JMS implementation. JMS leaves the details of accessing JMS connection factories and destinations as provider specific details. What you need to know to use the JBoss Messaging layer is:

- The location of the javax.jms.QueueConnectionFactory and javax.jms.TopicConnectionFactory. In JBoss both connection factory implementations are located under the JNDI name “ConnectionFactory”.
- How to lookup JMS destinations (javax.jmx.Queue and javax.jms.Topic). Destinations are configured via MBeans as we will see when we discuss the messaging MBeans. Several default queues are defined and are located at the JNDI names: .”queue/testQueue”, “queue/ex”. “queue/A”, “queue/B”, “queue/C”, and “queue/D”. The default topics are located at the JNDI names: “topic/testTopic”, “topic/securedTopic”, and “topic/testDurableTopic”.
- The JBoss Messaging jars. These include concurrent.jar, jbossmq-client.jar, jboss-common-client.jar, jboss-system-client.jar, jnp-client.jar, log4j.jar and jboss.net.jar (only for JDK 1.3)

In the following subsections we will look at examples of the various JMS messaging models and message driven beans. The chapter example source is located under the src/main/org/jboss/chap6 directory of the book examples.

A Point-To-Point Example

Let's start out with a point-to-point (P2P) example. In the P2P model, a sender delivers messages to a queue and a single receiver pulls the message off of the queue. The receiver does not need to be listening to the queue at the time the message is sent. Listing 6-1 shows a complete P2P example that sends a javax.jms.TextMessage to a the queue “queue/testQueue” and asynchronously receives the message from the same queue.

LISTING 6-1. A P2P JMS client example

```
package org.jboss.chap6.ex1;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/** A complete JMS client example program that sends a
TextMessage to a Queue and asynchronously receives the
message from the same Queue.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
```

```
public class SendRecvClient
{
    static CountDown done = new CountDown(1);
    QueueConnection conn;
    QueueSession session;
    Queue que;

    public static class ExListener implements MessageListener
    {
        public void onMessage(Message msg)
        {
            done.release();
            TextMessage tm = (TextMessage) msg;
            try
            {
                System.out.println("onMessage, recv text="
                    + tm.getText());
            }
            catch(Throwable t)
            {
                t.printStackTrace();
            }
        }
    }

    public void setupPTP()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
        conn = qcf.createQueueConnection();
        que = (Queue) iniCtx.lookup("queue/testQueue");
        session = conn.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendRecvAsync(String text)
        throws JMSException, NamingException
    {
        System.out.println("Begin sendRecvAsync");
        // Setup the PTP connection, session
        setupPTP();
        // Set the async listener
        QueueReceiver recv = session.createReceiver(que);
        recv.setMessageListener(new ExListener());
        // Send a text msg
        QueueSender send = session.createSender(que);
        TextMessage tm = session.createTextMessage(text);
        send.send(tm);
        System.out.println("sendRecvAsync, sent text="
            + tm.getText());
        send.close();
        System.out.println("End sendRecvAsync");
    }
}
```

```
}

public void stop() throws JMSEException
    conn.stop();
    {
    session.close();
    conn.close();
}

public static void main(String args[]) throws Exception
{
    SendRecvClient client = new SendRecvClient();
    client.sendRecvAsync("A text msg");
    client.done.acquire();
    client.stop();
    System.exit(0);
}

}
```

The client may be run using the following command line:

```
examples 792>ant -Dchap=6 -Dex=1p2p run-example
Buildfile: build.xml
...
run-example1p2p:
[java] Begin SendRecvClient, now=1027380633296
[java] Begin sendRecvAsync
[java] sendRecvAsync, sent text=A text msg
[java] End sendRecvAsync
[java] onMessage, recv text=A text msg
[java] End SendRecvClient

BUILD SUCCESSFUL

Total time: 5 seconds
```

A Pub-Sub Example

The JMS publish/subscribe (Pub-Sub) message model is a one-to-many model. A publisher sends a message to a topic and all active subscribers of the topic receive the message. Subscribers that are not actively listening to the topic will miss the published message. shows a complete JMS client that sends a javax.jms.TextMessage to a topic and asynchronously receives the message from the same topic.

LISTING 6-2. A Pub-Sub JMS client example

```
package org.jboss.chap6.ex1;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
```

```
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/** A complete JMS client example program that sends a
TextMessage to a Topic and asynchronously receives the
message from the same Topic.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class TopicSendRecvClient
{
    static CountDown done = new CountDown(1);
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public static class ExListener implements MessageListener
    {
        public void onMessage(Message msg)
        {
            done.release();
            TextMessage tm = (TextMessage) msg;
            try
            {
                System.out.println("onMessage, recv text="
                    + tm.getText());
            }
            catch(Throwable t)
            {
                t.printStackTrace();
            }
        }
    }

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }
}
```

```
public void sendRecvAsync(String text)
    throws JMSException, NamingException
{
    System.out.println("Begin sendRecvAsync");
    // Setup the PubSub connection, session
    setupPubSub();
    // Set the async listener

    TopicSubscriber recv = session.createSubscriber(topic);
    recv.setMessageListener(new ExListener());
    // Send a text msg
    TopicPublisher send = session.createPublisher(topic);
    TextMessage tm = session.createTextMessage(text);
    send.publish(tm);
    System.out.println("sendRecvAsync, sent text="
        + tm.getText());
    send.close();
    System.out.println("End sendRecvAsync");
}

public void stop() throws JMSException
{
    conn.stop();
    session.close();
    conn.close();
}

public static void main(String args[]) throws Exception
{
    System.out.println("Begin TopicSendRecvClient,
now="+System.currentTimeMillis());
    TopicSendRecvClient client = new TopicSendRecvClient();
    client.sendRecvAsync("A text msg, now="+System.currentTimeMillis());
    client.done.acquire();
    client.stop();
    System.out.println("End TopicSendRecvClient");
    System.exit(0);
}

}
```

The client may be run using the following command line:

```
examples 796>ant -Dchap=6 -Dex=1ps run-example
Buildfile: build.xml
...
run-example1ps:
[java] Begin TopicSendRecvClient, now=1027381995265
[java] Begin sendRecvAsync
[java] sendRecvAsync, sent text=A text msg, now=1027381995265
[java] End sendRecvAsync
[java] onMessage, recv text=A text msg, now=1027381995265
[java] End TopicSendRecvClient
```

BUILD SUCCESSFUL

Total time: 10 seconds

Now let's break the publisher and subscribers into separate programs to demonstrate that subscribers only receive messages while they are listening to a topic. Listing 6-3 shows a variation of the previous Pub-Sub client that only publishes messages to the "topic/testTopic" topic. The subscriber only client is shown in Listing 6-4.

LISTING 6-3. A JMS publisher client

```
package org.jboss.chap6.ex1;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** A JMS client example program that sends a TextMessage to a Topic

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class TopicSendClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSEException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendAsync(String text)
        throws JMSEException, NamingException
    {
```

```
System.out.println("Begin sendAsync");
// Setup the pub/sub connection, session
setupPubSub();
// Send a text msg
TopicPublisher send = session.createPublisher(topic);
TextMessage tm = session.createTextMessage(text);
send.publish(tm);
System.out.println("sendAsync, sent text="
    + tm.getText());
send.close();
System.out.println("End sendAsync");
}

public void stop() throws JMSEException
{
    conn.stop();
    session.close();
    conn.close();
}

public static void main(String args[]) throws Exception
{
    System.out.println("Begin TopicSendClient,
now="+System.currentTimeMillis());
    TopicSendClient client = new TopicSendClient();
    client.sendAsync("A text msg, now="+System.currentTimeMillis());
    client.stop();
    System.out.println("End TopicSendClient");
    System.exit(0);
}

}
```

LISTING 6-4. A JMS subscriber client

```
package org.jboss.chap6.ex1;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** A JMS client example program that synchronously receives a message a Topic

@author Scott.Stark@jboss.org
@version $Revision:$
```

```
/*
public class TopicRecvClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void recvSync()
        throws JMSException, NamingException
    {
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Wait upto 5 seconds for the message
        TopicSubscriber recv = session.createSubscriber(topic);
        Message msg = recv.receive(5000);
        if( msg == null )
            System.out.println("Timed out waiting for msg");
        else
            System.out.println("TopicSubscriber.recv, msgt="+msg);
    }

    public void stop() throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin TopicRecvClient,
now="+System.currentTimeMillis());
        TopicRecvClient client = new TopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End TopicRecvClient");
        System.exit(0);
    }
}
```

Run the TopicSendClient followed by the TopicRecvClient as follows:

```
examples 800>ant -Dchap=6 -Dex=1ps2 run-example
Buildfile: build.xml
...
run-example1ps2:
    [java] Begin TopicSendClient, now=1027382545921
    [java] Begin sendAsync
    [java] sendAsync, sent text=A text msg, now=1027382545921
    [java] End sendAsync
    [java] End TopicSendClient
    [java] Begin TopicRecvClient, now=1027382548781
    [java] Begin recvSync
    [java] Timed out waiting for msg
    [java] End TopicRecvClient

BUILD SUCCESSFUL

Total time: 21 seconds
```

The output shows that the topic subscriber client (TopicRecvClient) fails to receive the message sent by the publisher.

A Pub-Sub With Durable Topic Example

JMS supports a messaging model that is a cross between the P2P and Pub-Sub models. When a Pub-Sub client wants to receive all messages posted to the topic it subscribes to even when it is not actively listening to the topic, the client may achieve this behavior using a durable topic. Let's look at a variation of the preceding subscriber client that uses a durable topic to ensure that it receives all messages, include those published when the client is not listening to the topic. Listing 6-5 shows the durable topic client with the key differences between the Listing 6-4 client highlighted in bold.

LISTING 6-5. A durable topic JMS client example

```
package org.jboss.chap6.ex1;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** A JMS client example program that synchronously receives a message a Topic
```

```
@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class DurableTopicRecvClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
conn = tcf.createTopicConnection("john", "needle");
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void recvSync()
        throws JMSException, NamingException
    {
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Wait upto 5 seconds for the message
TopicSubscriber recv = session.createDurableSubscriber(topic, "chap6-ex1dtps");
        Message msg = recv.receive(5000);
        if( msg == null )
            System.out.println("Timed out waiting for msg");
        else
            System.out.println("DurableTopicRecvClient.recv, msgt="+msg);
    }

    public void stop() throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println("Begin DurableTopicRecvClient,
now=" +System.currentTimeMillis());
        DurableTopicRecvClient client = new DurableTopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End DurableTopicRecvClient");
        System.exit(0);
    }
}
```

```
}
```

```
}
```

Now run the previous topic publisher with the durable topic subscriber as follows:

```
examples 802>ant -Dchap=6 -Dex=1psdt run-example
Buildfile: build.xml
...
run-example1psdt:
    [java] Begin DurableTopicSetup
    [java] End DurableTopicSetup
    [java] Begin TopicSendClient, now=1027383022625
    [java] Begin sendAsync
    [java] sendAsync, sent text=A text msg, now=1027383022625
    [java] End sendAsync
    [java] End TopicSendClient
    [java] Begin DurableTopicRecvClient, now=1027383024000
    [java] Begin recvSync
    [java] DurableTopicRecvClient.recv, msigt=org.jboss.mq.SpyTextMessage {
    [java] Header {
    [java]     jmsDestination : TOPIC.testTopic.DurableSubscriberExample.chap6-
ex1dtps
        [java]     jmsDeliveryMode : 2
        [java]     jmsExpiration : 0
        [java]     jmsPriority : 4
        [java]     jmsMessageID : ID:1-10273818977181
        [java]     jmsTimeStamp : 1027381897718
        [java]     jmsCorrelationID: null
        [java]     jmsReplyTo : null
        [java]     jmsType : null
        [java]     jmsRedelivered : false
        [java]     jmsPropertiesReadWrite:false
        [java]     msgReadOnly : true
        [java]     producerClientId: ID:1
        [java]   }
        [java] Body {
        [java]   text :A text msg, now=1027381896187
        [java]   }
        [java]   }
    [java] End DurableTopicRecvClient

BUILD SUCCESSFUL

Total time: 8 seconds
```

Items of note for the durable topic example include:

- The TopicConnectionFactory creation in the durable topic client used a username and password, and the TopicSubscriber creation was done using the createDurableSubscriber(Topic, String) method. This is a requirement of durable topic subscribers. The messaging server needs to know what client is requesting the durable topic and what the name of the durable topic subscription is. We will discuss the details of durable topic setup in the configuration section.

- An `org.jboss.chap6.DurableTopicSetup` client was run prior to the `TopicSendClient`. The reason for this is a durable topic subscriber must have registered a subscription at some point in the past in order for the messaging server to save messages. JBoss supports dynamic durable topic subscribers and the `DurableTopicSetup` client simply creates a durable subscription receiver and then exits. This leaves an active durable topic subscriber on the “topic/testTopic” and the messaging server knows that any messages posted to this topic must be saved for latter delivery.
- The `TopicSendClient` does not change for the durable topic. The notion of a durable topic is a subscriber only notion.
- The `DurableTopicRecvClient` sees the message published to the “topic/testTopic” even though it was not listening to the topic at the time the message was published.

A Point-To-Point With MDB Example

The EJB 2.0 specification added the notion of message driven beans (MDB). A MDB is a business component that may be invoked asynchronously. As of the EJB 2.0 specification, JMS was the only mechanism by which MDBs could be accessed. Listing 6-6 shows an MDB that transforms the TextMessages it receives and sends the transformed messages to the queue found in the incoming message `JMSReplyTo` header.

LISTING 6-6. A TextMessage processing MDB

```
package org.jboss.chap6.ex2;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.EJBException;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** An MDB that transforms the TextMessages it receives and send the transformed
 *  messages to the Queue found in the incoming message JMSReplyTo header.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class TextMDB implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx = null;
    private QueueConnection conn;
    private QueueSession session;
```

```
public TextMDB()
{
    System.out.println("TextMDB.ctor, this="+hashCode());
}

public void setMessageDrivenContext(MessageDrivenContext ctx)
{
    this.ctx = ctx;
    System.out.println("TextMDB.setMessageDrivenContext, this="+hashCode());
}

public void ejbCreate()
{
    System.out.println("TextMDB.ejbCreate, this="+hashCode());
    try
    {
        setupPTP();
    }
    catch(Exception e)
    {
        throw new EJBException("Failed to init TextMDB", e);
    }
}
public void ejbRemove()
{
    System.out.println("TextMDB.ejbRemove, this="+hashCode());
    ctx = null;
    try
    {
        if( session != null )
            session.close();
        if( conn != null )
            conn.close();
    }
    catch(JMSEException e)
    {
        e.printStackTrace();
    }
}

public void onMessage(Message msg)
{
    System.out.println("TextMDB.onMessage, this="+hashCode());
    try
    {
        TextMessage tm = (TextMessage) msg;
        String text = tm.getText() + "processed by: "+hashCode();
        Queue dest = (Queue) msg.getJMSReplyTo();
        sendReply(text, dest);
    }
    catch(Throwable t)
    {
        t.printStackTrace();
    }
}
```

```
}

private void setupPTP()
    throws JMSException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("java:comp/env/jms/QCF");
    QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
    conn = qcf.createQueueConnection();
    session = conn.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);
    conn.start();
}
private void sendReply(String text, Queue dest)
    throws JMSException
{
    System.out.println("TextMDB.sendReply, this="+hashCode()
        +", dest='"+dest+"');
    QueueSender sender = session.createSender(dest);
    TextMessage tm = session.createTextMessage(text);
    sender.send(tm);
    sender.close();
}

}
```

The MDB ejb-jar.xml and jboss.xml deployment descriptors are shown in Listing 6-7.

LISTING 6-7. The MDB ejb-jar.xml and jboss.xml descriptors

```
// The ejb-jar.xml descriptor
<?xml version="1.0"?>
<!DOCTYPE ejb-jar
    PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
    "http://java.sun.com/dtd/ejb-jar_2_0.dtd"
>

<ejb-jar>
    <enterprise-beans>
        <message-driven>
            <ejb-name>TextMDB</ejb-name>
            <ejb-class>org.jboss.chap6.ex2.TextMDB</ejb-class>
            <transaction-type>Container</transaction-type>
            <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
            </message-driven-destination>
            <res-ref-name>jms/QCF</res-ref-name>           <resource-ref>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </message-driven>
    </enterprise-beans>
</ejb-jar>
```

```
// The jboss.xml descriptor
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>TextMDB</ejb-name>
      <destination-jndi-name>queue/B</destination-jndi-name>
      <resource-ref>
        <res-ref-name>jms/QCF</res-ref-name>
        <jndi-name>ConnectionFactory</jndi-name>
      </resource-ref>
    </message-driven>
  </enterprise-beans>
</jboss>
```

Listing 6-8 shows a variation of the P2P client that sends several messages to the “queue/B” destination and asynchronously receives the messages as modified by TextMDB from Queue A.

LISTING 6-8. A JMS client that interacts with the TextMDB

```
package org.jboss.chap6.ex2;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/** A complete JMS client example program that sends N
TextMessages to a Queue B and asynchronously receives the
messages as modified by TextMDB from Queue A.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class SendRecvClient
{
  static final int N = 10;
  static CountDown done = new CountDown(N);
  QueueConnection conn;
  QueueSession session;
  Queue queA;
  Queue queB;
```

```
public static class ExListener implements MessageListener
{
    public void onMessage(Message msg)
    {
        done.release();
        TextMessage tm = (TextMessage) msg;
        try
        {
            System.out.println("onMessage, recv text="+tm.getText());
        }
        catch(Throwable t)
        {
            t.printStackTrace();
        }
    }
}

public void setupPTP()
    throws JMSException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("ConnectionFactory");
    QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
    conn = qcf.createQueueConnection();
    queA = (Queue) iniCtx.lookup("queue/A");
    queB = (Queue) iniCtx.lookup("queue/B");
    session = conn.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);
    conn.start();
}

public void sendRecvAsync(String textBase)
    throws JMSException, NamingException, InterruptedException
{
    System.out.println("Begin sendRecvAsync");
    // Setup the PTP connection, session
    setupPTP();
    // Set the async listener for queA
    QueueReceiver recv = session.createReceiver(queA);
    recv.setMessageListener(new ExListener());
    // Send a few text msgs to queB
    QueueSender send = session.createSender(queB);
    for(int m = 0; m < 10; m++)
    {
        TextMessage tm = session.createTextMessage(textBase+"#" +m);
        tm.setJMSReplyTo(queA);
        send.send(tm);
        System.out.println("sendRecvAsync, sent text="+tm.getText());
    }
    System.out.println("End sendRecvAsync");
}

public void stop() throws JMSException
{
```

```
conn.stop();
session.close();
conn.close();
}

public static void main(String args[]) throws Exception
{
    System.out.println("Begin SendRecvClient,now="+System.currentTimeMillis());
    SendRecvClient client = new SendRecvClient();
    client.sendRecvAsync("A text msg");
    client.done.acquire();
    client.stop();
    System.exit(0);
    System.out.println("End SendRecvClient");
}

}
```

Run the client as follows:

```
examples 804>ant -Dchap=6 -Dex=2 run-example
Buildfile: build.xml
...
chap6-ex2-jar:
[jar] Building jar: G:\JBossDocs\AdminDevel\examples\output\chap6\chap6-
ex2.jar

run-example2:
[copy] Copying 1 file to D:\usr\JBoss3.0\jboss-all\build\output\jboss-
3.0.1RC1\server\default\deploy
[echo] Waiting 5 seconds for deploy...
[java] Begin SendRecvClient, now=1027385360031
[java] Begin sendRecvAsync
[java] sendRecvAsync, sent text=A text msg#0
[java] sendRecvAsync, sent text=A text msg#1
[java] sendRecvAsync, sent text=A text msg#2
[java] sendRecvAsync, sent text=A text msg#3
[java] sendRecvAsync, sent text=A text msg#4
[java] sendRecvAsync, sent text=A text msg#5
[java] sendRecvAsync, sent text=A text msg#6
[java] sendRecvAsync, sent text=A text msg#7
[java] sendRecvAsync, sent text=A text msg#8
[java] sendRecvAsync, sent text=A text msg#9
[java] End sendRecvAsync
[java] onMessage, recv text=A text msg#1processed by: 4245685
[java] onMessage, recv text=A text msg#4processed by: 3332063
[java] onMessage, recv text=A text msg#0processed by: 2972067
[java] onMessage, recv text=A text msg#2processed by: 6826579
[java] onMessage, recv text=A text msg#7processed by: 8256600
[java] onMessage, recv text=A text msg#6processed by: 5767514
[java] onMessage, recv text=A text msg#5processed by: 3740851
[java] onMessage, recv text=A text msg#9processed by: 1506732
[java] onMessage, recv text=A text msg#8processed by: 8032804
[java] onMessage, recv text=A text msg#3processed by: 1095232
```

BUILD SUCCESSFUL

Total time: 11 seconds

The corresponding JBoss server console output is:

```
17:49:16,765 INFO [MainDeployer] Starting deployment of package: file:/D:/usr/JBoss3.0/jboss-all/build/output/jboss-3.0.1RC1/server/default/deploy/chap6-ex2.jar
17:49:17,578 INFO [EjbModule] Creating
17:49:17,609 INFO [EjbModule] Deploying TextMDB
17:49:17,718 INFO [JMSContainerInvoker] Creating
17:49:17,781 INFO [DLQHandler] Creating
17:49:18,031 INFO [DLQHandler] Created
17:49:18,218 WARN [SecurityManager] No SecurityMetadadata was available for B
adding default security conf
17:49:18,234 INFO [JMSContainerInvoker] Created
17:49:18,234 INFO [EjbModule] Created
17:49:18,234 INFO [EjbModule] Starting
17:49:18,250 INFO [JMSContainerInvoker] Starting
17:49:18,250 INFO [DLQHandler] Starting
17:49:18,250 INFO [DLQHandler] Started
17:49:18,250 INFO [JMSContainerInvoker] Started
17:49:18,250 INFO [EjbModule] Started
17:49:18,250 INFO [MainDeployer] Deployed package: file:/D:/usr/JBoss3.0/jboss-all/build/output/jboss-3.0.1RC1/server/default/deploy/chap6-ex2.jar
17:49:21,015 WARN [SecurityManager] No SecurityMetadadata was available for A
adding default security conf
17:49:21,218 INFO [STDOUT] TextMDB.ctor, this=1095232
17:49:21,218 INFO [STDOUT] TextMDB.ctor, this=6826579
17:49:21,218 INFO [STDOUT] TextMDB.ctor, this=2972067
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=3332063
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=8032804
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=5767514
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=1506732
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=4245685
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=8256600
17:49:21,234 INFO [STDOUT] TextMDB.ctor, this=3740851
17:49:21,234 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=3332063
17:49:21,234 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=5767514
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=2972067
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=8032804
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=6826579
17:49:21,250 INFO [STDOUT] TextMDB.ejbCreate, this=3332063
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=4245685
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=8256600
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=3740851
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=1506732
17:49:21,250 INFO [STDOUT] TextMDB.setMessageDrivenContext, this=1095232
17:49:21,250 INFO [STDOUT] TextMDB.ejbCreate, this=5767514
17:49:21,250 INFO [STDOUT] TextMDB.ejbCreate, this=2972067
17:49:21,265 INFO [STDOUT] TextMDB.ejbCreate, this=8032804
17:49:21,343 INFO [STDOUT] TextMDB.ejbCreate, this=6826579
17:49:21,343 INFO [STDOUT] TextMDB.ejbCreate, this=4245685
17:49:21,359 INFO [STDOUT] TextMDB.ejbCreate, this=8256600
```

```
17:49:21,375 INFO [STDOUT] TextMDB.ejbCreate, this=3740851
17:49:21,375 INFO [STDOUT] TextMDB.ejbCreate, this=1506732
17:49:21,375 INFO [STDOUT] TextMDB.ejbCreate, this=1095232
17:49:21,421 INFO [STDOUT] TextMDB.onMessage, this=6826579
17:49:21,421 INFO [STDOUT] TextMDB.sendReply, this=6826579, dest=QUEUE.A
17:49:21,421 INFO [STDOUT] TextMDB.onMessage, this=5767514
17:49:21,421 INFO [STDOUT] TextMDB.sendReply, this=5767514, dest=QUEUE.A
17:49:21,453 INFO [STDOUT] TextMDB.onMessage, this=4245685
17:49:21,453 INFO [STDOUT] TextMDB.sendReply, this=4245685, dest=QUEUE.A
17:49:21,468 INFO [STDOUT] TextMDB.onMessage, this=3332063
17:49:21,468 INFO [STDOUT] TextMDB.sendReply, this=3332063, dest=QUEUE.A
17:49:21,468 INFO [STDOUT] TextMDB.onMessage, this=8256600
17:49:21,484 INFO [STDOUT] TextMDB.sendReply, this=8256600, dest=QUEUE.A
17:49:21,484 INFO [STDOUT] TextMDB.onMessage, this=2972067
17:49:21,484 INFO [STDOUT] TextMDB.sendReply, this=2972067, dest=QUEUE.A
17:49:21,562 INFO [STDOUT] TextMDB.onMessage, this=3740851
17:49:21,562 INFO [STDOUT] TextMDB.sendReply, this=3740851, dest=QUEUE.A
17:49:21,578 INFO [STDOUT] TextMDB.onMessage, this=1506732
17:49:21,578 INFO [STDOUT] TextMDB.sendReply, this=1506732, dest=QUEUE.A
17:49:21,578 INFO [STDOUT] TextMDB.onMessage, this=8032804
17:49:21,578 INFO [STDOUT] TextMDB.sendReply, this=8032804, dest=QUEUE.A
17:49:21,578 INFO [STDOUT] TextMDB.onMessage, this=1095232
17:49:21,593 INFO [STDOUT] TextMDB.sendReply, this=1095232, dest=QUEUE.A
```

Items of note in this example include:

- The JMS client has no explicit knowledge that it is dealing with an MDB. The client simply uses the standard JMS APIs to send messages to a queue and receive messages from another queue.
- The MDB declares whether it will listen to a queue or topic in the ejb-jar.xml descriptor. The name of the queue or topic must be specified using a jboss.xml descriptor. In this example the MDB also sends messages to a JMS queue. MDBs may act as queue senders or topic publishers within their onMessage callback.
- The messages received by the client include a “processed by: NNN” suffix, where NNN is the hashCode value of the MDB instance that processed the message. This shows that many MDBs may actively process messages posted to a destination. Concurrent processing is one of the benefits of MDBs.

JBoss Messaging Overview

Hiram Chirino

JBossMQ is composed of several subsystems working together to provide JMS API level services to client applications. To obtain a deeper understanding of the JBossMQ server, and how to optimize it its subsystem will be reviewed in this section.

Invocation Layer

The Invocation Layer (IL) is the subsystem that is responsible for defining the communications protocol that allows clients to send messages to a destination and vice versa. JBossMQ can support running different types of Invocation Layers concurrently. All Invocation Layers have a dual channel nature that allows clients to send messages as it concurrently receives messages from the server.

Each different type of Invocation Layer will bind a JMS ConnectionFactory (configured to use the IL) to a different location in the JNDI tree. Clients can pick the protocol they wish to use by looking up the correct JNDI location.

JBossMQ currently supports four different invocation layers. This section will further examine these ILs.

RMI IL

The first Invocation Layer that was developed was based on Java's Remote Method Invocation (RMI). This is a very robust IL since it is based on standard RMI technology. The RMI IL should be used when your client application has multiple threads sharing one connection.

NOTE: This IL will try to establish a TCP/IP socket from the server to the client. Therefore, clients that sit behind firewalls or have security restrictions prohibiting the use of ServerSockets should not use this IL.

OIL IL

The next Invocation Layer that was developed was the “Optimized” IL (OIL). The OIL uses a custom TCP/IP protocol and serialization protocol that has very low overhead. This is the recommended protocol to be used since it has the best general-purpose performance characteristics.

NOTE: This IL will try to establish a TCP/IP socket from the server to the client. Therefore, clients that sit behind firewalls or have security restrictions prohibiting the use of ServerSockets should not use this IL.

UIL IL

The Unified Invocation Layer (UIL) was developed to allow Applet clients to connect to the server. It is almost identical to the OIL protocol except that a multiplexing layer is used to provide the dual channel characteristics of the IL. The multiplexing layer creates two virtual sockets over one physical socket. This IL is slower than the OIL due to the higher overhead incurred by the multiplexing layer.

JVM IL

The Java Virtual Machine (JVM) Invocation Layer was developed to cut out the TCP/IP overhead when the JMS client is running in the same JVM as the server. This IL uses direct method calls for the server to service the client requests. This increases efficiency since no sockets are created and there is no need for the associated worker threads. This is the IL that should be used by Message

Driven Beans (MDB) or any other component that runs in the same virtual machine as the server such as servlets or MBeans.

Security Manager

The JBossMQ Security Manager is the subsystem that enforces an Access Control List to guard access to your destinations. This subsystem works closely with the State Manager subsystem, which will be discussed later.

Destination Manager

The Destination Manager can be thought as being the central sever for JBossMQ. It keeps track of all the destinations that have been created on the server. It also keeps track of the other server subsystems such as the Message Cache, State Manager, and Persistence Manager.

Message Cache

Messages created in the server are subsequently passed to the Message Cache for memory management. JVM memory usage goes up as messages are added to a destination that does not have any receivers. These messages are held in the main memory until the receiver picks them up. If the Message Cache notices that the JVM memory usage starts passing the defined limits, the Message Cache starts moving those messages from memory to persistent storage on disk. The Message Cache uses a Least Recently Used algorithm to determine which messages should go to disk.

State Manager

The State Manager (SM) is in charge of keeping track of who is allowed to log into the server and what their durable subscriptions are.

Persistence Manager

The Persistence Manager (PM) is used by a destination to store messages marked as being persistent. JBossMQ has several different implementations of the Persistent Manager, but only one can be enabled per server instance. You should enable the Persistence Manager that best matches your requirements.

This section will give you a brief description of the three types of PMs.

File PM

The File PM is the one of the most robust Persistence Manager that comes with JBossMQ. It creates separate directories for each of the destination created on the server. It then stores the each persistent

message as a separate file in the appropriate directory. It does not have the best performance characteristics since it is frequently opening and closing files.

Rolling Logged PM

The Rolling Logged PM is also a file based Persistence Manager but it has better performance than the File PM because it stores multiple messages in one file reducing the file opening and closing overhead. This is a very fast PM but it is less transactionally reliable than the File PM due to its reliance on the `FileOutputStream.flush()` method call. On some operating systems/JVMs the `FileOutputStream.flush()` method does not guarantee that the data has been written to disk by the time the call returns.

JDBC2 PM

The JDBC2 PM is the second version of the original JDBC PM in JBossMQ 2.4.x. It has been substantially simplified and improved. This PM allows you to store persistent messages to relational database using JDBC. The performance of this PM is directly related to the performance that can be obtained from the database. This PM has a very low memory overhead compared to the other Persistence Managers. Furthermore it is also highly integrated with the Message Cache to provide efficient persistence on a system that has a very active Message Cache.

Destinations

A Destination is the object on the JBossMQ server that clients use to send and receive messages. There are two types of destination objects, Queues and Topics. References to the destinations created by JBossMQ are stored in JNDI.

Queues

Clients that are in the Point-to-Point paradigm typically use Queues. They expect that message sent to a Queue will be received by only one other client “once and only once”. If multiple clients are receiving messages from a single queue, the messages will be load balanced across the receivers. Queue objects, by default, will be stored under the JNDI “queue/” sub context.

Topics

Topics are used in the Publish-Subscribe paradigm. When a client publishes a message to a topic, he expects that a copy of the message will be delivered to each client that has subscribed to the topic. Topic messages are delivered in the same manner a television show is delivered. Unless you have the TV on and are watching the show, you will miss it. Similarly, if the client is not up, running and receiving messages from the topics, it will miss messages published to the topic. To get around this problem of missing messages, clients can start a durable subscription. This is like having a VCR record a show you are missing, so that you can see what you missed when you turn your TV back on.

JBoss Messaging Configuration and MBeans

Hiram Chirino

Like all JBoss components the JBoss Messaging layer is configured using MBeans. The configuration files that make up the default JBoss Messaging include:

- conf/jbossmq-state.xml: the configuration file read by the org.jboss.mq.sm.file.DynamicStateManager MBean. This controls the valid username/passwords as well as the active durable topic subscriptions.
- deploy/jbossmq-destinations-service.xml: the default JMS queue and topic destination configurations.
- deploy/jbossmq-service.xml: the service descriptor for the core JBoss Messaging MBeans.
- deploy/jms-ra.rar: a JCA resource adaptor for JMS providers.
- deploy/jms-service.xml: the JMS provider integration services descriptor setup to configure JBoss Messaging as the JMS provider.

We will look at the details of these configuration files as we discuss the associated MBeans in the following subsections. All JBoss JMS objects and sever subsystems are configured via JMX MBeans. Like most other services running in the JBoss micro-kernel architecture, an XML file deployed to the server/default/deploy controls the configuration of the service.

The sever subsystems are configured in the jbossmq-service.xml file. Unless you are an advanced user, you should not have to edit this file. Adjusting this file allows you to optimize and change how the server operates. For more details on how to configure this file see the section below. If you edit this file, you should restart the JBoss server for the changes to correctly take effect.

Most users will need to edit the jbossmq-destinations-service.xml file. This file allows you to define the destinations that your applications need. If you edit this file, you should restart the JBoss server for the changes to correctly take effect.

This section can be used as your reference guide to the jbossmq-service.xml file. This file uses the standard JBoss service XML format that is used to configure MBeans in the JBoss server.Server Subsystem MBeans

org.jboss.mq.il.jvm.JVMServerILService

The org.jboss.mq.il.jvm.JVMServerILService MBean is used to configure the JVM IL. The configurable attributes are as follows:

- **Invoker:** The JMX Object Name of the Invoker that is used to pass client requests to the Destination Manager. This attribute should be setup via a <depends optional-attribute-name="Invoker"> XML tag.

- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a ConnectionFactory setup to use this IL.
- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a XAConnectionFactory setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent. Since it is impossible for JVM IL connection to go bad, it is recommended that you keep this set to “0”.

org.jboss.mq.il.rmi.RMIServerILService

The org.jboss.mq.il.rmi.RMIServerILService is used to configure the RMI IL. The configurable attributes are as follows:

- **Invoker**: The JMX Object Name of the Invoker that is used to pass client requests to the Destination Manager. This attribute should be setup via a `<depends optional-attribute-name="Invoker">` XML tag.
- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a ConnectionFactory setup to use this IL.
- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a XAConnectionFactory setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent.

org.jboss.mq.il.oil.OILServerILService

The org.jboss.mq.il.oil.OILServerILService is used to configure the OIL IL. The configurable attributes are as follows:

- **Invoker**: The JMX Object Name of the Invoker that is used to pass client requests to the Destination Manager. This attribute should be setup via a `<depends optional-attribute-name="Invoker">` XML tag.
- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a ConnectionFactory setup to use this IL.
- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a XAConnectionFactory setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent.
- **ServerBindPort**: The protocol listening port for this IL. If not specified default is 0, which means that a random port will be chosen.
- **BindAddress**: The specific address this IL listens on. This can be used on a multi-homed host for a `java.net.ServerSocket` that will only accept connection requests on one of its addresses.

- **EnableTcpNoDelay:** If set to true, then the TcpNodDelay option is enabled. This improves request response times since TCP/IP packets are sent as soon as the request is flushed. Otherwise request packets may be buffered by the operating system to create larger IP packets.

org.jboss.mq.il.ul.UILServerILService

The org.jboss.mq.il.ul.UILServerILService is used to configure the UIL IL. The configurable attributes are as follows:

- **Invoker:** The JMX Object Name of the Invoker that is used to pass client requests to the Destination Manager. This attribute should be setup via a `<depends optional-attribute-name="Invoker">` XML tag.
- **ConnectionFactoryJNDIRef:** The JNDI location that this IL will bind a ConnectionFactory setup to use this IL.
- **XAConnectionFactoryJNDIRef:** The JNDI location that this IL will bind a XAConnectionFactory setup to use this IL.
- **PingPeriod:** How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent.
- **ServerBindPort:** The protocol listening port for this IL. If not specified default is 0, which means that a random port will be chosen.
- **BindAddress:** The specific address this IL listens on. This can be used on a multi-homed host for a `java.net.ServerSocket` that will only accept connection requests on one of its addresses.
- **EnableTcpNoDelay:** If set to true, then the TcpNodDelay option is enabled. This improves request response times since TCP/IP packets are sent as soon as the request is flushed. Otherwise request packets may be buffered by the operating system to create larger IP packets.

org.jboss.mq.server.jmx.Invoker

The org.jboss.mq.server.jmx.Invoker is used to pass Invocation Layer (IL) requests down to the Destination Manager through an Interceptor Chain. The configurable attributes are as follows:

- **NextInterceptor:** The JMX object name of the next request Interceptor. This attribute is used by all the Interceptors to create the Interceptor Chain. The last interceptor in the chain should be the DestinationManager. This attribute should be setup via a `<depends optional-attribute-name="NextInterceptor">` XML tag.

org.jboss.mq.server.jmx.InterceptorLoader

The org.jboss.mq.server.jmx.InterceptorLoader is used to load a Generic Interceptor and make it part of the Interceptor Chain. This MBean is typically used to load the org.jboss.mq.server.TracingInterceptor, which is used to efficiently log all client requests via trace level log messages. The configurable attributes are as follows:

- **NextInterceptor**: The JMX object name of the next request interceptor. This attribute in all the interceptors forms the chain. The last interceptor in the chain should be the Destination-Manager. This attribute should be setup via a <depends optional-attribute-name="NextInterceptor"> XML tag.
- **InterceptorClass**: The class name of the interceptor that will be loaded and made part of the Interceptor Chain. This class specified here must extend the org.jboss.mq.server.JMSServer-Interceptor java class.

org.jboss.mq.security.SecurityManager

If the org.jboss.mq.security.SecurityManager is part of the Interceptor Chain, then it will enforce the access control lists assigned to the destinations. The Security Manager uses JAAS, and as such requires that an application policy be setup for in the JBoss login-config.xml file. The default configuration is shown in Listing 6-9 .

LISTING 6-9. The default login-config.xml configuration for JBoss messaging

```
<application-policy name = "jbossmq">
    <authentication>
        <login-module code = "org.jboss.mq.sm.file.DynamicLoginModule"
                      flag = "required">
            <module-option name="unauthenticatedIdentity">guest</module-option>
            <module-option
                name="sm.objectnam">jboss.mq:service=StateManager</module-option>
        </login-module>
    </authentication>
</application-policy>
```

The default configuration maps any unauthenticated JBossMQ client to the “guest” role. The configurable attributes are as follows:

- **NextInterceptor**: The JMX object name of the next request interceptor. This attribute in all the interceptors forms the chain. The last interceptor in the chain should be the Destination-Manager. This attribute should be setup via a <depends optional-attribute-name="NextInterceptor"> XML tag.

org.jboss.mq.server.jmx.DestinationManager

The org.jboss.mq.server.jmx.DestinationManager must be the last Interceptor in the Interceptor Chain. The configurable attributes are as follows:

- **PersistenceManager**: The JMX object name of the Persistence Manager you want the server to use. This attribute should be setup via a <depends optional-attribute-name="PersistenceManager"> XML tag.
- **StateManager**: The JMX object name of the State Manager you want the server to use. This attribute should be setup via a <depends optional-attribute-name="StateManager"> XML tag.

org.jboss.mq.server.MessageCache

The server determines when to move messages to secondary storage by using the org.jboss.mq.server.MessageCache MBean. The configurable attributes are as follows:

- **CacheStore:** The JMX object name of an Object that will act as the Cache Store. The Cache Store is used by the MessageCache to move messages to persistent storage. The value you set here typically depends on the type of Persistence Manager you are using. This attribute should be setup via a <depends optional-attribute-name="CacheStore"> XML tag.
- **HighMemoryMark:** The amount of JVM heap memory in megabytes that must be reached before the Message Cache starts to move messages to secondary storage.
- **MaxMemoryMark:** The maximum amount of JVM heap memory in megabytes that the Message Cache considers to be the Max Memory Mark. As memory usage approaches the Max Memory Mark, the Message Cache will move messages to persistent storage so that the number of messagea kept in memory approaches zero.

org.jboss.mq.pm.file.CacheStore

The org.jboss.mq.pm.file.CacheStore MBean should be used as the Cache Store for the Message Cache when you are using the File or Rolling Logged PM. The configurable attributes are as follows:

- **DataDirectory:** The directory used to store messages for the Message Cache.

org.jboss.mq.sm.file.DynamicStateManager

The org.jboss.mq.sm.file.DynamicStateManager MBean should be used as the State Manager assigned to the Destination Manager. The configurable attributes are as follows:

- **StateFile:** The file used to store state information such as created durable subscriptions. The StateFile is in XML format that the server reads and writes data to. You should never edit the XML file while the server is running.

org.jboss.mq.pm.file.PersistenceManager

The org.jboss.mq.pm.file.PersistenceManager should be used as the Persistence Manager assigned to the Destination Manager if you wish to use the File PM. The configurable attributes are as follows:

- **MessageCache:** The JMX object name of the MessageCache that has been assigned to the Destination Manager. This attribute should be setup via a <depends optional-attribute-name="MessageCache"> XML tag.
- **DataDirectory:** The directory used to store persistent messages.

org.jboss.mq.pm.rollinglogged.PersistenceManager

The org.jboss.mq.pm.rollinglogged.PersistenceManager should be used as the Persistence Manager assigned to the Destination Manager if you wish to use the Rolling Logged PM. The configurable attributes are as follows:

- **MessageCache**: The JMX object name of the MessageCache that has been assigned to the Destination Manager. This attribute should be setup via a <depends optional-attribute-name="MessageCache"> XML tag.
- **DataDirectory**: The directory used to store persistent messages.

org.jboss.mq.pm.jdbc2.PersistenceManager

The org.jboss.mq.pm.jdbc2.PersistenceManager should be used as the Persistence Manager assigned to the Destination Manager if you wish to use the JDBC2 PM. This PM has been tested against the Hypersonic Database. The configurable attributes are as follows:

- **MessageCache**: The JMX object name of the MessageCache that has been assigned to the Destination Manager. This attribute should be setup via a <depends optional-attribute-name="MessageCache"> XML tag.
- **DataSource**: The JMX object name of the JCA data source that will be used to obtain JDBC connections. This attribute should be setup via a <depends optional-attribute-name="DataSource"> XML tag. You may also need to add another <depends> XML tag to wait for the data source Connection Manager to be started before this PM is started.
- **SqlProperties**: A property list is used to define the SQL Queries and other JDBC2 Persistence Manager options. You will need to adjust these properties if you which to run against another database other than HypersonicSQL. See Listing 6-10 for default setting for this attribute.

LISTING 6-10. Default SqlProperties

```
<attribute name="SqlProperties">
    BLOB_TYPE=OBJECT_BLOB
    INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
    INSERT_MESSAGE = INSERT INTO JMS_MESSAGES (MESSAGEID, DESTINATION, \
        MESSAGEBLOB, TXID, TXOP) VALUES(?, ?, ?, ?, ?)
    SELECT_ALL_UNCOMMITTED_TXS = SELECT TXID FROM JMS_TRANSACTIONS
    SELECT_MAX_TX = SELECT MAX(TXID) FROM JMS_MESSAGES
    SELECT_MESSAGES_IN_DEST = SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES \
        WHERE DESTINATION=?
    SELECT_MESSAGE = SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE \
        MESSAGEID=? AND DESTINATION=?
    MARK_MESSAGE = UPDATE JMS_MESSAGES SET (TXID, TXOP) VALUES(?, ?) WHERE \
        MESSAGEID=? AND DESTINATION=?
    DELETE_ALL_MESSAGE_WITH_TX = DELETE FROM JMS_MESSAGES WHERE TXID=?
    DELETE_TX = DELETE FROM JMS_TRANSACTIONS WHERE TXID = ?
    DELETE_MARKED_MESSAGES = DELETE FROM JMS_MESSAGES WHERE TXID=? AND TXOP=?
```

```
DELETE_MESSAGE = DELETE FROM JMS_MESSAGES WHERE MESSAGEID=? AND
DESTINATION=?
CREATE_MESSAGE_TABLE = CREATE TABLE JMS_MESSAGES ( MESSAGEID INTEGER NOT
NULL, \
DESTINATION VARCHAR(50) NOT NULL, TXID INTEGER, TXOP CHAR(1), \
MESSAGEBLOB OBJECT, PRIMARY KEY (MESSAGEID, DESTINATION) )
CREATE_TX_TABLE = CREATE TABLE JMS_TRANSACTIONS ( TXID INTEGER )
</attribute>
```

Destination MBeans

This section can be used as your reference guide to the jbossmq-destinations-service.xml file. This uses the standard JBoss service XML format, which is used to configure MBeans in the JBoss server.

org.jboss.mq.server.jmx.Queue

The org.jboss.mq.server.jmx.Queue is used to define a Queue Destination on the JBossMQ server. The “name” attribute of the JMX object name of this MBean is used to determine the destination name. For example, if the JMX MBean begins with:

```
<mbean code="org.jboss.mq.server.jmx.Queue"
       name="jboss.mq.destination:service=Queue,name=testQueue">
```

Then, the JMX object name is “jboss.mq.destination:service=Queue,name=testQueue” and the name of the queue is “testQueue”. The configurable attributes are as follows:

- **DestinationManager:** The JMX object name of the Destination Manager configured for the server. This attribute should be setup via a <depends optional-attribute-name="DestinationManager"> XML tag.
- **SecurityManager:** The JMX object name of the Security Manager that is being used to validate client requests. This attribute should be setup via a <depends optional-attribute-name="SecurityManager"> XML tag.
- **SecurityConf:** An XML based description of an access control list that will be used by the SecurityManager to authorize client operations against the Destination. See Listing 6-11 for an example of the configuration.
- **JNDIName:** The location in JNDI to which the queue object will be bound. If this is not set it will default to “queue/queue-name”.

org.jboss.mq.server.jmx.Topic

The org.jboss.mq.server.jmx.Topic is used to define a Topic Destination on the JBossMQ server. The “name” attribute of the JMX object name of this MBean is used to determine the destination name. For example, if the JMX MBean begins with:

```
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jboss.mq.destination:service=Topic,name=testTopic">
```

Then, the JMX object name is “jboss.mq.destination:service=Topic,name=testTopic” and the name of the topic is “testTopic”. The configurable attributes are as follows:

- **DestinationManager**: The JMX object name of the Destination Manager configured for the server. This attribute should be setup via a <depends optional-attribute-name="DestinationManager"> XML tag.
- **SecurityManager**: The JMX object name of the Security Manager that is being used to validate client requests. This attribute should be setup via a <depends optional-attribute-name="SecurityManager"> XML tag.
- **SecurityConf**: An XML based description of an access control list that will be used by the SecurityManager to authorize client operations against the Destination. See Listing 6-11 for an example of the configuration.
- **JNDIName**: The location in JNDI to which the queue object will be bound. If this is not set it will default to “topic/*topic-name*”.

Destination Security Configuration

To apply role based Access Control List to a Destination, setup the list via the **SecurityConf** MBean attribute on the Destination MBeans. It should contain a “security” XML element with zero or more “role” sub elements. A sample configuration is show in Listing 6-11.

LISTING 6-11. Sample Destination Security Configuration

```
<attribute name="SecurityConf">
  <security>
    <role name="guest" read="false" write="true"/>
    <role name="durablesub" read="true" write="false" create="true"/>
  </security>
</attribute>
```

The configuration in Listing 6-11 sets up two roles. The first is a guest role that is allowed to only write to the destination. The second role is only allowed to read from the destination and create a durable subscription. The attributes that can be set on the role element are:

- **name**: The name of the role that you are adding a security permission for.
- **read**: set to true or false. False if not set. Set to true to allow any clients who have the named role to receive messages from the destination.
- **write**: set to true or false. False if not set. Set to true to allow any clients who have the named role to send or publish messages to the destination.
- **create**: set to true or false. False if not set. Set to true to allow any clients who have the named role to create durable subscriptions for the destination.

Administration Via JMX

JBossMQ statistics and several management functions are accessible via JMX. JMX can be accessed interactively via a Web Application or programmatically via the JMX API. It is recommended that you use the <http://localhost:8080/jmx-console> web application to get familiar with all the JBossMQ JMX MBeans running inside the server and how to invoke methods on those MBeans via the jmx-

console web application. This section will outline the most common runtime management tasks that administrators must perform.

Creating Queues At Runtime

Applications that require the dynamic creation of queues at runtime can use the Destination Manager's MBean createQueue method:

```
void createQueue(String name, String jndiLocation)
```

This method creates a queue with the given *name* and binds it in JNDI at the *jndiLocation*. Queues created via this method exist until the server is restarted.

To destroy a previously created Queue, you would issue a:

```
void destroyQueue(String name)
```

Creating Topics At Runtime

Applications that require the dynamic creation of topics at runtime can use the Destination Manager's MBean createTopic method:

```
void createTopic(String name, String jndiLocation)
```

This method creates a topic with the given *name* and binds it in JNDI at the *jndiLocation*. Topics created via this method exist until the server is restarted.

To destroy a previously created Topic, you would issue a:

```
void destroyTopic(String name)
```

Managing a JBossMQ User IDs at Runtime

The org.jboss.mq.sm.file.DynamicStateManager's MBean can be used to add and remove user ids and roles at runtime. To add a user id, you would use:

```
void addUser(String name, String password, String clientID)
```

This method creates a user id with the given *name* and *password* and configures him to have the given *clientID*.

To remove a previously created user id, you would call the following method:

```
void removeUser(String name)
```

To manage the roles that the user ids belong to, you would use the following set of methods to create roles, remove roles, add users to roles, and remove users from roles:

```
void addRole(String name)
void removeRole(String name)
void addUserToRole(String roleName, String user)
void removeUserFromRole(String roleName, String user)
```

Checking how many messages are on a Queue

The org.jboss.mq.server.jmx.Queue MBeans provides a QueueDepth attribute which reflects the current number of messages sitting the on the given queue.

Checking to see how the Message Cache is performing

The Message Cache exposes several attributes for you to monitor the performance of the Message Cache. The attributes that can be viewed via JMX are:

- **HardRefCacheSize:** The number of messages the Message Cache forcing to stay inn memory by using a hard reference.
- **SoftRefCacheSize:** The number of messages the Message Cache has persisted but is still lingering around in memory as soft references due to the garbage collector not being eager to free up space.
- **TotalCacheSize:** The total number of messages that are being managed by the Cache Manager.
- **CacheHits:** The number of times a message was requested and it was found to be in memory.
- **CacheMisses:** The number of times a message was requested and it was not found in memory so a read from persistent storage was required to retrieve the message.

Connectors on JBoss - The JCA Configuration and Architecture

This chapter discusses the JBoss server implementation of the J2EE Connector Architecture (JCA). JCA is a resource manager integration API whose goal is to standardize access to non-relational resources in the same way the JDBC API standardized access to relational data. The purpose of this chapter is to introduce the utility of the JCA APIs and then describe the architecture of JCA in JBoss 3.0.x.

JCA Overview

J2EE 1.3 contains a connector architecture (JCA) specification that allows for the integration of transacted and secure resource adaptors into a J2EE application server environment. The full JCA specification is available from the JCA home page here: <http://java.sun.com/j2ee/connector/>. The JCA specification describes the notion of such resource managers as Enterprise Information Systems (EIS). Examples of EIS systems include enterprise resource planning packages, mainframe transaction processing, non-Java legacy applications, etc.

The reason for focusing on EIS is primarily because the notions of transactions, security, and scalability are requirements in enterprise software systems. However, the JCA is applicable to any resource that needs to integrate into JBoss in a secure, scalable and transacted manner. In this introduction we will focus on resource adapters as a generic notion rather than something specific to the EIS environment.

The connector architecture defines a standard SPI (Service Provider Interface) for integrating the transaction, security and connection management facilities of an application server with those of a resource manager. The SPI defines the system level contract between the resource adaptor and the application server.

The connector architecture also defines a Common Client Interface (CCI) for accessing resources. The CCI is targeted at EIS development tools and other sophisticated users of integrated resources. The CCI provides a way to minimize the EIS specific code required by such tools. Typically J2EE developers will access a resource using such a tool, or a resource specific interface rather than using CCI directly. The reason is that the CCI is not a type specific API. To be used effectively it must be used in conjunction with metadata that describes how to map from the generic CCI API to the resource manager specific data types used internally by the resource manager.

The purpose of the connector architecture is to enable a resource vendor to provide a standard adaptor for its product. A resource adaptor is a system-level software driver that is used by a Java application to connect to resource. The resource adaptor plugs into an application server and provides connectivity between the resource manager, the application server, and the enterprise application. A resource vendor need only implement a JCA compliant adaptor once to allow use of the resource manager in any JCA capable application server.

An application server vendor extends its architecture once to support the connector architecture and is then assured of seamless connectivity to multiple resource managers. Likewise, a resource manager vendor provides one standard resource adaptor and it has the capability to plug in to any application server that supports the connector architecture.

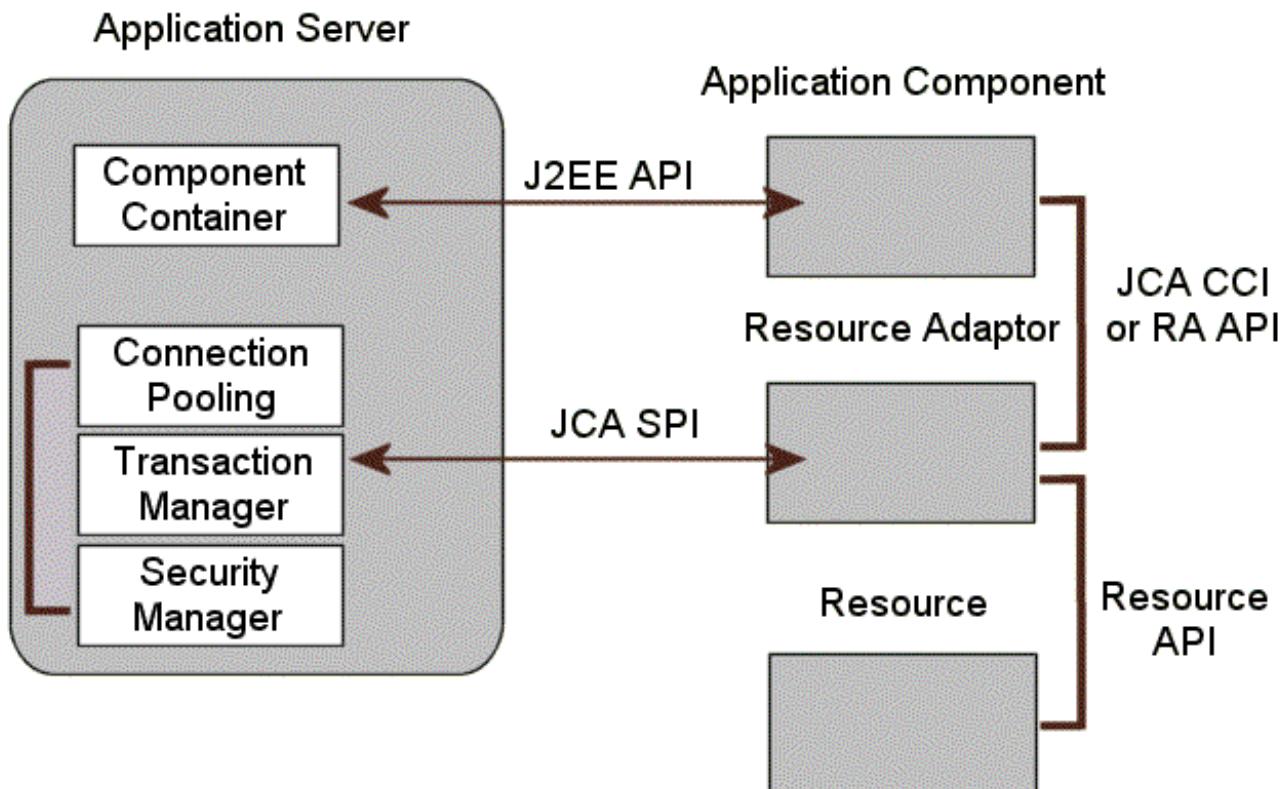


FIGURE 7-1. The relationship between a J2EE application server and a JCA resource adaptor

Figure 7-1 illustrates that the application server is extended to provide support for the JCA SPI to allow a resource adaptor to integrate with the server connection pooling, transaction management and security management facilities. This integration API defines a system contract that consists of:

- Connection management: a contract that allows the application server to pool resource connections. The purpose of the pool management is to allow for scalability. Resource connections are typically expensive objects to create and pooling them allows for more effective reuse and management.
- Transaction Management: a contract that allows the application server transaction manager to manage transactions that engage resource managers.
- Security Management: a contract that enables secured access to resource managers.

The resource adaptor implements the resource manager side of the system contract. This entails using the application server connection pooling, providing transaction resource information and using the security integration information. The resource adaptor also exposes the resource manager to the application server components. This can be done using the CCI and/or a resource adaptor specific API.

The application component integrates into the application server using a standard J2EE container to component contract. For an EJB component this contract is defined by the EJB specification. The application component interacts with the resource adaptor in the same way as it would with any other standard resource factory, for example, a javax.sql.DataSource JDBC resource factory. The only difference with a JCA resource adaptor is that the client has the option of using the resource adaptor independent CCI API if the resource adaptor supports this.

Figure 6.0 of the JCA 1.0 specification illustrates the relationship between the JCA architecture participants in terms of how they relate to the JCA SPI, CCI and JTA packages. This figure is recreated here as Figure 7-2.

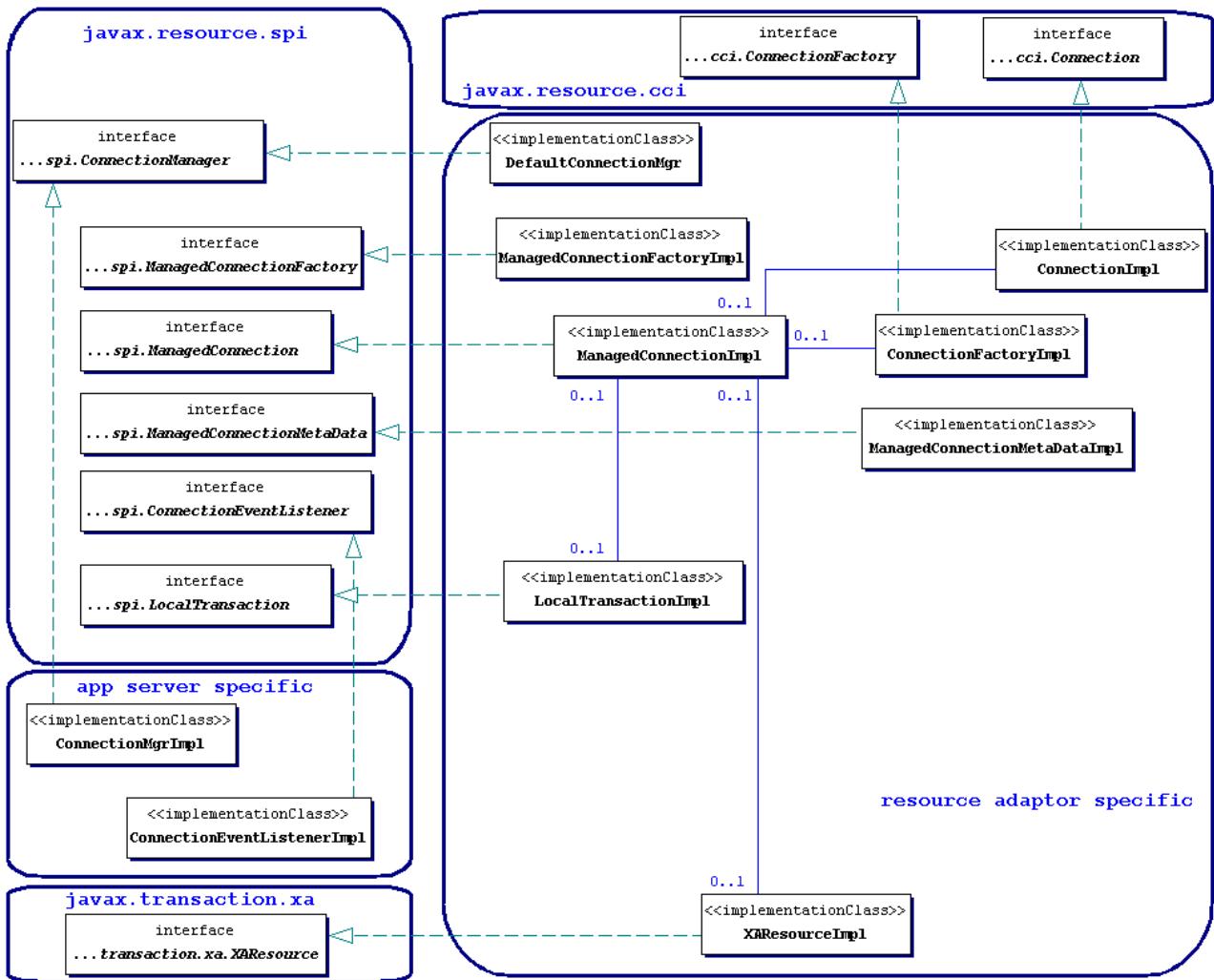


FIGURE 7-2. The JCA 1.0 specification class diagram for the connection management architecture.

The JBossCX architecture provides the implementation of the application server specific classes. Figure 7-2 shows that this comes down to the implementation of the `javax.resource.spi.ConnectionManager` and `javax.resource.spi.ConnectionEventListener` interfaces. The key aspects of this implementation are discussed in the following section on the JBossCX architecture.

An Overview of the JBossCX Architecture

The JBossCX framework provides the application server architecture extension required for the use of JCA resource adaptors. This is primarily a connection pooling and management extension along with a number of MBeans for loading resource adaptors into the JBoss server. Figure 7-3 expands the generic view given by Figure 7-2 to illustrate how the JBoss JCA layer implements the application server specific extension along with an example file system resource adaptor that we will look at later in this chapter.

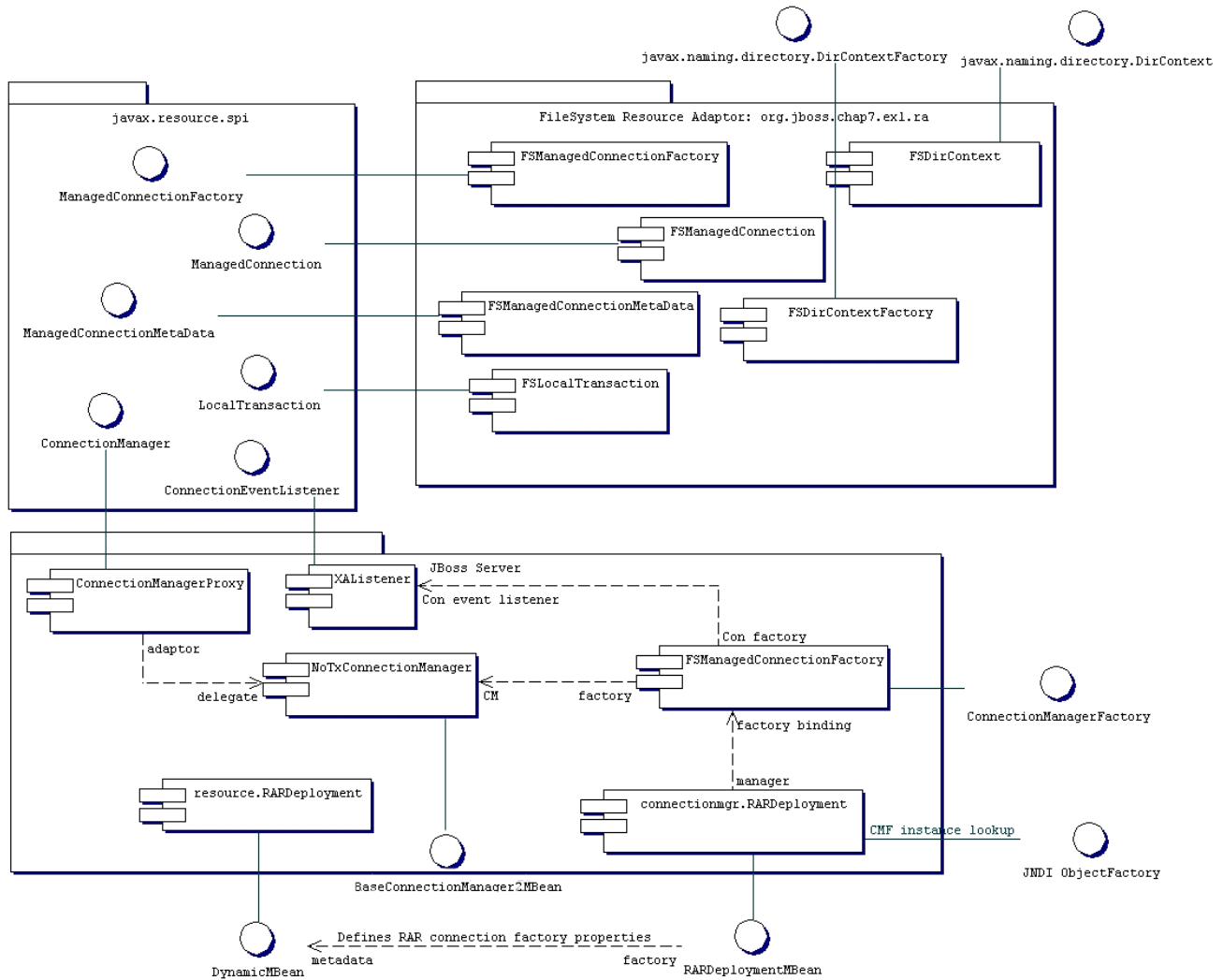


FIGURE 7-3. The JBoss JCA implementation components

There are three coupled MBeans that make up a RAR deployment. These are the [org.jboss.resource.RARDeployment](#), [org.jboss.resource.connectionmanager.RARDeployment](#), and [org.jboss.resource.connectionmanager.BaseConnectionManager2](#). The [org.jboss.resource.RARDeployment](#) is simply an encapsulation of the metadata of a RAR META-INF/ra.xml descriptor. It

exposes this information as a [DynamicMBean](#) simply to make it available to the [org.jboss.resource.connectionmanager.RARDeployment](#) MBean.

The [RARDeployer](#) service handles the deployment of archives files containing resource adaptors (RARs). It creates the [org.jboss.resource.RARDeployment](#) MBeans when a RAR file is deployed. Deploying the RAR file is the first step in making the resource adaptor available to application components. For each deployed RAR, one or more connection factories must be configured and bound into JNDI. This task performed using a JBoss service descriptor that sets up a [org.jboss.resource.connectionmanager.BaseConnectionManager2](#) MBean implementation with a [org.jboss.resource.connectionmgr.RARDeployment](#) dependent.

BaseConnectionManager2 MBean

The [org.jboss.resource.connectionmanager.BaseConnectionManager2](#) MBean is a base class for the various types of connection managers required by the JCA spec. Subclasses include [org.jboss.resource.connectionmanager.NoTxConnectionManager](#), [org.jboss.resource.connectionmanager.LocalTxConnectionManager](#) and [org.jboss.resource.connectionmanager.XATxConnectionManager](#), and these correspond to resource adaptors that support no transactions, local transaction and XA transaction respectively. You choose which subclass to use based on the type of transaction semantics you want, provided the JCA resource adaptor supports the corresponding transaction capability.

The common attributes supported by the [BaseConnectionManager2](#) MBean are:

- **ManagedConnectionFactoryName:** This specifies the [ObjectName](#) of the MBean that creates [javax.resource.spi.ManagedConnectionFactory](#) instances. Normally this is configured as embedded mbean in a depends element rather than a separate mbean reference using the [org.jboss.resource.connectionmanager.RARDeployment](#) MBean. The MBean must provide an operation with the following signature:
`javax.resource.spi.ManagedConnectionFactory startManagedConnectionFactory(javax.resource.spi.ConnectionManager)`
- **ManagedConnectionPool:** This specifies the [ObjectName](#) of the MBean representing the pool for this connection manager. The MBean must have an [ManagedConnectionPool](#) attribute that is an implementation of the [org.jboss.resource.connectionmanager.ManagedConnectionPool](#) interface. Normally it will be an embedded mbean in a depends tag rather than an [ObjectName](#) reference to an existing mbean. The default MBean for use is the [org.jboss.resource.connectionmanager.JBossManagedConnectionPool](#). Its configurable attribute are discussed below.
- **CachedConnectionManager:** This specifies the [ObjectName](#) of the [org.jboss.resource.connectionmanager.CachedConnectionManager](#) MBean implementation used by the connection manager. Normally this will be a specified using a depends tag with the [ObjectName](#) of the unique [CachedConnectionManager](#) for the server. The name “jboss.jca:service=CachedConnectionManager” is the standard setting to use.
- **SecurityDomainJndiName:** This specifies the JNDI name of the security domain to use for authentication and authorization of resource connections. This is typically of the form “java:/jaas/<domain>” where the <domain> value is the name of an entry in the conf/login-config.xml JAAS

login module configuration file. This defines which JAAS login modules execute to perform authentication. See Chapter 8 for more information on the security settings.

- **JaasSecurityManagerService**: This is the ObjectName of the security manager service. This should be set to the security manager MBean name as defined in the conf/jboss-service.xml descriptor, and currently this is “jboss.security:service=JaasSecurityManager”. This attribute will likely be removed in the future.

RARDeployment MBean

The [org.jboss.resource.connectionmanager.RARDeployment](#) MBean manages configuration and instantiation [ManagedConnectionFactory](#) instance. It does this using the resource adaptor metadata settings from the RAR META-INF/ra.xml descriptor along with the [RARDeployment](#) attributes. The configurable attributes are:

- **OldRarDeployment**: This is the [ObjectName](#) of the [org.jboss.resource.RarDeployment](#) MBean that contains the resource adaptor metadata. The form of this name is “jboss.jca:service=RARDerployment,name=<ra-display-name>” where the <ra-display-name> is the ra.xml descriptor display-name attribute value. This is created by the [RARDeployer](#) when it deploys a RAR file. This attribute will likely be removed in the future.
- **ManagedConnectionFactoryProperties**: This is a collection of (name, type, value) tripples that define attributes of the [ManagedConnectionFactory](#) instance. Therefore, the names of the attributes depend on the resource adaptor [ManagedConnectionFactory](#) instance. The structure of the content of this attribute is:

```
<properties>
  <config-property>
    <config-property-name>Attr0Name</config-property-name>
    <config-property-type>Attr0Type</config-property-type>
    <config-property-value>Attr0Value</config-property-value>
  </config-property>
  <config-property>
    <config-property-name>Attr1Name</config-property-name>
    <config-property-type>Attr2Type</config-property-type>
    <config-property-value>Attr2Value</config-property-value>
  </config-property>
  ...
</properties>
```

where AttrXName is the Xth attribute name, AttrXType is the fully qualified Java type of the attribute, and AttrXValue is the string representation of the value. The conversion from string to AttrXType is done using the [java.beans.PropertyEditor](#) class for the AttrXType.

- **JndiName**: This is the JNDI name under which the will be made available. Clients of the resource adaptor use this name to obtain either the [javax.resource.cci.ConnectionFactory](#) or resource adaptor specific connection factory. The full JNDI name will be “java:/<JndiName>” meaning that the JndiName attribute value will be prefixed with “java:/”. This prevents use of the connection factory outside of the JBoss server VM. In the future this restriction may be configurable.

JBossManagedConnectionPool MBean

The [org.jboss.resource.connectionmanager.JBossManagedConnectionPool](#) MBean is a connection pooling MBean. It is typically used as the embedded MBean value of the [BaseConnectionManager2](#) ManagedConnectionPool attribute. When you setup a connection manager MBean you typically embed the pool configuration in the connection manager descriptor. The configurable attributes of the [JBossManagedConnectionPool](#) are:

- **MinSize:** This attribute indicates the minimum number of connections this pool should hold. These are not created until a Subject is known from a request for a connection. MinSize connections will be created for each sub-pool.
- **MaxSize:** This attribute indicates the maximum number of connections for a pool. No more than MaxSize connections will be created in each sub-pool.
- **BlockingTimeoutMillis:** This attribute indicates the maximum time to blockwhile waiting for a connection before throwing an exception. Note that this blocks only while waiting for a permit for a connection, and will never throw an exception if creating a new connection takes an inordinately long time.
- **IdleTimeoutMinutes:** This attribute indicates the maximum time a connection may be idle before being closed. The actual maximum time depends also on the idle remover thread scan time, which is 1/2 the smallest idle timeout of any pool.
- **Criteria:** This attribute indicates if the JAAS [javax.security.auth.Subject](#) from security domain associated with the connection, or app supplied parameters (such as from `getConnection(user, pw)`) are used to distinguish connections in the pool. The allowed values are:
 - ByContainerAndApplication (use both),
 - ByContainer (use Subject),
 - ByApplication (use app supplied params only),
 - ByNothing (all connections are equivalent, usually if adapter supports reauthentication)

CachedConnectionManager MBean

The [org.jboss.resource.connectionmanager.CachedConnectionManager](#) MBean manages associations between meta-aware objects (those accessed through interceptor chains) and connection handles, as well as between user transactions and connection handles. Normally there should only be one such MBean, and this is configured in the core jboss-service.xml descriptor. It is used by [org.jboss.resource.connectionmanager.CachedConnectionInterceptor](#), JTA [javax.transaction.UserTransaction](#) implementation, and all [BaseConnectionManager2](#) instances. The [CachedConnectionManager](#) MBean has no configurable attributes.

A Sample Skeleton JCA Resource Adaptor

To conclude our discussion of the JBoss JCA framework we will create and deploy a single non-transacted resource adaptor that simply provides a skeleton implementation that stubs out the required interfaces and logs all method calls. We will not discuss the details of the requirements of a resource adaptor provider as these are discussed in detail in the JCA specification. The purpose of the adaptor

is to demonstrate the steps required to create and deploy a RAR in JBoss, and to see how JBoss interacts with the adaptor.

The adaptor we will create could be used as the starting point for a non-transacted file system adaptor. The source to the example adaptor can be found in the src/main/org/jboss/chap7/ex1 directory of the book examples. A class diagram that shows the mapping from the required javax.resource.spi interfaces to the resource adaptor implementation is given in Figure 7-4.

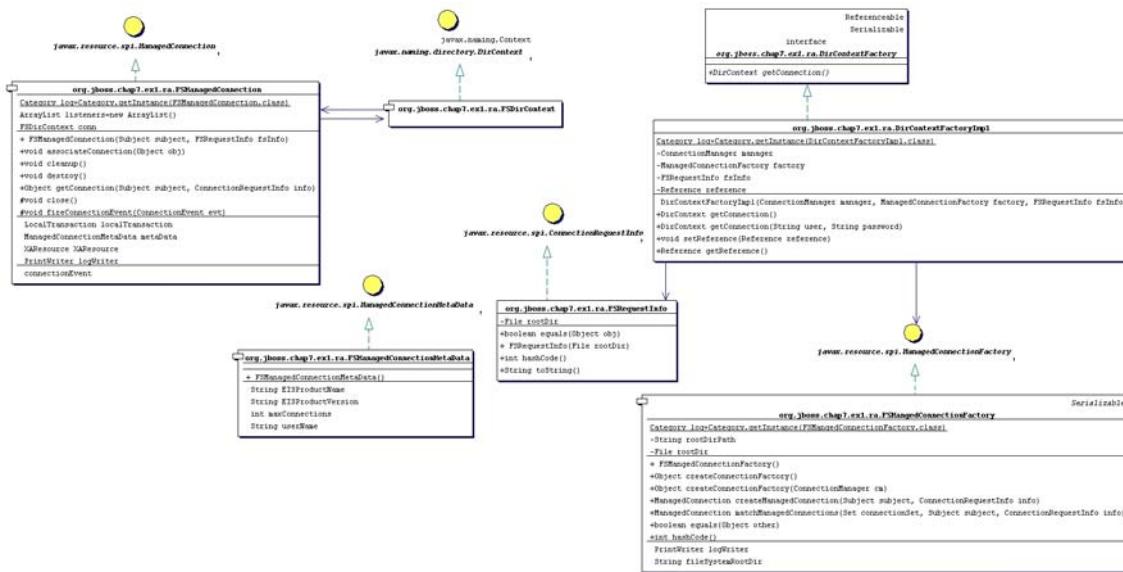


FIGURE 7-4. The file system RAR class diagram

We will build the adaptor, deploy it to the JBoss server and then run an example client against an EJB that uses the resource adaptor to demonstrate the basic steps in a complete context. We'll then take a look at the JBoss server log to see how the JBoss JCA framework interacts with the resource adaptor to help you better understand the components in the JCA system level contract.

To build the example and deploy the RAR to the JBoss server deploy/lib directory, execute the following ant command in the book examples directory:

```

examples 917>ant -Dchap=7 build-chap
Buildfile: build.xml

...

chap7-ex1-rar:
    [jar] Building jar: /JBossDocs/examples/output/chap7/chap7-ex1.rar

prepare:

chap7-ex1-jar:


```

```
[jar] Building jar: /JBossDocs/examples/output/chap7/chap7-ex1.jar
[copy] Copying 1 file to /tmp/jboss-3.0.1RC1/server/default/deploy
[copy] Copying 1 file to /tmp/jboss-3.0.1RC1/server/default/deploy

BUILD SUCCESSFUL

Total time: 6 seconds
```

The deployed files include a chap7-ex1.sar and a notxfs-service.xml service descriptor. The example resource adaptor deployment descriptor is shown in Listing 7-1 while the connection manager MBeans service descriptor is shown in Listing 7-2.

LISTING 7-1. The nontransactional file system resource adaptor deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connector PUBLIC
  "-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
  "http://java.sun.com/dtd/connector_1_0.dtd">

<connector>
  <display-name>File System Adapter</display-name>
  <vendor-name>JBoss Group</vendor-name>
  <spec-version>1.0</spec-version>
  <version>1.0</version>
  <eis-type>FileSystem</eis-type>
  <license>
    <description>GPL</description>
    <license-required>false</license-required>
  </license>
  <resourceadapter>
    <managedconnectionfactory-
class>org.jboss.chap7.ex1.ra.FSMangedConnectionFactory
    </managedconnectionfactory-class>
    <connectionfactory-interface>org.jboss.chap7.ex1.ra.DirContextFactory
    </connectionfactory-interface>
    <connectionfactory-impl-class>org.jboss.chap7.ex1.ra.DirContextFactoryImpl
    </connectionfactory-impl-class>
    <connection-interface>javax.naming.directory.DirContext
    </connection-interface>
    <connection-impl-class>org.jboss.chap7.ex1.ra.FSDirContext
    </connection-impl-class>
    <transaction-support>NoTransaction</transaction-support>
    <config-property>
      <config-property-name>FileSystemRootDir</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-value>/tmp/db/fs_store</config-property-value>
    </config-property>
    <config-property>
      <config-property-name>UserName</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-value></config-property-value>
    </config-property>
    <config-property>
```

```
<config-property-name>Password</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value></config-property-value>
</config-property>
<authentication-mechanism>
    <authentication-mechanism-type>BasicPassword</authentication-mechanism-
type>
    <credential-interface>javax.resource.security.PasswordCredential</
credential-interface>
    </authentication-mechanism>
    <reauthentication-support>true</reauthentication-support>
    <security-permission>
        <description>Read/Write access is required to the contents of
the FileSystemRootDir</description>
        <security-permission-spec>permission java.io.FilePermission
        "/tmp/db/fs_store/*", "read,write";</security-permission-spec>
    </security-permission>
</resourceadapter>
</connector>
```

LISTING 7-2. The nontransactional file system resource adaptor MBeans service descriptor.

```
<!-- The non-transaction FileSystem resource adaptor service configuration
-->
<server>
    <mbean code="org.jboss.resource.connectionmanager.NoTxConnectionManager"
           name="jboss.jca:service=NoTxCM,name=filesystem">
        <depends>jboss.jca:service=RARDeployer</depends>
        <depends optional-attribute-name="ManagedConnectionFactoryName">
            <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
                   name="jboss.jca:service=NoTxFS,name=filesystem">
                <depends optional-attribute-name="OldRarDeployment">
                    jboss.jca:service=RARDeployment, name=File System Adapter
                </depends>

                <attribute name="ManagedConnectionFactoryProperties">
                    <properties>
                        <config-property>
                            <config-property-name>FileSystemRootDir</config-property-name>
                            <config-property-type>java.lang.String</config-property-type>
                            <config-property-value>/tmp/db/fs_store</config-property-
value>
                        </config-property>
                    </properties>
                </attribute>
                <attribute name="JndiName">NoTransFS</attribute>
            </mbean>
        </depends>

        <depends optional-attribute-name="ManagedConnectionPool">
            <mbean
code="org.jboss.resource.connectionmanager.JBossManagedConnectionPool"
           name="jboss.jca:service=NoTxPool,name=filesystem">
                <attribute name="MinSize">0</attribute>
```

```
<attribute name="MaxSize">50</attribute>
<attribute name="BlockingTimeoutMillis">5000</attribute>
<attribute name="IdleTimeoutMinutes">15</attribute>
<attribute name="Criteria">ByContainer</attribute>
</mbean>
</depends>
<depends optional-attribute-name="CachedConnectionManager">
jboss.jca:service=CachedConnectionManager
</depends>
<depends optional-attribute-name="JaasSecurityManagerService">
jboss.security:service=JaasSecurityManager
</depends>
</mbean>
</server>
```

The key items in the resource adaptor deployment descriptor are highlighted in bold. These define the classes of the resource adaptor, and the elements are:

- **display-name**: Recall from our discussion of the connection manager factory MBeans that the association between the factory and the resource adaptor classes was done via a RARDeployment DynamicMBean located by name. The name was based on the display-name value found in the ra.xml descriptor. Here the name is “File System Adaptor” and we will use it in the connection manager service descriptor.
- **managedconnectionfactory-class**: The implementation of the javax.resource.spi.ManagedConnectionFactory interface, org.jboss.chap7.ex1.ra.FSMangedConnectionFactory
- **connectionfactory-interface**: The interface that clients will obtain when they lookup the connection factory instance from JNDI, here a proprietary resource adaptor value, org.jboss.chap7.ex1.ra.DirContextFactory
- **connectionfactory-impl-class**: The class that provides the implementation of the connectionfactory-interface, org.jboss.chap7.ex1.ra.DirContextFactoryImpl
- **connection-interface**: The interface for the connections returned by the resource adaptor connection factory, here the JNDI javax.naming.directory.DirContext interface.
- **connection-impl-class**: The class that provides the connection-interface implementation, org.jboss.chap7.ex1.ra.FSDirContext
- **transaction-support**: The level of transaction support, here defined as NoTransaction, meaning the file system resource adaptor does not do transactional work.

See the JCA 1.0 spec, or the book “J2EE Connector Architecture and Enterprise Application Integration” by Sharma, Stearns and Ng for the full details of the ra.xml descriptor elements.

The RAR classes and deployment descriptor only define a resource adaptor. To use the resource adaptor it must be integrated into the JBoss application server. As we have discussed this is done with a connection factory MBeans. The various mbean element tags are highlighted in bold in Listing 7-2, and the following notes apply.

- The main MBean is the org.jboss.resource.connectionmanager.NoTxConnectionManager. This is the subclass of BaseConnectionManager2 that provides no support for transactions, and this was selected because of the lack of transaction support in the resource adaptor. Its name is

ObjectName “jboss.jca:service=NoTxCM,name=filesystem”. The configuration contains two nested mbean elements for the setup of the connection factory binding and connection pooling.

- The first nested mbean element associated with the ManagedConnectionFactoryName attribute is the org.jboss.resource.connectionmanager.RARDeployment MBean. This sets up the file system connection manager factory binding in JNDI. Its attributes are:
 - OldRarDeployment, specifies which resource adaptor the connection factory is associated with. This is done using the org.jboss.resource.RARDeployment ObjectName pattern discussed previously. The name “jboss.jca:service=RARDeployment,name=File System Adaptor” ties the connection manager to the RAR descriptor of because of Listing 7-1 the “File System Adaptor” link from the name property of the ObjectName to the display-name element of the ra.xml descriptor.
 - The ManagedConnectionFactoryProperties attribute provides non-default settings to apply to the resource adaptor connection factory. Here the FileSystemRootDir attribute is being set to “/tmp/db/fs_store”.
 - JndiName=NoTransFS, declares that the adaptor factory will be bound into JNDI under the name "java:/NoTransFS".
- The next nested mbean element associated with the ManagedConnectionPool attribute is the org.jboss.resource.connectionmanager.JBossManagedConnectionPool. This configures the manner in which connection will be pooled.

You have already deployed this RAR and connection manager to the JBoss server. Now startup the JBoss server and the console should show output similar to the following a little before the final startup time line:

```
15:13:05,265 INFO [MainDeployer] Starting deployment of package: .../notxfs-
service.xml
15:13:05,359 WARN [ServiceController] jboss.jca:service=NoTxFS,name=filesystem
does not implement any Service methods
15:13:05,359 INFO [JBossManagedConnectionPool] Creating
15:13:05,359 INFO [JBossManagedConnectionPool] Created
15:13:05,359 INFO [JBossManagedConnectionPool] Starting
15:13:05,359 INFO [JBossManagedConnectionPool] Started
15:13:05,375 INFO [MainDeployer] Deployed package: .../notxfs-service.xml
...
15:13:05,593 INFO [MainDeployer] Starting deployment of package: ...chap7-
ex1.rar
15:13:05,703 INFO [RARMetaData] License terms present. See deployment
descriptor.
15:13:05,781 WARN [ServiceController] jboss.jca:service=RARDeployment,name=File
System Adapter does not implement any Service methods
15:13:05,781 WARN [ServiceController] jboss.jca:service=NoTxFS,name=filesystem
does not implement any Service methods
15:13:05,781 INFO [NoTxConnectionManager] Creating
15:13:05,843 INFO [NoTxConnectionManager] Created
15:13:05,906 INFO [NoTxConnectionManager] Starting
15:13:06,000 INFO [NoTransFS] Bound connection factory for resource adapter 'Fi
le System Adapter' to JNDI name 'java:/NoTransFS'
15:13:06,015 INFO [NoTxConnectionManager] Started
15:13:06,015 INFO [MainDeployer] Deployed package: .../deploy/chap7-ex1.rar
```

This indicates that the resource adaptor has been successfully deployed and its connection factory has been bound into JNDI under the name “java:/NoTransFS”. You can ignore the warning messages about the RARDeployment MBeans not implementing any Service methods. Its fine as they are not used as JBoss services, just standard MBeans.

Now we want to test access of the resource adaptor by a J2EE component. To do this we have created a trivial stateless session bean that has a single method called echo. Inside of the echo method the EJB accesses the resource adaptor connection factory, creates a connection, and then immediately closes the connection. The echo method code is shown in Listing 7-3.

LISTING 7-3. The stateless session bean echo method code which shows the access of the resource adaptor connection factory.

```
public String echo(String arg)
{
    log.debug("echo, arg="+arg);
    try
    {
        InitialContext iniCtx = new InitialContext();
        Context enc = (Context) iniCtx.lookup("java:comp/env");
        Object ref = enc.lookup("ra/DirContextFactory");
        log.debug("echo, ra/DirContextFactory="+ref);
        DirContextFactory dcf = (DirContextFactory) ref;
        log.debug("echo, found dcf="+dcf);
        DirContext dc = dcf.getConnection();
        log.debug("echo, lookup dc="+dc);
        dc.close();
    }
    catch(NamingException e)
    {
        log.error("Failed during JNDI access", e);
    }
    return arg;
}
```

The EJB is not using the CCI interface to access the resource adaptor. Rather, it is using the resource adaptor specific API based on the proprietary DirContextFactory interface that returns a JNDI DirContext object as the connection object. The example EJB is simply exercising the system contract layer by looking up the resource adaptor connection factory, creating a connection to the resource and closing the connection. The EJB does not actually do anything with the connection, as this would only exercise the resource adaptor implementation since this is a non-transactional resource.

Run the test client which calls the EchoBean.echo method by running ant as follows from the examples directory:

```
examples 920>ant -Dchap=7 -Dex=1 run-example
Buildfile: build.xml

...
run-example1:
```

```
[copy] Copying 1 file to /tmp/jboss-3.0.1RC1/server/default/deploy
[echo] Waiting for deploy...
[java] Created Echo
[java] Echo.echo('Hello') = Hello
```

Now let's look at the output that has been logged by the resource adaptor to understand the interaction between the adaptor and the JBoss JCA layer. The output is in the server/default/log/server.log file of the JBoss server distribution. We'll summarize the events seen in the log using a sequence diagram.

Those are the steps involved with making the resource adaptor connection factory available to application server components. The remaining log messages are the result of the example client invoking the EchoBean.echo method and this method's interaction with the resource adaptor connection factory. Figure 7-5 is a sequence diagram that summarizes the events that occur when the EchoBean accesses the resource adaptor connection factory from JNDI and creates a connection.

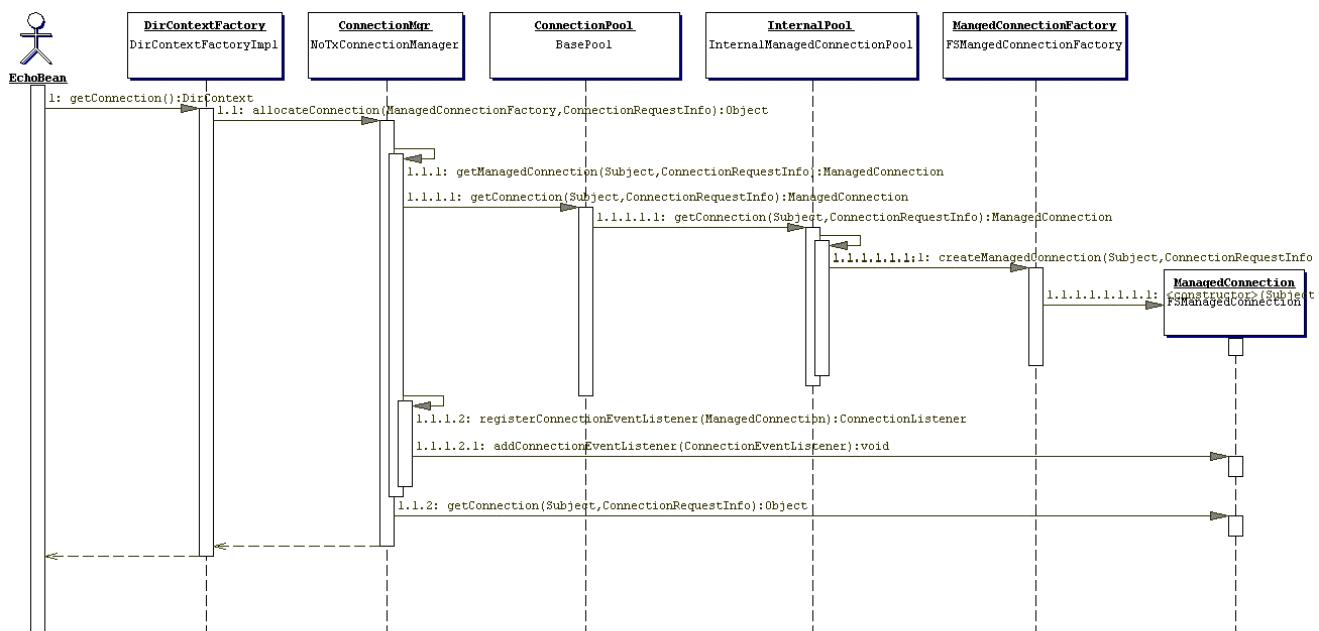


FIGURE 7-5. A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor that result when the EchoBean accesses the resource adaptor connection factory.

The starting point is the client's invocation of the EchoBean.echo method. For the sake of conciseness of the diagram, the client is shown directly invoking the EchoBean.echo method when in reality the JBoss EJB container handles the invocation. There are three distinct interactions between the Echo-Bean and the resource adaptor; the lookup of the connection factory, the creation of a connection, and the close of the connection.

The lookup of the resource adaptor connection factory is illustrated by the 1.1 sequences of events. The events are:

- 1, the echo method invokes the `getConnection` method on the resource adaptor connection factory obtained from the JNDI lookup on the “`java:comp/env/ra/DirContextFactory`” name which is a link to the “`java:/NoTransFS`” location.
- 1.1, the `DirContextFactoryImpl` class asks its associated `ConnectionManager` to allocate a connection. It passes in the `ManagedConnectionFactory` and `FSRequestInfo` that were associated with the `DirContextFactoryImpl` during its construction.
- 1.1.1, the `ConnectionManager` invokes its `getManagedConnection` method with the current `Subject` and `FSRequestInfo`.
- 1.1.1.1, the `ConnectionManager` asks its object pool for a connection object. The `JBossManagedConnectionPool$BasePool` is get the key for the connection and then asks the matching `InternalPool` for a connection.
- 1.1.1.1.1, Since no connections have been created the pool must create a new connection. This is done by requesting a new managed connection from the `ManagedConnectionFactory`. The `Subject` associated with the pool as well as the `FSRequestInfo` data are passed as arguments to the `createManagedConnection` method invocation.
- 1.1.1.1.1.1, the `FSManagedConnectionFactory` creates a new `FSManagedConnection` instance and passes in the `Subject` and `FSRequestInfo` data.
- 1.1.1.2, a `javax.resource.spi.ConnectionListener` instance is created. The type of listener created is based on the type of `ConnectionManager`. In this case it is an `org.jboss.resource.connectionmgr.BaseConnectionManager2$NoTransactionListener` instance.
- 1.1.1.2.1, the listener registers as a `javax.resource.spi.ConnectionEventListener` with the `ManagedConnection` instance created in 1.2.1.1.
- 1.1.2, the `ManagedConnection` is asked for the underlying resource manager connection. The `Subject` and `FSRequestInfo` data are passed as arguments to the `getConnection` method invocation.
- The resulting connection object is cast to a `javax.naming.directory.DirContext` instance since this is the public interface defined by the resource adaptor.
- After the EchoBean has obtained the `DirContext` for the resource adaptor, it simply closes the connection to indicate its interaction with the resource manager is complete.

This concludes the resource adaptor example. Our investigation into the interaction between the JBossCX layer and a trivial resource adaptor should give you sufficient understanding of the steps required to configure any resource adaptor. The example adaptor can also serve as a starting point for the creation of your own custom resource adaptors if you need to integrate non-JDBC resources into the JBoss server environment.

Example Configurations

Example configurations of many third-party JDBC datasources is included in the `JBOSS_DIST/docs/examples/jca` directory. Current example configurations include:

- `db2-service.xml`
- `firebird-service.xml`
- `hsqldb-service.xml`
- `informix-service.xml`

- informix-xa-service.xml
- jdatastore-service.xml
- lido-versant-service.xml
- msaccess-service.xml
- mssql-service.xml
- mssql-xa-service.xml
- mysql-service.xml
- oracle-service.xml
- oracle-xa-service.xml
- postgres-service.xml
- sapdb-service.xml
- solid-service.xml
- sybase-service.xml

Security on JBoss - J2EE Security Configuration and Architecture

Security is a fundamental part of any enterprise application. You need to be able to restrict who is allowed to access your applications and control what operations application users may perform. The J2EE specifications define a simple role-based security model for EJBs and Web components. The JBoss component framework that handles security is the JBossSX extension framework. The JBossSX security extension provides support for both the role-based declarative J2EE security model as well as integration of custom security via a security proxy layer. The default implementation of the declarative security model is based on Java Authentication and Authorization Service (JAAS) login modules and subjects. The security proxy layer allows custom security that cannot be described using the declarative model to be added to an EJB in a way that is independent of the EJB business object. Before getting into the JBoss security implementation details, we will review the EJB 2.0, the Servlet 2.2 specification security model, and JAAS to establish the foundation for these details.

J2EE Declarative Security Overview

The security model advocated by the J2EE specification is a declarative model. It is declarative in that you describe the security roles and permissions using a standard XML descriptor rather than embedding security into your business component. This isolates security from business-level code because security tends to be a more a function of where the component is deployed, rather than an inherent aspect of the component's business logic. For example, consider an ATM component that is to be used to access a bank account. The security requirements, roles and permissions will vary independent of how one accesses the bank account based on what bank is managing the account, where the ATM machine is deployed, and so on.

Securing a J2EE application is based on the specification of the application security requirements via the standard J2EE deployment descriptors. You secure access to EJBs and Web components in an enterprise application by using the ejb-jar.xml and web.xml deployment descriptors. Figure 8-1 and

Figure 8-2 illustrate the security-related elements in the EJB 2.0 and Servlet 2.2 deployment descriptors, respectively.

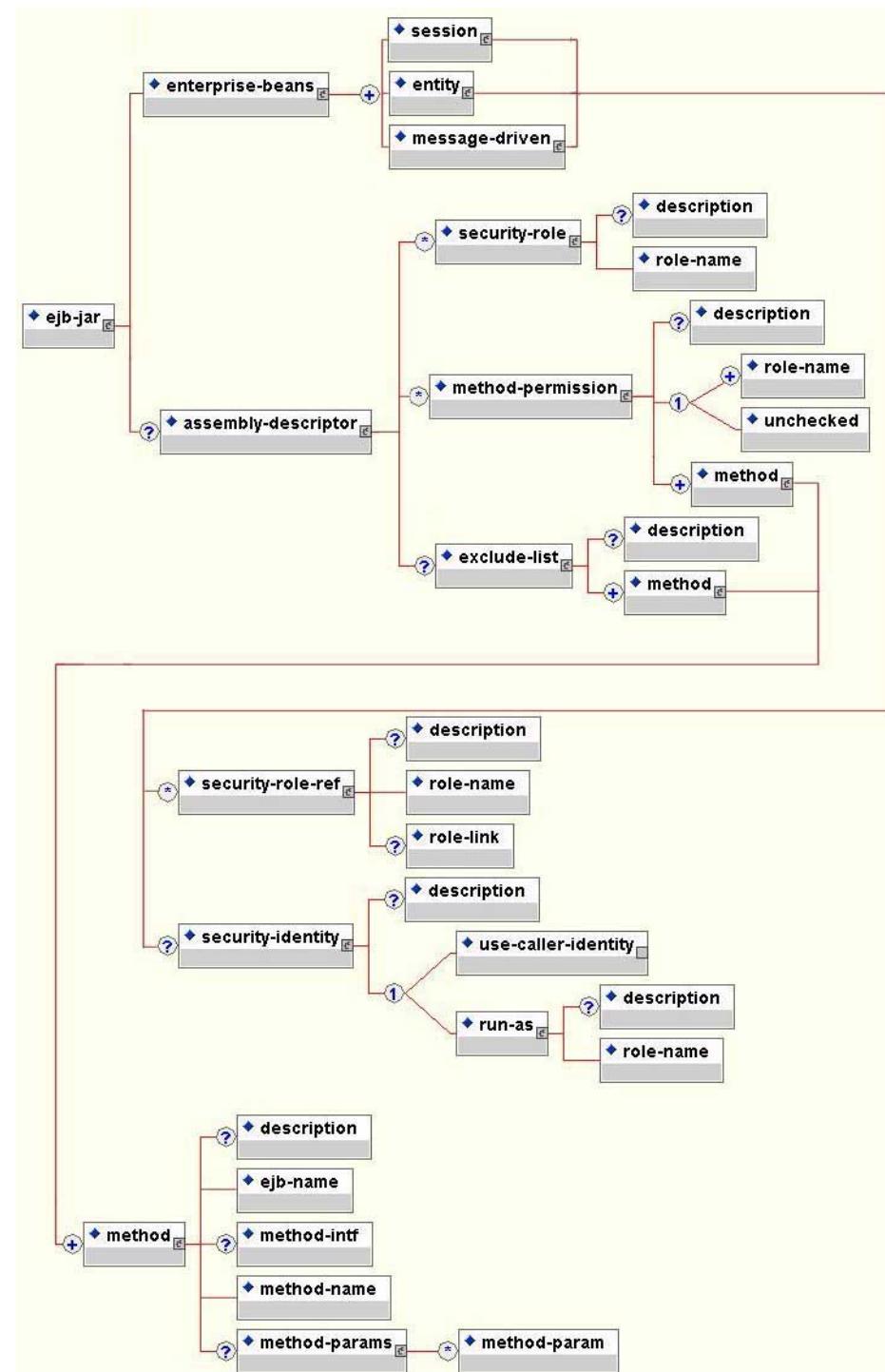


FIGURE 8-1. A subset of the EJB 2.0 deployment descriptor content model that shows the security related elements.

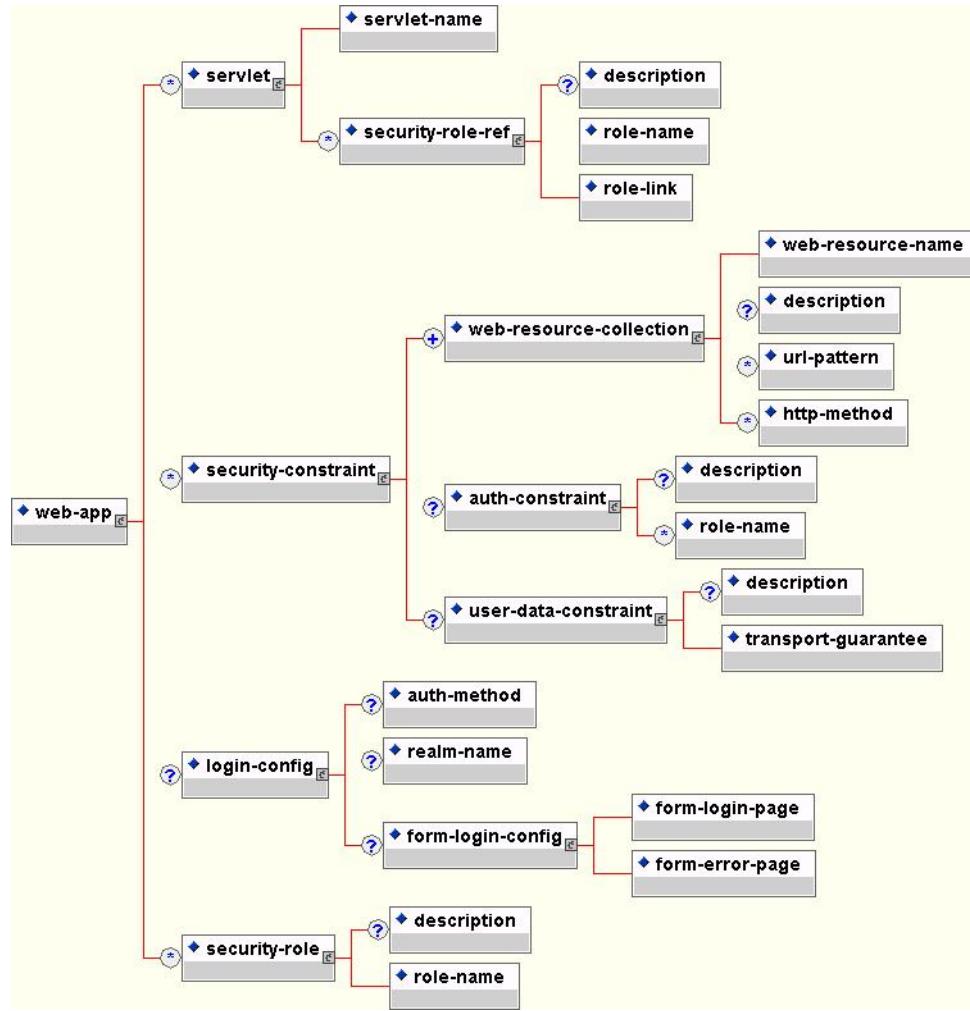


FIGURE 8-2. A subset of the Servlet 2.2 deployment descriptor content model that shows the security related elements.

The purpose and usage of the various security elements given in Figure 8-1 and Figure 8-2 is discussed in the following subsections.

Security References

Both EJBs and servlets may declare one or more **security-role-ref** elements. This element is used to declare that a component is using the **role-name** value as an argument to the **isCallerInRole(String)** method. Using the **isCallerInRole** method, a component can verify if the caller is in a role that has been declared with a **security-role-ref/role-name** element. The **role-name** element value must link to a **security-role** element through the **role-link** element. The typical use of **isCallerInRole** is to perform a security check that cannot be defined using the role based **method-permissions** elements. However, use of **isCallerInRole** is discouraged because this results in security logic embedded inside of the

component code. Example descriptor fragments that illustrate security-role-ref usage are presented in Listing presented in Listing 8-1.

LISTING 8-1. An example ejb-jar.xml and web.xml descriptor fragments which illustrate the security-role-ref element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
        <role-name>TheRoleICheck</role-name>
        <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>

<!-- A sample web.xml fragment -->
<web-app>
  <servlet>
    <servlet-name>AServlet</servlet-name>
    ...
    <security-role-ref>
      <role-name>TheServletRole</role-name>
      <role-link>TheApplicationRole</role-link>
    </security-role-ref>
  </servlet>
  ...
</web-app>
```

Security Identity

EJBs can optionally declare a security-identity element. New to EJB 2.0 is the capability to specify what identity an EJB should use when it invokes methods on other components. The invocation identity can be that of the current caller, or a specific role. The application assembler uses the security-identity element with a use-caller-identity child element to indicate the current caller's identity should be propagated as the security identity for method invocations made by the EJB. Propagation of the caller's identity is the default used in the absence of an explicit security-identity element declaration.

Alternatively, the application assembler can use the run-as/role-name child element to specify that a specific security role given by the role-name value should be used as the security identity for method invocations made by the EJB. Note that this does not change the caller's identity as seen by EJBContext.getCallerPrincipal(). Rather, the caller's security roles are set to the single role specified by the run-as/role-name element value. One use case for the run-as element is to prevent external clients from accessing internal EJBs. This is accomplished by assigning the internal EJB method-permission elements that restrict access to a role never assigned to an external client. EJBs that need to use inter-

nal EJB are then configured with a run-as/role-name equal to the restricted role. An example descriptor fragment that illustrates security-identity element usage is presented in Listing 8-2.

LISTING 8-2. An example ejb-jar.xml descriptor fragment which illustrates the security-identity element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
<enterprise-beans>
    <session>
        <ejb-name>ASessionBean</ejb-name>
        ...
        <security-identity>
            <use-caller-identity/>
        </security-identity>
    </session>
    <session>
        <ejb-name>RunAsBean</ejb-name>
        ...
        <security-identity>
            <run-as>
                <description>A private internal role</description>
                <role-name>InternalRole</role-name>
            </run-as>
        </security-identity>
    </session>
</enterprise-beans>
...
</ejb-jar>
```

The same security identity capability has been introduced for servlets as of the J2EE 2.3 servlet specification but this capability is currently unsupported in JBoss 3.0.

Security roles

The security role name referenced by either the security-role-ref or security-identity element needs to map to one of the application's declared roles. An application assembler defines logical security roles by declaring security-role elements. The role-name value is a logical application role name like Administrator, Architect, SalesManager, etc.

What is a role? The J2EE specifications note that it is important to keep in mind that the security roles in the deployment descriptor are used to define the logical security view of an application. Roles defined in the J2EE deployment descriptors should not be confused with the user groups, users, principals, and other concepts that exist in the target enterprise's operational environment. The deployment descriptor roles are application constructs with application domain specific names. For example, a banking application might use role names like BankManager, Teller, and Customer.

In JBoss, a security-role is only used to map security-role-ref/role-name values to the logical role that the component role referenced. The user's assigned roles are a dynamic function of the application's security manager, as you will see when we discuss the JBossSX implementation details. JBoss does not require the definition of security-roles in order to declare method permissions. Therefore, the specification of security-role elements is simply a good practice to ensure portability across application servers and for deployment descriptor maintenance. Example descriptor fragments that illustrate security-role usage are presented in Listing 8-3.

LISTING 8-3. An example ejb-jar.xml and web.xml descriptor fragments which illustrate the security-role element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
...
<assembly-descriptor>
    <security-role>
        <description>The single application role</description>
        <role-name>TheApplicationRole</role-name>
    </security-role>
</assembly-descriptor>
</ejb-jar>

<!-- A sample web.xml fragment -->
<web-app>
...
    <security-role>
        <description>The single application role</description>
        <role-name>TheApplicationRole</role-name>
    </security-role>
</web-app>
```

EJB method permissions

An application assembler can set the roles that are allowed to invoke an EJB's home and remote interface methods through method-permission element declarations. Each method-permission element contains one or more role-name child elements that define the logical roles allowed access the EJB methods as identified by method child elements. As of EJB 2.0, you can now specify an unchecked element instead of the role-name element to declare that any authenticated user can access the methods identified by method child elements. In addition, you can declare that no one should have access to a method with the exclude-list element. If an EJB has methods that have not been declared as accessible by a role using a method-permission element, the EJB methods default to being excluded from use. This is equivalent to defaulting the methods into the exclude-list.

There are three supported styles of method element declarations.

- Style 1, is used for referring to all of the home and component interface methods of the named enterprise bean.

```
<method>
    <ejb-name>EJBNAME</ejb-name>
```

```
<method-name>*</method-name>
</method>
```

- Style 2, is used for referring to a specified method of the home or component interface of the named enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods.

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
</method>
```

- Style 3, is used to refer to a specified method within a set of methods with an overloaded name. The method must be defined in the specified enterprise bean's home or remote interface. The method-param element values are the fully qualified name of the corresponding method parameter type. If there are multiple methods with the same overloaded signature, the permission applies to all of the matching overloaded methods.

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
  <method-params>
    <method-param>PARAMETER_1</method-param>
    ...
    <method-param>PARAMETER_N</method-param>
  </method-params>
</method>
```

The optional method-intf element can be used to differentiate methods with the same name and signature that are defined in both the home and remote interfaces of an enterprise bean. Listing 8-4 provides examples of the method-permission element usage.

LISTING 8-4. An example ejb-jar.xml descriptor fragment which illustrates the method-permission element usage.

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles may
        access any method of the EmployeeService bean
      </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>

    <method-permission>
      <description>The employee role may access the
        findByPrimaryKey, getEmployeeInfo, and the
        updateEmployeeInfo(String) method of the AardvarkPayroll
        bean
      </description>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

```
<role-name>employee</role-name>
<method>
    <ejb-name>AardvarkPayroll</ejb-name>
    <method-name>findByPrimaryKey</method-name>
</method>

<method>
    <ejb-name>AardvarkPayroll</ejb-name>
    <method-name>getEmployeeInfo</method-name>
</method>

<method>
    <ejb-name>AardvarkPayroll</ejb-name>
    <method-name>updateEmployeeInfo</method-name>
    <method-params>
        <method-param>java.lang.String</method-param>
    </method-params>
</method>
</method-permission>

<method-permission>
    <description>The admin role may access any method
        of the EmployeeServiceAdmin bean
    </description>
    <role-name>admin</role-name>
    <method>
        <ejb-name>EmployeeServiceAdmin</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>

<method-permission>
    <description>Any authenticated user may access any method
        of the EmployeeServiceHelp bean
    </description>
    <unchecked/>
    <method>
        <ejb-name>EmployeeServiceHelp</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>

<exclude-list>
    <description>No fireTheCTO methods of the EmployeeFiring
        bean may be used in this deployment
    </description>
    <method>
        <ejb-name>EmployeeFiring</ejb-name>
        <method-name>fireTheCTO</method-name>
    </method>
</exclude-list>
</assembly-descriptor>
</ejb-jar>
```

Web content security constraints

In a Web application, security is defined by the roles allowed access to content by a URL pattern that identifies the protected content. This set of information is declared using the web.xml security-constraint element. The content to be secured is declared using one or more web-resource-collection elements. Each web-resource-collection element contains an optional series of url-pattern elements followed by an optional series of http-method elements. The url-pattern element value specifies a URL pattern against which a request URL must match for the request to correspond to an attempt to access secured content. The http-method element value specifies a type of HTTP request to allow.

The optional user-data-constraint element specifies the requirements for the transport layer of the client to server connection. The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The transport-guarantee element value specifies the degree to which communication between client and server should be protected. Its values are NONE, INTEGRAL, or CONFIDENTIAL. A value of NONE means that the application does not require any transport guarantees. A value of INTEGRAL means that the application requires the data sent between the client and server be sent in such a way that it can't be changed in transit. A value of CONFIDENTIAL means that the application requires the data be transmitted in a fashion that prevents other entities from observing the contents of the transmission. In most cases, the presence of the INTEGRAL or CONFIDENTIAL flag indicates that the use of SSL is required.

The optional login-config is used to configure the authentication method that should be used, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism. The auth-method child element specifies the authentication mechanism for the Web application. As a prerequisite to gaining access to any Web resources that are protected by an authorization constraint, a user must have authenticated using the configured mechanism. Legal values for auth-method are BASIC, DIGEST, FORM, or CLIENT-CERT. The realm-name child element specifies the realm name to use in HTTP BASIC and DIGEST authorization. The form-login-config child element specifies the log in as well as error pages that should be used in form-based login. If the auth-method value is not FORM, form-login-config and its child elements are ignored.

As an example, the web.xml descriptor fragment given in Listing 8-5 indicates that any URL lying under the web application “/restricted” path requires an AuthorizedUser role. There is no required transport guarantee and the authentication method used for obtaining the user identity is BASIC HTTP authentication.

LISTING 8-5. A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements.

```
<web-app>
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secure Content</web-resource-name>
    <url-pattern>/restricted/*</ url-pattern></
  <web-resource-collection>
    <auth-constraint>
```

```
<role-name>AuthorizedUser</role-name>
</auth-constraint>
<user-data-constraint>
  <transport-guarantee>NONE</transport-guarantee>
</user-data-constraint>
</security-constraint>
...
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>The Restricted Zone</realm-name>
</login-config>
...
<security-role>
  <description>The role required to access restricted content
  </description>
  <role-name>AuthorizedUser</role-name>
</security-role>
</web-app>
```

Enabling Declarative Security in JBoss

The J2EE security elements that have been covered describe only the security requirements from the application's perspective. Since J2EE security elements declare logical roles, the application deployer maps the roles from the application domain onto the deployment environment. The J2EE specifications omit these application-server-specific details. In JBoss, mapping the application roles onto the deployment environment entails specifying a security manager that implements the J2EE security model using JBoss server specific deployment descriptors. We will avoid discussion the details of this step for now. The details behind the security configuration will be discussed when we describe the generic JBoss server security interfaces in the section “The JBoss Security Model” on page 251 .

An Introduction to JAAS

The default implementation of the JBossSX framework is based on the JAAS API. Because this is a relatively new API, one which has not seen wide spread use, its important that you understand the basic elements of the JAAS API to understand the implementation details of JBossSX. This section provides an introduction to JAAS to prepare you for the JBossSX architecture discussion.

Additional details on the JAAS package can be found at the JAAS home page at: <http://java.sun.com/products/jaas/>.

What is JAAS?

The JAAS 1.0 API consists of a set of Java packages designed for user authentication and authorization. It implements a Java version of the standard Pluggable Authentication Module (PAM) frame-

work and compatibly extends the Java 2 Platform's access control architecture to support user-based authorization. JAAS was first released as an extension package for JDK 1.3 and is bundled with the current JDK 1.4 beta. Because the JBossSX framework uses only the authentication capabilities of JAAS to implement the declarative role-based J2EE security model, this introduction focuses on only that topic.

Much of this section's material is derived from the JAAS 1.0 Developers Guide, so if you are familiar with its content you can skip ahead to the JBossSX architecture discussion section "The JBoss Security Extension Architecture" on page 259

JAAS authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies and allows the JBossSX security manager to work in different security infrastructures. Integration with a security infrastructure can be achieved without changing the JBossSX security manager implementation. All that needs to change is the configuration of the authentication stack that JAAS uses.

The JAAS Core Classes

The JAAS core classes can be broken down into three categories: common, authentication, and authorization. The following list presents only the common and authentication classes because these are the specific classes used to implement the functionality of JBossSX covered in this chapter.

Common classes:

- Subject ([javax.security.auth.Subject](#))
- Principal ([java.security.Principal](#))

Authentication classes:

- Callback ([javax.security.auth.callback.Callback](#))
- CallbackHandler ([javax.security.auth.callback.CallbackHandler](#))
- Configuration ([javax.security.auth.login.Configuration](#))
- LoginContext ([javax.security.auth.login.LoginContext](#))
- LoginModule ([javax.security.auth.spi.LoginModule](#))

SUBJECT AND PRINCIPAL

To authorize access to resources, applications first need to authenticate the request's source. The JAAS framework defines the term subject to represent a request's source. The [Subject](#) class is the central class in JAAS. A [Subject](#) represents information for a single entity, such as a person or service. It encompasses the entity's principals, public credentials, and private credentials. The JAAS APIs use the existing Java 2 [java.security.Principal](#) interface to represent a principal, which is essentially just a typed name.

public final class Subject implements java.io.SerializableDuring the authentication process, a [Subject](#) is populated with associated identities, or [Principals](#). A [Subject](#) may have many [Principals](#). For example, a person may have a name [Principal](#) (John Doe) and a social security number [Principal](#) (123-45-

6789), and a username Principal (johnd), all of which help distinguish the Subject from other Subjects. To retrieve the Principals associated with a Subject, two methods are available:

```
{  
...  
    public Set getPrincipals() {...}  
    public Set getPrincipals(Class c) {...}  
}
```

The first method returns all Principals contained in the Subject. The second method only returns those Principals that are instances of Class c or one of its subclasses. An empty set will be returned if the Subject has no matching Principals. Note that the java.security.acl.Group interface is a subinterface of java.security.Principal, and so an instance in the principals set may represent a logical grouping of other principals or groups of principals.

AUTHENTICATION OF A SUBJECT

Authentication of a Subject requires a JAAS login. The login procedure consists of the following steps:

1. An application instantiates a LoginContext passing in the name of the login configuration and a CallbackHandler to populate the Callback objects as required by the configuration LoginModules.
2. The LoginContext consults a Configuration to load all of the LoginModules included in the named login configuration. If no such named configuration exists the “other” configuration is used as a default.
3. The application invokes the LoginContext.login method.
4. The login method invokes all the loaded LoginModules. As each LoginModule attempts to authenticate the Subject, it invokes the handle method on the associated CallbackHandler to obtain the information required for the authentication process. The required information is passed to the handle method in the form of an array of Callback objects. Upon success, the LoginModules associate relevant Principals and credentials with the Subject.
5. The LoginContext returns the authentication status to the application. Success is represented by a return from the login method. Failure is represented through a LoginException being thrown by the login method.
6. If authentication succeeds, the application retrieves the authenticated Subject using the LoginContext.getSubject method.
7. After the scope of the Subject authentication is complete, all Principals and related information associated with the Subject by the login method may be removed by invoking the LoginContext.logout method.

The LoginContext class provides the basic methods for authenticating Subjects and offers a way to develop an application independent of the underlying authentication technology. The LoginContext consults a Configuration to determine the authentication services configured for a particular application. LoginModules classes represent the authentication services. Therefore, you can plug in different LoginModules into an application without changing the application itself. Listing 8-6 provides code fragments that illustrate the steps required by an application to authenticate a Subject.

LISTING 8-6. An illustration of the steps of the authentication process from the application perspective.

```
CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);
try
{
    lc.login();
    Subject subject = lc.getSubject();
}
catch(LoginException e)
{
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
...

// Scope of work complete, logout to remove authentication info
try
{
    lc.logout();
}
catch(LoginException e)
{
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++)
        {
            if (callbacks[i] instanceof NameCallback)
            {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            }
            else if (callbacks[i] instanceof PasswordCallback)
            {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                pc.setPassword(password);
            }
            else
            {
                throw new UnsupportedCallbackException(callbacks[i],
                    "Unrecognized Callback");
            }
        }
    }
}
```

```
}
```

Developers integrate with an authentication technology by creating an implementation of the LoginModule interface. This allows different authentication technologies to be plugged into an application by administrator. Multiple LoginModules can be chained together to allow for more than one authentication technology as part of the authentication process. For example, one LoginModule may perform username/password-based authentication, while another may interface to hardware devices such as smart card readers or biometric authenticators. The life cycle of a LoginModule is driven by the LoginContext object against which the client creates and issues the login method. The process consists of a two phases. The steps of the process are as follows:

1. The LoginContext creates each configured LoginModule using its public no-arg constructor.
2. Each LoginModule is initialized with a call to its initialize method. The Subject argument is guaranteed to be non-null. The signature of the initialize method is:

```
public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options);
```

3. The login method is then called to start the authentication process. An example method implementation might prompt the user for a username and password, and then verify the information against data stored in a naming service such as NIS or LDAP. Alternative implementations might interface to smart cards and biometric devices, or simply extract user information from the underlying operating system. The validation of user identity by each LoginModule is considered phase 1 of JAAS authentication. The signature of the login method is:

```
boolean login() throws LoginException;
```

4. If the LoginContext's overall authentication succeeds, commit is invoked on each LoginModule. If phase 1 succeeded for a LoginModule, then the commit method continues with phase 2: associating relevant Principals, public credentials, and/or private credentials with the Subject. If phase 1 fails for a LoginModule, then commit removes any previously stored authentication state, such as usernames or passwords. The signature of the commit method is:

```
boolean commit() throws LoginException;
```

5. If the LoginContext's overall authentication failed, then the abort method is invoked on each LoginModule. The abort method removes/destroys any authentication state created by the login or initialize methods. The signature of the abort method is:

```
boolean abort() throws LoginException;
```

6. Removal of the authentication state after a successful login is accomplished when the application invokes logout on the LoginContext. This in turn results in a logout method invocation on each LoginModule. The logout method removes the Principals and credentials originally associated with the Subject during the commit operation. Credentials should be destroyed upon removal. The signature of the logout method is:

```
boolean logout() throws LoginException;
```

When a LoginModule must communicate with the user to obtain authentication information, it uses a CallbackHandler object. Applications implement the CallbackHandler interface and pass it to the LoginContext, which forwards it directly to the underlying LoginModules. LoginModules use the CallbackHandler both to gather input from users, such as a password or smart-card PIN number, and to supply information to users, such as status information. By allowing the application to specify the CallbackHandler, underlying LoginModules remain independent from the different ways applications

interact with users. For example, a CallbackHandler's implementation for a GUI application might display a window to solicit user input. On the other hand, a CallbackHandler's implementation for a non-GUI environment, such as an application server, might simply obtain credential information using an application server API. The CallbackHandler interface has one method to implement:

```
void handle(Callback[] callbacks)
throws java.io.IOException, UnsupportedCallbackException;
```

The last authentication class to cover is the Callback interface. This is a tagging interface for which several default implementations are provided, including NameCallback and PasswordCallback that were used in Listing 8-6. LoginModules use a Callback to request information required by the authentication mechanism the LoginModule encapsulates. LoginModules pass an array of Callbacks directly to the CallbackHandler.handle method during the authentication's login phase. If a CallbackHandler does not understand how to use a Callback object passed into the handle method, it throws an UnsupportedCallbackException to abort the login call.

The JBoss Security Model

Similar to the rest of the JBoss architecture, security at the lowest level is defined as a set of interfaces for which alternate implementations may be provided. There are three basic interfaces that define the JBoss server security layer – org.jboss.security.AuthenticationManager, org.jboss.security.Realm-Mapping, and org.jboss.security.SecurityProxy. Figure 8-3 shows a class diagram of the security interfaces and their relationship to the EJB container architecture.

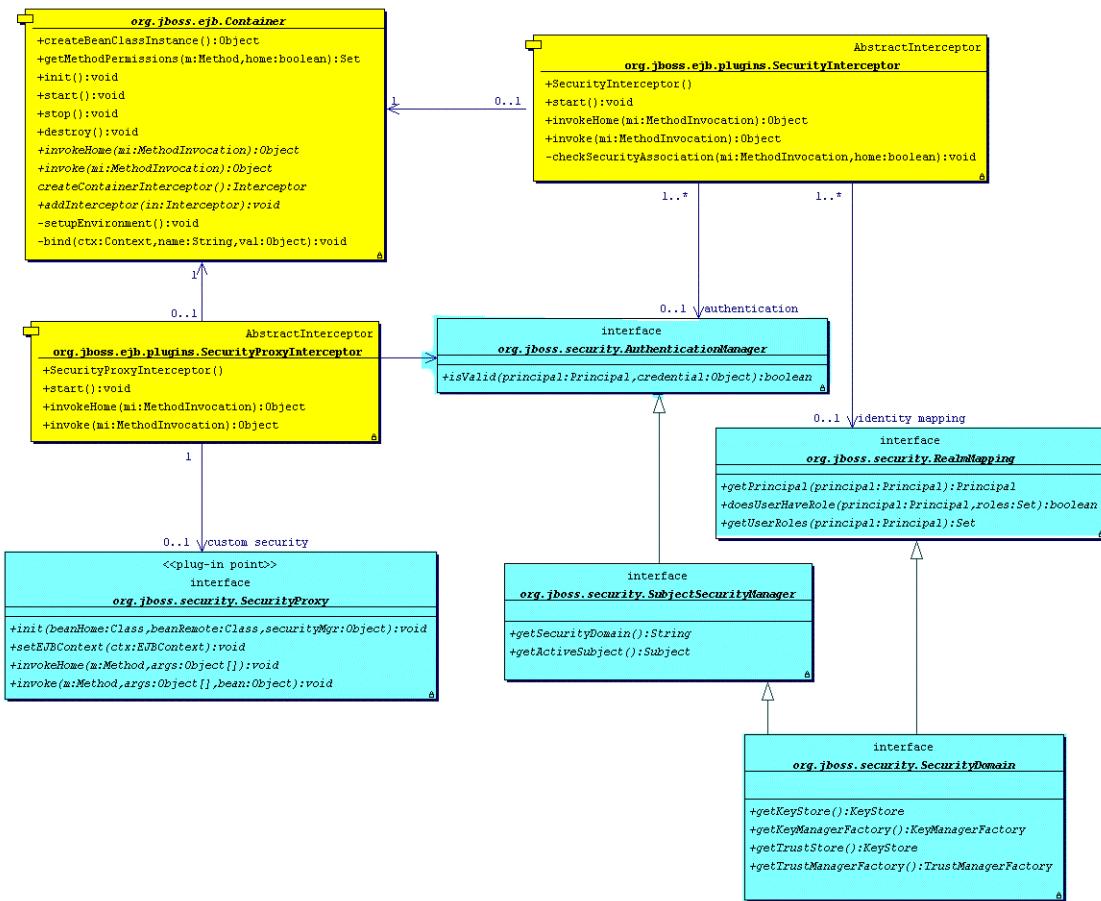


FIGURE 8-3. The key security model interfaces and their relationship to the JBoss server EJB container elements.

The light blue classes represent the security interfaces while the yellow classes represent the EJB container layer. The two interfaces required for the implementation of the J2EE security model are the `org.jboss.security.AuthenticationManager` and `org.jboss.security.RealmMapping`. The roles of the security interfaces presented in Figure 8-3 are summarized in the following list.

- **AuthenticationManager** is an interface responsible for validating credentials associated with principals. Principals are identities and examples include usernames, employee numbers, social security numbers, and so on. Credentials are proof of the identity and examples include passwords, session keys, digital signatures, and so on. The `isValid` method is invoked to see if a user identity and associated credentials as known in the operational environment are valid proof of the user identity..
- **RealmMapping** is an interface responsible for principal mapping and role mapping. The `getPrincipal` method takes a user identity as known in the operational environment and returns the application domain identity. The `doesUserHaveRole` method validates that the user identity in the operation environment has been assigned the indicated role from the application domain.

- SecurityProxy is an interface describing the requirements for a custom SecurityProxyInterceptor plugin. A SecurityProxy allows for the externalization of custom security checks on a per-method basis for both the EJB home and remote interface methods.
- SubjectSecurityManager is a subinterface of AuthenticationManager that simply adds accessor methods for obtaining the security domain name of the security manager and the current thread's authenticated Subject. In future releases this interface will simply be integrated into the SecurityDomain interface.
- SecurityDomain is an extension of the AuthenticationManager, RealmMapping, and SubjectSecurityManager interfaces. It is a move to a comprehensive security interface based on the JAAS Subject, a java.security.KeyStore, and the JSSE com.sun.net.ssl.KeyManagerFactory and com.sun.net.ssl.TrustManagerFactory interfaces. This interface is still a work in progress that will be the basis of a multi-domain security architecture that will better support ASP style deployments of applications and resources.

Note that the AuthenticationManager, RealmMapping and SecurityProxy interfaces have no association to JAAS related classes. Although the JBossSX framework is heavily dependent on JAAS, the basic security interfaces required for implementation of the J2EE security model are not. The JBossSX framework is simply an implementation of the basic security plug-in interfaces that are based on JAAS. The component diagram presented in Figure 8.4 illustrates this fact. The implication of this plug-in architecture is that you are free to replace the JAAS-based JBossSX implementation classes with your own custom security manager implementation that does not make use of JAAS, if you so desire. You'll see how to do this when you look at the JBossSX MBeans available for the configuration of JBossSX in Figure 8-4.

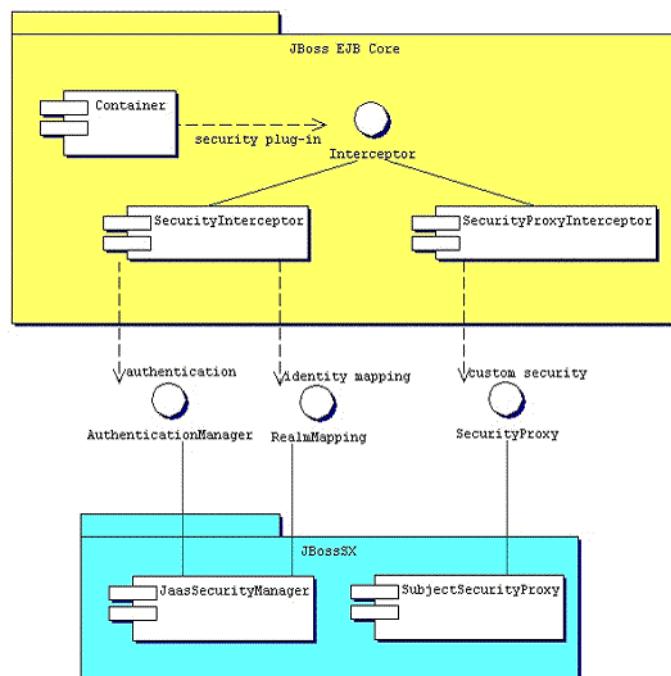


FIGURE 8-4. The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.

Enabling Declarative Security in JBoss Revisited

Recall that our discussion of the J2EE standard security model ended with a requirement for the use of JBoss server specific deployment descriptor to enable security. The details of this configuration is presented here, as this is part of the generic JBoss security model. Figure 8-5 shows the JBoss-specific EJB and Web application deployment descriptor's security-related elements.

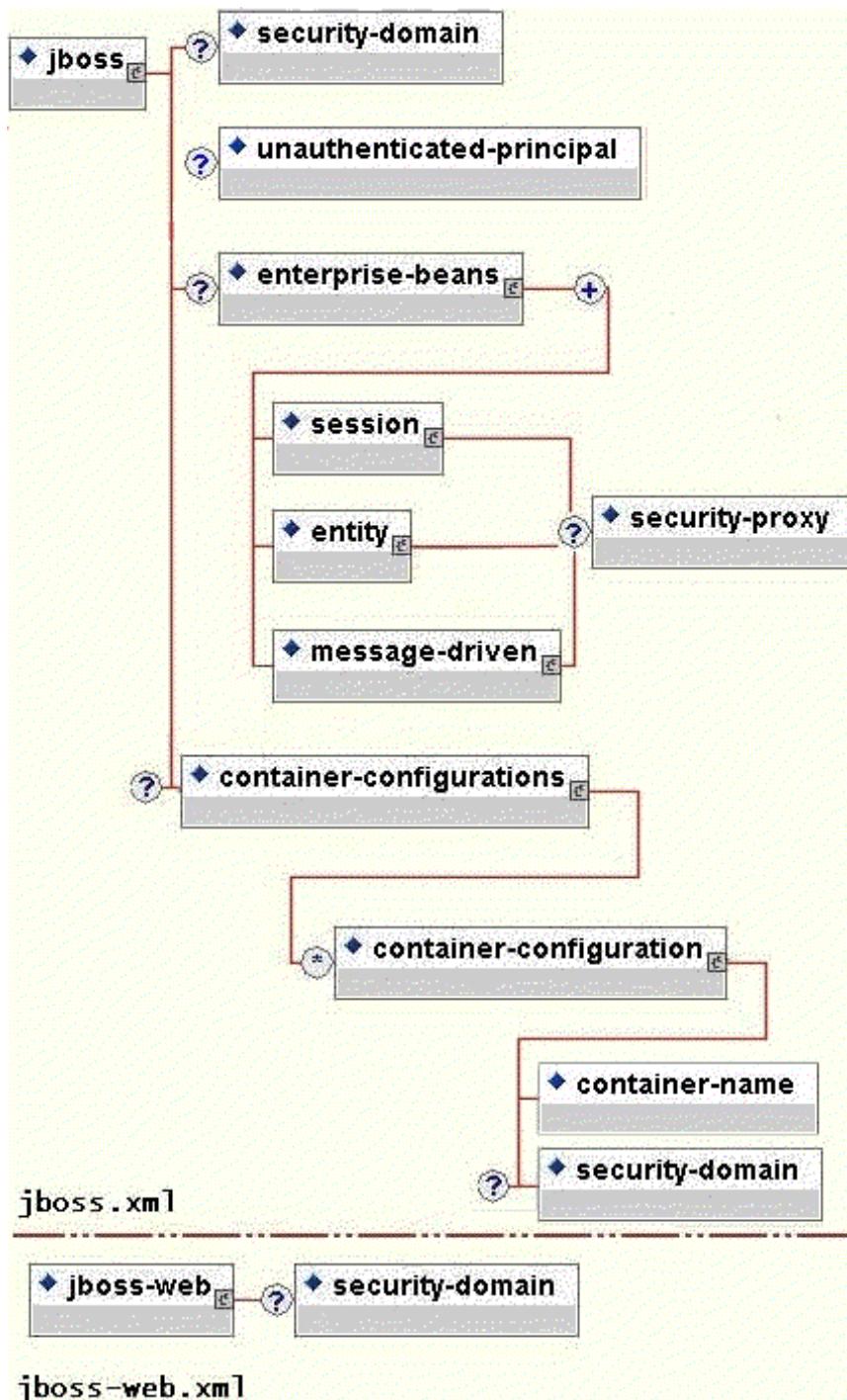


FIGURE 8-5. The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.

The value of a security-domain element specifies the JNDI name of the security manager interface implementation that JBoss uses for the EJB and Web containers. This is an object that implements both of the AuthenticationManager and RealmMapping interfaces. When specified as a top-level element it defines what security domain in effect for all EJBs in the deployment unit. This is the typical usage because mixing security managers within a deployment unit complicates inter-component operation and administration.

To specify the security-domain for an individual EJB, you specify the security-domain at the container configuration level. This will override any top-level security-domain element..

The unauthenticated-principal element specifies the name to use for the Principal object returned by the EJBContext.getUserPrincipal method when an unauthenticated user invokes an EJB. Note that this conveys no special permissions to an unauthenticated caller. Its primary purpose is to allow uncured servlets and JSP pages to invoke unsecured EJBs and allow the target EJB to obtain a non-null Principal for the caller using the getUserPrincipal method. This is a J2EE specification requirement.

The security-proxy element identifies a custom security proxy implementation that allows per-request security checks outside the scope of the EJB declarative security model without embedding security logic into the EJB implementation. This may be an implementation of the org.jboss.security.SecurityProxy interface, or just an object that implements methods in the home or remote interface of the EJB to secure without implementing any common interface. If the given class does not implement the SecurityProxy interface, the instance must be wrapped in a SecurityProxy implementation that delegates the method invocations to the object. The org.jboss.security.SubjectSecurityProxy is an example implementation used by the default JBossSX installation.

Take a look at a simple example of a custom SecurityProxy in the context of a trivial stateless session bean. The custom SecurityProxy validates that no one invokes the bean's echo method with a four-letter word as its argument. This is a check that is not possible with role-based security[md]you cannot define a FourLetterEchoInvoker role because the security context is the method argument, not a property of the caller. The code for the custom SecurityProxy is given in Listing 8-7, and the full source code is available in the src/main/org/jboss/chap8/ex1 directory of the book examples. The associated jboss.xml descriptor that installs the EchoSecurityProxy as the custom proxy for the EchoBean is given in Listing 8-8.

LISTING 8-7. The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.

```
package org.jboss.chap8.ex1;

import java.lang.reflect.Method;
import javax.ejb.EJBContext;

import org.apache.log4j.Category;
```

```
import org.jboss.security.SecurityProxy;

/** A simple example of a custom SecurityProxy implementation
that demonstrates method argument based security checks.

 * @author Scott.Stark@jboss.org
 * @version $Revision:$
 */
public class EchoSecurityProxy implements SecurityProxy
{
    Category log = Category.getInstance(EchoSecurityProxy.class);
    Method echo;

    public void init(Class beanHome, Class beanRemote,
        Object securityMgr)
        throws InstantiationException
    {
        log.debug("init, beanHome="+beanHome
            + ", beanRemote="+beanRemote
            + ", securityMgr="+securityMgr);
        // Get the echo method for equality testing in invoke
        try
        {
            Class[] params = {String.class};
            echo = beanRemote.getDeclaredMethod("echo", params);
        }
        catch(Exception e)
        {
            String msg = "Failed to finde an echo(String) method";
            log.error(msg, e);
            throw new InstantiationException(msg);
        }
    }
    public void setEJBContext(EJBContext ctx)
    {
        log.debug("setEJBContext, ctx="+ctx);
    }
    public void invokeHome(Method m, Object[] args)
        throws SecurityException
    {
        // We don't validate access to home methods
    }
    public void invoke(Method m, Object[] args, Object bean)
        throws SecurityException
    {
        log.debug("invoke, m="+m);
        // Check for the echo method
        if( m.equals(echo) )
        {
            // Validate that the msg arg is not 4 letter word
            String arg = (String) args[0];
            if( arg == null || arg.length() == 4 )
                throw new SecurityException("No 4 letter words");
        }
        // We are not responsible for doing the invoke
    }
}
```

```
    }
}
```

LISTING 8-8. The jboss.xml descriptor which configures the EchoSecurityProxy as the custom security proxy for the EchoBean.

```
<jboss>
  <security-domain>java:/jaas/other</security-domain>

  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <security-proxy>org.jboss.chap8.ex1.EchoSecurityProxy
      </security-proxy>
    </session>
  </enterprise-beans>
</jboss>
```

The EchoSecurityProxy checks that the method to be invoked on the bean instance corresponds to the echo(String) method loaded the init method. If there is a match, the method argument is obtained and its length compared against 4 or null. Either case results in a SecurityException being thrown. Certainly this is a contrived example, but only in its application. It is a common requirement that applications must perform security checks based on the value of method arguments. The point of the example is to demonstrate how custom security beyond the scope of the standard declarative security model can be introduced independent of the bean implementation. This allows the specification and coding of the security requirements to be delegated to security experts. Since the security proxy layer can be done independent of the bean implementation, security can be changed to match the deployment environment requirements.

Now test the custom proxy by running a client that attempts to invoke the EchoBean.echo method with the arguments "Hello" and "Four" as illustrated in this fragment:

```
public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Logger log = Logger.getLogger("ExClient");
        log.info("Looking up EchoBean");
        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();
        log.info("Created Echo");
        log.info("Echo.echo('Hello') = "+echo.echo("Hello"));
        log.info("Echo.echo('Four') = "+echo.echo("Four"));
    }
}
```

The first call should succeed, while the second should fail due to the fact that "Four" is a four-letter word. Run the client as follows using Ant from the examples directory:

```
examples 817>ant -Dchap=8 -Dex=1 run-example
```

```
Buildfile: build.xml

...
chap8-ex1-jar:

run-example1:
[copy] Copying 1 file to G:\JBossReleases\jboss-3.0.1RC1\server\default\deploy
[echo] Waiting for 5 seconds for deploy...
[java] [INFO,ExClient] Looking up EchoBean
[java] [INFO,ExClient] Created Echo
[java] [INFO,ExClient] Echo.echo('Hello') = Hello
[java] java.rmi.ServerException: RemoteException occurred in server thread;
nested exception is:
[java]     java.rmi.ServerException: No 4 letter words; nested exception is:
[java]     java.lang.SecurityException: No 4 letter words
[java]     java.rmi.ServerException: No 4 letter words; nested exception is:
[java]     java.lang.SecurityException: No 4 letter words
[java]     java.lang.SecurityException: No 4 letter words
[java]     at
...sport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:240
)
[java]     at
sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:215)
[java]     at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:117)
[java]     at org.jboss.invocation.jrmp.server.JRMPInvoker_Stub.invoke(Unknown
Source)
[java]
at....invocation.jrmp.interfaces.JRMPInvokerProxy.invoke(JRMPInvokerProxy.java:
128)
[java]     at
org.jboss.invocation.InvokerInterceptor.invoke(InvokerInterceptor.java:108)
[java]     at
org.jboss.proxy.TransactionInterceptor.invoke(TransactionInterceptor.java:73)
[java]     at
org.jboss.proxy.SecurityInterceptor.invoke(SecurityInterceptor.java:76)
[java]     at
....ejb.StatelessSessionInterceptor.invoke(StatelessSessionInterceptor.java:111
)
[java]     at org.jboss.proxy.ClientContainer.invoke(ClientContainer.java:76)
[java]     at $Proxyl.echo(Unknown Source)
[java]     at org.jboss.chap8.ex1.ExClient.main(ExClient.java:23)
[java] Exception in thread "main"
[java] Java Result: 1
```

The result is that the `echo('Hello')` method call succeeds as expected and the `echo('Four')` method call results in a rather messy looking exception, which is also expected. The above output has been truncated to fit in the book. The key part to the exception is that the `SecurityException("No 4 letter words")` generated by the `EchoSecurityProxy` was thrown to abort the attempted method invocation as desired.

The JBoss Security Extension Architecture

The preceding discussion of the general JBoss security layer has stated that the JBossSX security extension framework is an implementation of the security layer interfaces. This is the primary purpose of the JBossSX framework. The details of the implementation are interesting in that it offers a great deal of customization for integration into existing security infrastructures. A security infrastructure can be anything from a database or LDAP server to a sophisticated security software suite. The integration flexibility is achieved using the pluggable authentication model available in the JAAS framework.

The heart of the JBossSX framework is [org.jboss.security.plugins.JaasSecurityManager](#). This is the default implementation of the [AuthenticationManager](#) and [RealmMapping](#) interfaces. Figure 8-6 shows how the JaasSecurityManager integrates into the EJB and Web container layers based on the [security-domain](#) element of the corresponding component deployment descriptor.

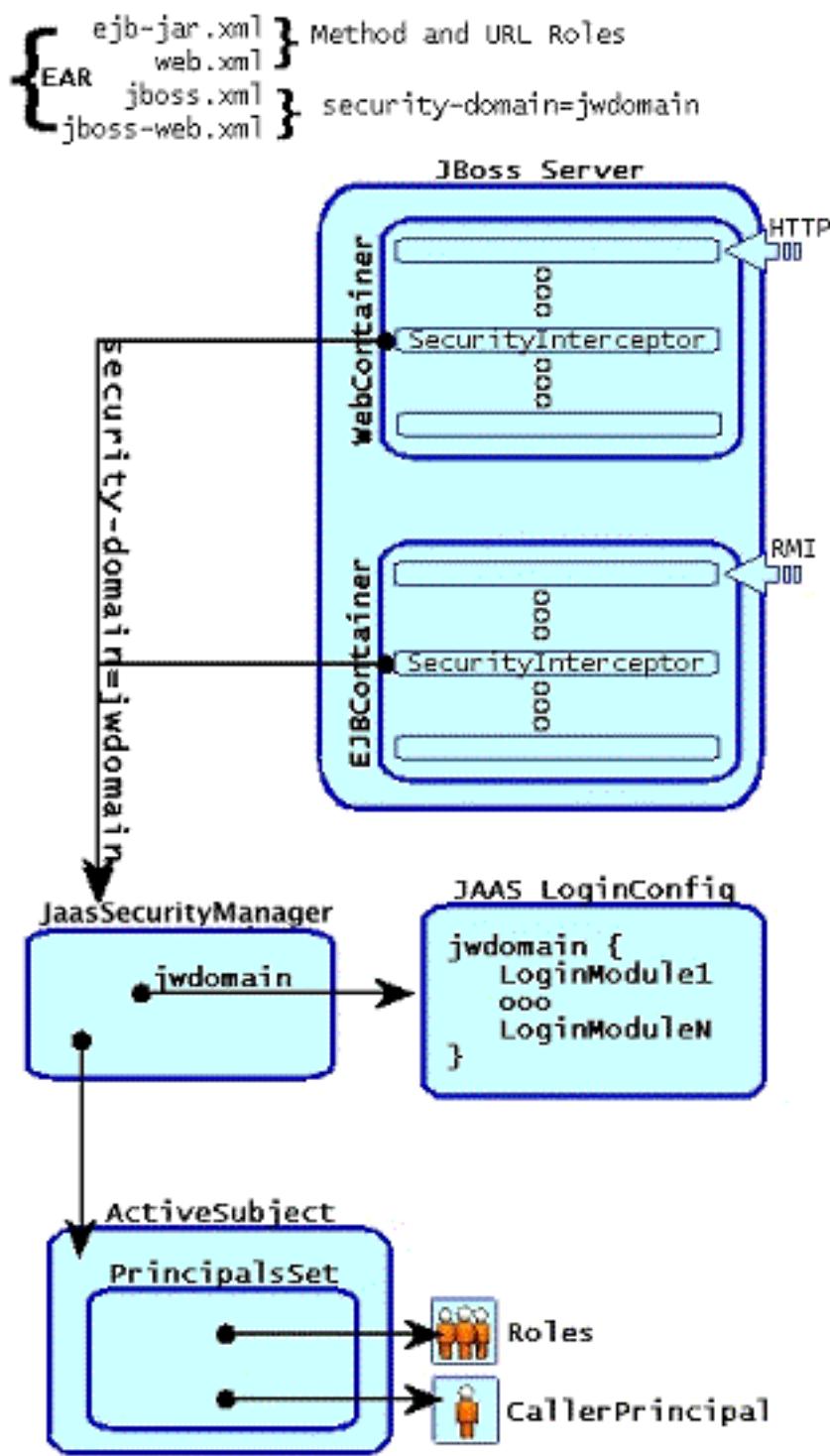


FIGURE 8-6. The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.

Figure 8-6 depicts an enterprise application that contains both EJBs and Web content secured under the security domain jwdomain. The EJB and Web containers have a request interceptor architecture

that includes a security interceptor, which enforces the container security model. At deployment time, the security-domain element value in the jboss.xml and jboss-web.xml descriptors is used to obtain the security manager instance associated with the container. The security interceptor then uses the security manager to perform its role. When a secured component is requested, the security interceptor delegates security checks to the security manager instance associated with the container.

The JBossSX JaasSecurityManager implementation, shown in Figure 8-6 as the JaasSecurityMgr component, performs security checks based on the information associated with the Subject instance that results from executing the JAAS login modules configured under the name matching the security-domain element value. We will drill into the JaasSecurityManager implementation and its use of JAAS in the following section.

How the JaasSecurityManager Uses JAAS

The JaasSecurityManager uses the JAAS packages to implement the AuthenticationManager and RealmMapping interface behavior. In particular, its behavior derives from the execution of the login module instances that are configured under the name that matches the security domain to which the JaasSecurityManager has been assigned. The login modules implement the security domain's principal authentication and role-mapping behavior. Thus, you can use the JaasSecurityManager across different security domains simply by plugging in different login module configurations for the domains.

To illustrate the details of the JaasSecurityManager's usage of the JAAS authentication process, you will walk through a client invocation of an EJB home method invocation. The prerequisite setting is that the EJB has been deployed in the JBoss server and its home interface methods have been secured using method-permission elements in the ejb-jar.xml descriptor, and it has been assigned a security domain named "jwdomain" using the jboss.xml descriptor security-domain element.

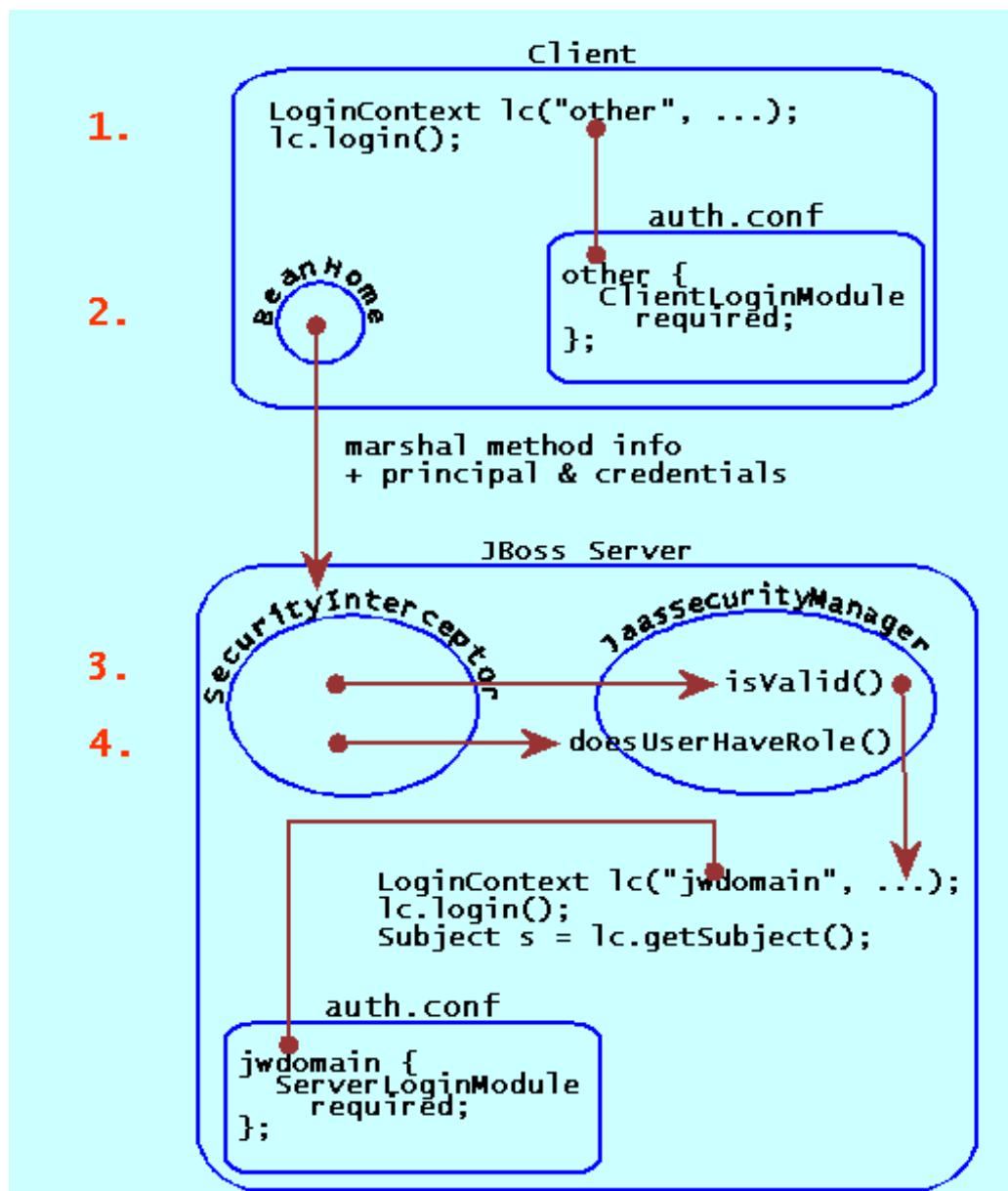


FIGURE 8-7. An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.

Figure 8-7 provides a view of the client to server communication we will discuss. The numbered steps shown in Figure 8-7 are:

1. The client first has to perform a JAAS login to establish the principal and credentials for authentication, and this is labeled *Client Side Login* in Figure 8-7. This is how clients establish their login identities in JBoss. Support for presenting the login information via JNDI InitialContext properties is not provided. A JAAS login entails creating a LoginContext instance and passing the name of the configuration to use. In Figure 8.7, the configuration name is “other”. This

one-time login associates the login principal and credentials with all subsequent EJB method invocations. Note that the process might not authenticate the user. The nature of the client-side login depends on the login module configuration that the client uses. In Figure 8.7, the “other” client-side login configuration entry is set up to use the ClientLoginModule module (an `org.jboss.security.ClientLoginModule`). This is the default client side module that simply binds the username and password to the JBoss EJB invocation layer for later authentication on the server. The identity of the client is not authenticated on the client.

2. Later, the client obtains the EJB home interface and attempts to create a bean. This event is labeled as *Home Method Invocation* in Figure 8-7 . This results in a home interface method invocation being sent to the JBoss server. The invocation includes the method arguments passed by the client along with the user identity and credentials from the client-side JAAS login performed in step 1.
3. On the server side, the security interceptor first requires authentication of the user invoking the call, which, as on the client side, involves a JAAS login.
4. The security domain under which the EJB is secured determines the choice of login modules. The security domain name is used as the login configuration entry name passed to the LoginContext constructor. In Figure 8-7, the EJB security domain is “jwdomain”. If the JAAS login authenticates the user, a JAAS Subject is created that contains the following in its PrincipalsSet:

This usage pattern of the Subject Principals set is the standard usage that JBossSX expects of server side login modules. To ensure proper conformance to this pattern any custom login module you write should subclass the JBossSX AbstractServerLoginModule class or one of its subclasses, or at least follow the pattern as documented in the custom login module section “Writing Custom Login Modules” on page 277

- A `java.security.Principal` that corresponds to the client identity as known in the deployment security environment.
- A `java.security.acl.Group` named "Roles" that contains the role names from the application domain to which the user has been assigned. `org.jboss.security.SimplePrincipal` objects are used to represent the role names; `SimplePrincipal` is a simple string-based implementation of `Principal`. These roles are used to validate the roles assigned to methods in ejb-jar.xml and the `EJBContext.isCallerInRole(String)` method implementation.
- An optional `java.security.acl.Group` named "CallerPrincipal", which contains a single `org.jboss.security.SimplePrincipal` that corresponds to the identity of the application domain's caller. The "CallerPrincipal" sole group member will be the value returned by the `EJBContext.getCallerPrincipal()` method. The purpose of this mapping is to allow a `Principal` as known in the operational security environment to map to a `Principal` with a name known to the application. In the absence of a "CallerPrincipal" mapping the deployment security environment principal is used as the `getCallerPrincipal` method value. That is, the operational principal is the same as the application domain principal.
- The final step of the security interceptor check is to verify that the authenticated user has permission to invoke the requested method This is labeled as *Server Side Authorization* in 8-7. Performing the authorization this entails the following steps:
- Obtain the names of the roles allowed to access the EJB method from the EJB container. The role names are determined by ejb-jar.xml descriptor `role-name` elements of all `method-permission` elements containing the invoked method.

- If no roles have been assigned, or the method is specified in an exclude-list element, then access to the method is denied. Otherwise, the JaasSecurityManager.doesUserHaveRole(Principal, Set) method is invoked by the security interceptor to see if the caller has one of the assigned role names. The doesUserHaveRole method implementation iterates through the role names and checks if the authenticated user's Subject "Roles" group contains a SimplePrincipal with the assigned role name. Access is allowed if any role name is a member of the "Roles" group. Access is denied if none of the role names are members.
- If the EJB was configured with a custom security proxy, the method invocation is delegated to it. If the security proxy wants to deny access to the caller, it will throw a java.lang.SecurityException. If no SecurityException is thrown, access to the EJB method is allowed and the method invocation passes to the next container interceptor. Note that the SecurityProxyInterceptor handles this check and this interceptor is not shown in Figure 8-7.

Every secured EJB method invocation, or secured Web content access, requires the authentication and authorization of the caller because security information is handled as a stateless attribute of the request that must be presented and validated on each request. This can be an expensive operation if the JAAS login involves client-to-server communication. Because of this, the JaasSecurityManager supports the notion of an authentication cache that is used to store principal and credential information from previous successful logins. You can specify the authentication cache instance to use as part of the JaasSecurityManager configuration as you will see when the associated MBean service is discussed in following section. In the absence of any user-defined cache, a default cache that maintains credential information for a configurable period of time is used.

The JaasSecurityManagerService MBean

The JaasSecurityManagerService MBean service manages security managers. Although its name begins with Jaas, the security managers it handles need not use JAAS in their implementation. The name arose from the fact that the default security manager implementation is the JaasSecurityManager. The primary role of the JaasSecurityManagerService is to externalize the security manager implementation. You can change the security manager implementation by providing an alternate implementation of the AuthenticationManager and RealmMapping interfaces. Of course this is optional because, by default, the JaasSecurityManager implementation is used.

The second fundamental role of the JaasSecurityManagerService is to provide a JNDI javax.naming.spi.ObjectFactory implementation to allow for simple code-free management of the JNDI name to security manager implementation mapping. It has been mentioned that security is enabled by specifying the JNDI name of the security manager implementation via the security-domain deployment descriptor element. When you specify a JNDI name, there has to be an object-binding there to use. To simplify the setup of the JNDI name to security manager bindings, the JaasSecurityManagerService manages the association of security manager instances to names by binding a next naming system reference with itself as the JNDI ObjectFactory under the name "java:/jaas". This allows one to use a naming convention of the form "java:/jaas/XYZ" as the value for the security-domain element, and the security manager instance for the "XYZ" security domain will be created as needed for you. The security manager for the domain "XYZ" is created on the first lookup against the "java:/jaas/XYZ" binding by creating an instance of the class specified by the SecurityManagerClassName attribute

using a constructor that takes the name of the security domain. For example, consider the following container security configuration snippet:

```
<jboss>
  <!-- Configure all containers to be secured under the
      "hades" security domain -->
  <security-domain>java:/jaas/hades</security-domain>
  ...
</jboss>
```

Any lookup of the name "java:/jaas/hades" will return a security manager instance that has been associated with the security domain named "hades". This security manager will implement the AuthenticationManager and RealmMapping security interfaces and will be of the type specified by the JaasSecurityManagerService SecurityManagerClassName attribute.

The JaasSecurityManagerService MBean is configured by default for use in the standard JBoss distribution, and you can often use the default configuration as is. The configurable attributes of the JaasSecurityManagerService include:

- **SecurityManagerClassName:** The name of the class that provides the security manager implementation. The implementation must support both the org.jboss.security.AuthenticationManager and org.jboss.security.RealmMapping interfaces. If not specified this defaults to JAAS-based org.jboss.security.plugins.JaasSecurityManager.
- **SecurityProxyFactoryClassName:** The name of the class that provides the org.jboss.security.SecurityProxyFactory implementation. If not specified this defaults to org.jboss.security.SubjectSecurityProxyFactory.
- **AuthenticationCacheJndiName:** Specifies the location of the security credential cache policy. This is first treated as an ObjectFactory location capable of returning CachePolicy instances on a per-security-domain basis. This is done by appending the name of the security domain to this name when looking up the CachePolicy for a domain. If this fails, the location is treated as a single CachePolicy for all security domains. As a default, a timed cache policy is used.
- **DefaultCacheTimeout:** Specifies the default timed cache policy timeout in seconds, and defaults to 1800 seconds (30 minutes). The value you use for the timeout is a tradeoff between frequent authentication operations and how long credential information may be out of sync with respect to the security information store. This has no affect if the AuthenticationCacheJndiName has been changed from the default value.
- **DefaultCacheResolution:** Specifies the default timed cache policy resolution in seconds. This controls the interval between checks for timeouts. If not specified this defaults to 60 seconds(1 minute). This has no affect if the AuthenticationCacheJndiName has been changed from the default value.

The JaasSecurityManagerService also supports a mechanism that allows any security domain authentication cache to be flushed at runtime. This can be done to drop all cached credentials when the underlying store has been updated and you want the store state to be used immediately. The MBean operation signature is as follows:

```
public void flushAuthenticationCache(String securityDomain);
```

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "Security:name=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
server.invoke(jaasMgr, "flushAuthenticationCache", params, signature);
```

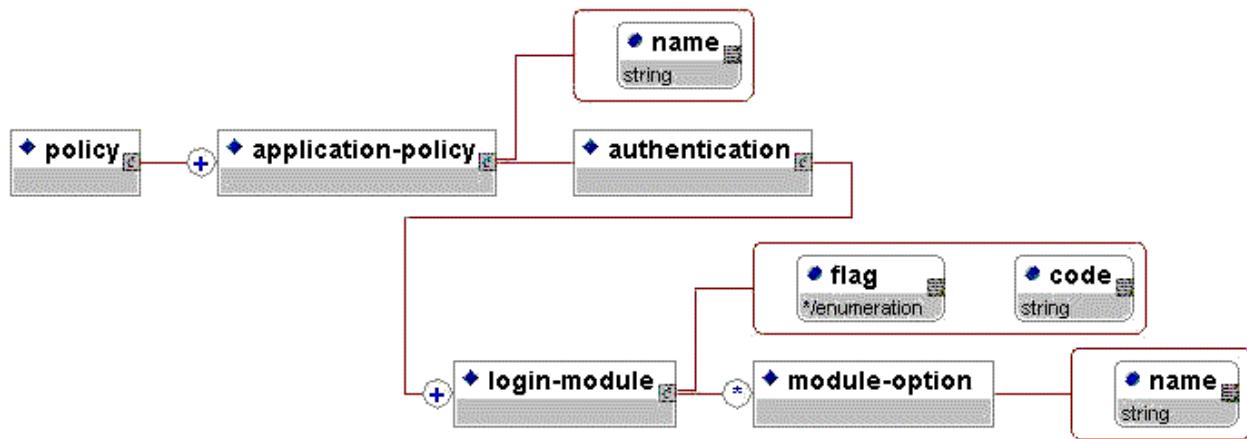
An Extension to JaasSecurityManager, the JaasSecurityDomain MBean

The org.jboss.security.plugins.JaasSecurityDomain is an extension of JaasSecurityManager that adds the notion of a KeyStore, and JSSE KeyManagerFactory and TrustManagerFactory for supporting SSL and other cryptographic use cases. The additional configurable attributes of the JaasSecurityDomain include:

- **KeyStoreType**: The type of the KeyStore implementation. This is the type argument passed to the java.security.KeyStore.getInstance(String type) factory method.
- **KeyStoreURL**: A URL to the location of the KeyStore database. This is used to obtain a java.io.InputStream using the URL.openStream() method to load the contents of a KeyStore instance.
- **KeyStorePass**: The password associated with the KeyStore database contents. This is used when the KeyStore instance is loaded from the KeyStoreURL contents. The Keyt-Store.load(InputStream, char[]) method. If this is not specified null will be used and the integrity of the database will not be checked.
- **LoadSunJSSEProvider**: A flag indicating if the Sun com.sun.net.ssl.internal.ssl.Provider security provider should be loaded on startup. This is needed when using the Sun JSSE jars without them installed as an extension with JDK 1.3. This should be set to false with JDK 1.4 or when using an alternate JSSE provider. This flag currently defaults to true.
- **ManagerServiceName**: Sets the JMX object name string of the security manager service MBean. This is used to register the defaults to register the JaasSecurityDomain as a the security manager under java:/jaas/<domain> where <domain> is the name passed to the MBean constructor. The name defaults to “jboss.security:service=JaasSecurityManager”.

An XML JAAS Login Configuration MBean

JBoss 3 uses a custom implementation of the javax.security.auth.login.Configuration class that is provided by the org.jboss.security.auth.login.XMLLoginConfig MBean. This configuration implementation uses an XML format that conforms to the DTD given by Figure 8-8.

**FIGURE 8-8. The XMLLoginConfig DTD**

The name attribute of the application-policy is the login configuration name. This corresponds to the portion of the jboss.xml and jboss-web.xml security-domain element value after the “java:/jaas/” prefix. The code attribute of the login-module element specifies the class name of the login module implementation. The flag attribute controls the overall behavior of the authentication stack. The allowed values and meanings are:

- **required**: the LoginModule is required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.
- **requisite**: the LoginModule is required to succeed. If it succeeds, authentication continues down the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed down the LoginModule list).
- **sufficient**: the LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed down the LoginModule list). If it fails, authentication continues down the LoginModule list.
- **optional**: the LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.

Zero or more module-option elements may be specified as child elements of a login-module. These define name/value string pairs that are made available to the login module during initialization. The name attribute specifies the option name while the module-option body provides the value. An example login configuration is given in Listing 8-9.

LISTING 8-9. A sample login module configuration suitable for use with XMLLoginConfig

```

<policy>
  <application-policy name="srp-test">
    <authentication>
      <login-module code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
                    flag="required">
        ...
      </login-module>
    </authentication>
  </application-policy>
</policy>
  
```

```
<module-option name="cacheJndiName">srp-test/AuthenticationCache
</module-option>
</login-module>

<login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
  flag="required">
  <module-option name="password-stacking">useFirstPass</module-option>
</login-module>

</authentication>
</application-policy>
</policy>
```

The XMLLoginConfig MBean supports the following attributes:

- **ConfigURL**: specifies the URL of the XML login configuration file that should be loaded by this mbean on startup. This must be a valid URL string representation.
- **ConfigResource**: specifies the resource name of the XML login configuration file that should be loaded by this mbean on startup. The name is treated as a classpath resource for which a URL is located using the thread context class loader.
- **ValidateDTD**: a flag indicating if the XML configuration should be validated against its DTD. This defaults to true.

The JAAS Login Configuration Management MBean

The installation of the custom javax.security.auth.login.Configuration is managed by the org.jboss.security.plugins.SecurityConfig MBean. The configurable attribute of the

- **LoginConfig**: Specifies the JMX ObjectName string of the that provides the default JAAS login configuration. When the SecurityConfig is started, this mean is queried for its javax.security.auth.login.Configuration by calling its getConfiguration(Configuration currentConfig) operation. If the **LoginConfig** attribute is not specified then the default Sun Configuration implementation described in the Configuration class javadocs is used.

In addition to allowing for a custom JAAS login configuration implementation, this service allows configurations to be chained together in a stack at runtime. This allows one to push a login configuration onto the stack and latter pop it. This is a feature used by the security unit tests to install custom login configurations into a default JBoss installation. Pusing a new configuration is done using:

```
public void pushLoginConfig(String objectName) throws JMException,
MalformedObjectNameException;
```

The objectName parameters specifies an MBean similar to the LoginConfig attribute. The current login configuration may be removed using:

```
public void popLoginConfig() throws JMException;
```

Using and Writing JBossSX Login Modules

The JaasSecurityManager implementation allows complete customization of the authentication mechanism using JAAS login module configurations. By defining the login module configuration entry that corresponds to the security domain name you have used to secure access to your J2EE components, you define the authentication mechanism and integration implementation.

The JBossSX framework includes a number of bundled login modules suitable for integration with standard security infrastructure store protocols such as LDAP and JDBC. It also includes standard base class implementations that help enforce the expected LoginModule to Subject usage pattern that was described in the “Writing Custom Login Modules” on page 277 These implementations allow for easy integration of your own authentication protocol, if none of the bundled login modules prove suitable. In this section we will first describe the useful bundled login modules and their configuration, and then end with a discussion of how to create your own custom LoginModule implementations for use with JBoss.

org.jboss.security.auth.spi.IdentityLoginModule

The IdentityLoginModule is a simple login module that associates the principal specified in the module options with any subject authenticated against the module. It creates a SimplePrincipal instance using the name specified by the "principal" option. Although this is certainly not an appropriate login module for production strength authentication, it can be of use in development environments when you want to test the security associated with a given principal and associated roles.

The supported login module configuration options include:

- **principal=string**, The name to use for the SimplePrincipal all users are authenticated as. The principal name defaults to "guest" if no principal option is specified.
- **roles=string-list**, The names of the roles that will be assigned to the user principal. The value is a comma-delimited list of role names.
- **password-stacking=useFirstPass**, When password-stacking option is set, this module first looks for a shared username under the property name "javax.security.auth.login.name" in the login module shared state Map. If found this is used as the principal name. If not found the principal name set by this login module is stored under the property name "javax.security.auth.login.name".

A sample login configuration entry that would authenticate all users as the principal named "jduke" and assign role names of "TheDuke", and "AnimatedCharacter" is:

```
testIdentity {
    org.jboss.security.auth.spi.IdentityLoginModule required
        principal=jduke
        roles=TheDuke,AnimatedCharater;
};
```

To add this entry to a JBoss server login configuration found in the default configuration file set you would modify the conf/default/auth.conf file of the JBoss distribution.

org.jboss.security.auth.spi.UsersRolesLoginModule

The UsersRolesLoginModule is another simple login module that supports multiple users and user roles, and is based on two Java Properties formatted text files. The username-to-password mapping file is called "users.properties" and the username-to-roles mapping file is called "roles.properties". The properties files are loaded during initialization using the initialize method thread context class loader. This means that these files can be placed into the J2EE deployment jar, the JBoss configuration directory, or any directory on the JBoss server or system classpath. The primary purpose of this login module is to easily test the security settings of multiple users and roles using properties files deployed with the application.

The users.properties file uses a "username=password" format with each user entry on a separate line as show here:

```
username1=password1  
username2=password2  
...  
...
```

The roles.properties file uses as "username=role1,role2,..." format with an optional group name value. For example:

```
username1=role1,role2,...  
username1.RoleGroup1=role3,role4,...  
username2=role1,role3,...
```

The "username.XXX" form of property name is used to assign the username roles to a particular named group of roles where the XXX portion of the property name is the group name. The "username=..." form is an abbreviation for "username.Roles=...", where the "Roles" group name is the standard name the JaasSecurityManager expects to contain the roles which define the users permissions.

The following would be equivalent definitions for the jduke username:

```
jduke=TheDuke,AnimatedCharacter  
jduke.Roles=TheDuke,AnimatedCharacter
```

The supported login module configuration options include the following:

- **unauthenticatedIdentity=name**, Defines the principal name that should be assigned to requests that contain no authentication information. This can be used to allow unprotected servlets to invoke methods on EJBs that do not require a specific role. Such a principal has no associated roles and so can only access either unsecured EJBs or EJB methods that are associated with the unchecked permission constraint.
- **password-stacking=useFirstPass**, When password-stacking option is set, this module first looks for a shared username and password under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively in the login module shared state Map. If found these are used as the principal name and password. If not found the principal name and password are set by this login module and stored under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively.
- **hashAlgorithm=string**: The name of the java.security.MessageDigest algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. When

hashAlgorithm is specified, the clear text password obtained from the [CallbackHandler](#) is hashed before it is passed to [UsernamePasswordLoginModule.validatePassword](#) as the inputPassword argument. The expectedPassword as stored in the users.properties file must be comparably hashed.

- **hashEncoding=base64|hex:** The string format for the hashed pass and must be either "base64" or "hex". Base64 is the default.
- **hashCharset=string:** The encoding used to convert the clear text password to a byte array. The platform default encoding is the default.
- **usersProperties=string:** (2.4.5+) The name of the properties resource containing the user-name to password mappings. This defaults to users.properties.
- **rolesProperties=string:** (2.4.5+) The name of the properties resource containing the user-name to roles mappings. This defaults to roles.properties.

A sample login configuration entry that assigned unauthenticated users the principal name "nobody" and contains based64 encoded, MD5 hashes of the passwords in a "usersb64.properties" file is:

```
testUsersRoles {
    org.jboss.security.auth.spi.UsersRolesLoginModule required
        usersProperties=usersb64.properties
        hashAlgorithm=MD5
        hashEncoding=base64
        unauthenticatedIdentity=nobody
    ;
}
```

[org.jboss.security.auth.spi.LdapLoginModule](#)

The [LdapLoginModule](#) is a LoginModule implementation that authenticates against an LDAP server using JNDI login using the login module configuration options. You would use the [LdapLoginModule](#) if your username and credential information are store in an LDAP server that is accessible using a JNDI LDAP provider.

The LDAP connectivity information is provided as configuration options that are passed through to the environment object used to create JNDI initial context. The standard LDAP JNDI properties used include the following:

- **java.naming.factory.initial**, The classname of the InitialContextFactory implementation. This defaults to the Sun LDAP provider implementation [com.sun.jndi.ldap.LdapCtxFactory](#).
- **java.naming.provider.url**, The ldap URL for the LDAP server
- **java.naming.security.authentication**, The security level to use. This defaults to "simple".
- **java.naming.security.protocol**, The transport protocol to use for secure access, such as, ssl
- **java.naming.security.principal**, The principal for authenticating the caller to the service. This is built from other properties as described below.
- **java.naming.security.credentials**, The value of the property depends on the authentication scheme. For example, it could be a hashed password, clear-text password, key, certificate, and so on.

The supported login module configuration options include the following:

- **principalDNPrefix=string**, A prefix to add to the username to form the user distinguished name. See principalDNSuffix for more info.
- **principalDNSuffix=string**, A suffix to add to the username when forming the user distinguished name. This is useful if you prompt a user for a username and you don't want the user to have to enter the fully distinguished name. Using this property and principalDNSuffix the userDN will be formed as:

```
String userDN = principalDNPrefix + username + principalDNSuffix;
```

- **useObjectCredential=true|false**, Indicates that the credential should be obtained as an opaque Object using the org.jboss.security.auth.callback.ObjectCallback type of Callback rather than as a char[] password using a JAAS PasswordCallback. This allows for passing non-char[] credential information to the LDAP server.

- **rolesCtxDN=string**, The fixed distinguished name to the context to search for user roles.
- **userRolesCtxDNAtributeName=string**, The name of an attribute in the user object that contains the distinguished name to the context to search for user roles. This differs from rolesCtxDN in that the context to search for a user's roles can be unique for each user.
- **roleAttributeID=string**, The name of the attribute that contains the user roles. If not specified this defaults to "roles".
- **uidAttributeID=string**, The name of the attribute in the object containing the user roles that corresponds to the userid. This is used to locate the user roles. If not specified this defaults to "uid".
- **matchOnUserDN=true|false**, A flag indicating if the search for user roles should match on the user's fully distinguished name. If false, just the username is used as the match value against the uidAttributeName attribute. If true, the full userDN is used as the match value.
- **unauthenticatedIdentity=string**, The principal name that should be assigned to requests that contain no authentication information. This behavior is inherited from the UsernamePasswordLoginModule superclass.
- **password-stacking=useFirstPass**, When the password-stacking option is set, this module first looks for a shared username and password under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively in the login module shared state Map. If found these are used as the principal name and password. If not found the principal name and password are set by this login module and stored under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively.
- **hashAlgorithm=string**: The name of the java.security.MessageDigest algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. When hashAlgorithm is specified, the clear text password obtained from the CallbackHandler is hashed before it is passed to UsernamePasswordLoginModule.validatePassword as the inputPassword argument. The expectedPassword as stored in the LDAP server must be comparably hashed.
- **hashEncoding=base64|hex**: The string format for the hashed pass and must be either "base64" or "hex". Base64 is the default.
- **hashCharset=string**: The encoding used to convert the clear text password to a byte array. The platform default encoding is the default.

The authentication of a user is performed by connecting to the LDAP server based on the login module configuration options. Connecting to the LDAP server is done by creating an InitialLdapContext with an environment composed of the LDAP JNDI properties described previously in this section. The Context.SECURITY_PRINCIPAL is set to the distinguished name of the user as obtained by the callback handler in combination with the principalDNPrefix and principalDNSuffix option values, and the Context.SECURITY_CREDENTIALS property is either set to the String password or the Object credential depending on the useObjectCredential option.

Once authentication has succeeded by virtue of being able to create an InitialLdapContext instance, the user's roles are queried by performing a search on the rolesCtxDN location with search attributes set to the roleAttributeName and uidAttributeName option values. The roles names are obtaining by invoking the toString method on the role attributes in the search result set.

A sample login configuration entry is:

```
testLdap {
    org.jboss.security.auth.spi.LdapLoginModule required
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url="ldap://ldaphost.jboss.org:1389/"
        java.naming.security.authentication=simple
        principalDNPrefix=uid=
        uidAttributeID=userid
        roleAttributeID=roleName
        principalDNSuffix=,ou=People,o=jboss.org
        rolesCtxDN=cn=JBossSX Tests,ou=Roles,o=jboss.org
};
```

To help you understand all of the options of the LdapLoginModule, consider the sample LDAP server data shown in Figure 8-9. This figure corresponds to the testLdap login configuration just shown.

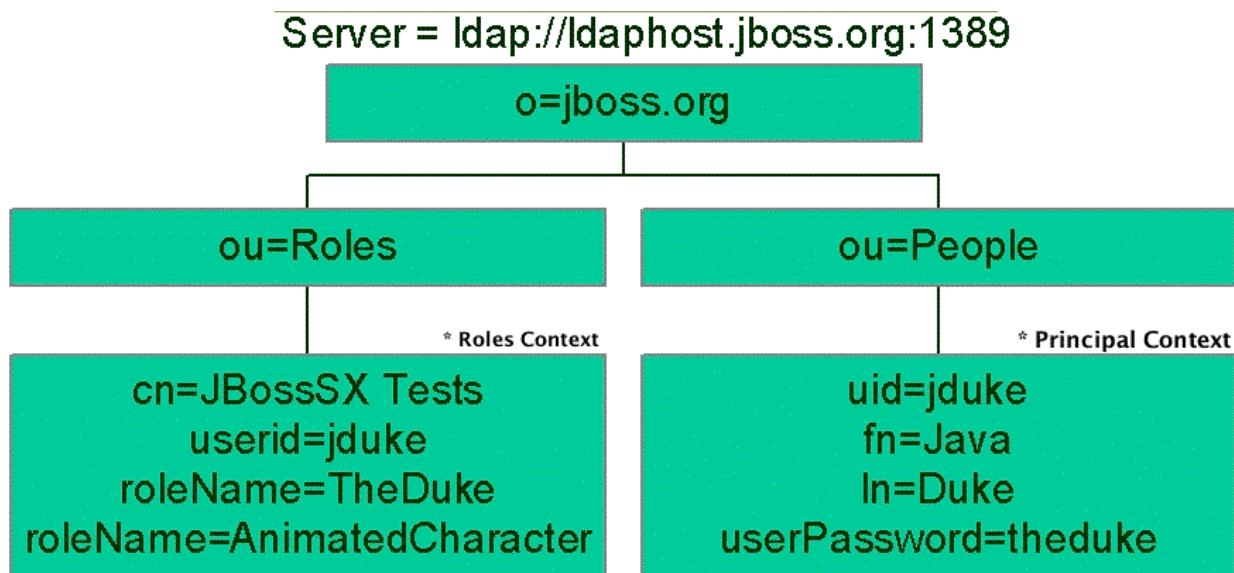


FIGURE 8-9. An LDAP server configuration compatible with the testLdap sample configuration.

Take a look at the testLdap login module configuration in comparison to the Figure 8-9 schema. The `java.naming.factory.initial`, `java.naming.factory.url` and `java.naming.security` options indicate the Sun LDAP JNDI provider implementation will be used, the LDAP server is located on host `ldaphost.jboss.org` on port 1389, and that simple username and password will be used to authenticate clients connecting to the LDAP server.

When the `LdapLoginModule` performs authentication of a user, it does so by connecting to the LDAP server specified by the `java.naming.factory.url`. The `java.naming.security.principal` property is built from the principalDNPrefix, passed in username and principalDNSuffix as described above. For the testLdap configuration example and a username of 'jduke', the `java.naming.security.principal` string would be '`uid=jduke,ou=People,o=jboss.org`'. This corresponds to the LDAP context on the lower right of Figure 8-9 labeled as *Principal Context*. The `java.naming.security.credentials` property would be set to the passed in password and it would have to match the `userPassword` attribute of the *Principal Context*. How a secured LDAP context stores the authentication credential information depends on the LDAP server, so your LDAP server may handle the validation of the `java.naming.security.credentials` property differently.

Once authentication succeeds, the roles on which authorization will be based are retrieved by performing a JNDI search of the LDAP context whose distinguished name is given by the `rolesCtxDN` option value. For the testLdap configuration this is '`cn=JBossSX Tests,ou=Roles,o=jboss.org`' and corresponds to the LDAP context on the lower left of Figure 8-9 labeled *Roles Context*. The search attempts to locate any subcontexts that contain an attribute whose name is given by the `uidAttributeID` option, and whose value matches the username passed to the login module. For any matching context, all values of the attribute whose name is given by the `roleAttributeID` option are obtained. For the testLdap configuration the attribute name that contains the roles is called `roleName`. The resulting `roleName` values are stored in the JAAS Subject associated with the `LdapLoginModule` as the Roles group principals that will be used for role-based authorization. For the LDAP schema shown in Figure 8-9, the roles that will be assigned to the user 'jduke' are 'TheDuke' and 'Animated-Character'.

[org.jboss.security.auth.spi.DatabaseServerLoginModule](#)

The `DatabaseServerLoginModule` is a JDBC based login module that supports authentication and role mapping. You would use this login module if you have your username, password and role information in a JDBC accessible database. The `DatabaseServerLoginModule` is based on two logical tables:

```
Table Principals(PrincipalID text, Password text)
Table Roles(PrincipalID text, Role text, RoleGroup text)
```

The `Principals` table associates the user `PrincipalID` with the valid password and the `Roles` table associates the user `PrincipalID` with its role sets. The roles used for user permissions must be contained in rows with a `RoleGroup` column value of `Roles`. The tables are logical in that you can specify the SQL query that the login module uses. All that is required is that the `java.sql.ResultSet` has the same logical structure as the `Principals` and `Roles` tables described previously. The actual names of the tables

and columns are not relevant as the results are accessed based on the column index. To clarify this notion, consider a database with two tables, Principals and Roles, as already declared. The following statements build the tables to contain a PrincipalID 'java' with a Password of 'echoman' in the Principals table, a PrincipalID 'java' with a role named 'Echo' in the 'Roles' RoleGroup in the Roles table, and a PrincipalID 'java' with a role named 'caller_java' in the 'CallerPrincipal' RoleGroup in the Roles table:

```
INSERT INTO Principals VALUES('java', 'echoman')
INSERT INTO Roles VALUES('java', 'Echo', 'Roles')
INSERT INTO Roles VALUES('java', 'caller_java', 'CallerPrincipal')
```

The supported login module configuration options include the following:

- **dsJndiName**: The JNDI name for the DataSource of the database containing the logical "Principals" and "Roles" tables. If not specified this defaults to "java:/DefaultDS".
- **principalsQuery**: The prepared statement query equivalent to: "select Password from Principals where PrincipalID=?". If not specified this is the exact prepared statement that will be used.
- **rolesQuery**: The prepared statement query equivalent to: "select Role, RoleGroup from Roles where PrincipalID=?". If not specified this is the exact prepared statement that will be used.
- **unauthenticatedIdentity=string**, The principal name that should be assigned to requests that contain no authentication information.
- **password-stacking=useFirstPass**, When password-stacking option is set, this module first looks for a shared username and password under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively in the login module shared state Map. If found these are used as the principal name and password. If not found the principal name and password are set by this login module and stored under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively.
- **hashAlgorithm=string**: The name of the java.security.MessageDigest algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. When hashAlgorithm is specified, the clear text password obtained from the CallbackHandler is hashed before it is passed to UsernamePasswordLoginModule.validatePassword as the inputPassword argument. The expectedPassword as obtained from the database must be comparably hashed.
- **hashEncoding=base64|hex**: The string format for the hashed pass and must be either "base64" or "hex". Base64 is the default.
- **hashCharset=string**: The encoding used to convert the clear text password to a byte array. The platform default encoding is the default

As an example DatabaseServerLoginModule configuration, consider a custom table schema like the following:

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), userRoles VARCHAR(32))
```

The corresponding DatabaseServerLoginModule configuration would be:

```
testDB {
    org.jboss.security.auth.spi.DatabaseServerLoginModule required
        dsJndiName="java:/MyDatabaseDS"
        principalsQuery="select passwd from Users username where username=?"
```

```
    rolesQuery="select userRoles, 'Roles' from UserRoles where username=?"
    ;
}
```

org.jboss.security.auth.spi.ProxyLoginModule

The ProxyLoginModule is a login module that loads a delegate LoginModule using the current thread context class loader. The purpose of this module is to work around the current JAAS 1.0 class loader limitation that requires LoginModules to be on the system classpath. Some custom LoginModules use classes that are loaded from the JBoss server lib/ext directory and these are not available if the LoginModule is placed on the system classpath. To work around this limitation you use the ProxyLoginModule to bootstrap the custom LoginModule. The ProxyLoginModule has one required configuration option called moduleName. It specifies the fully qualified class name of the LoginModule implementation that is to be bootstrapped. Any number of additional configuration options may be specified, and they will be passed to the bootstrapped login module.

As an example, consider a custom login module that makes use of some service that is loaded from the JBoss lib/ext directory. The class name of the custom login module is com.biz.CustomServiceLoginModule. A suitable ProxyLoginModule configuration entry for bootstrapping this custom login module would be:

```
testProxy {
    org.jboss.security.auth.spi.ProxyLoginModule required
        moduleName=com.biz.CustomServiceLoginModule
        customOption1=value1
        customOption2=value2
        customOption3=value3;
}
```

org.jboss.security.auth.spi.RunAsLoginModule

New in JBoss-3.0.3 is a helper login module called RunAsLoginModule. It pushes a run as role for the duration of the login phase of authentication, and pops the run as role in either the commit or abort phase. The purpose of this login module is to provide a role for other login modules that need to access secured resources in order to perform their authentication. An example would be a login module that accesses an secured EJB. This login module must be configured ahead of the login module(s) that need a run as role established.

The only login module configuration option is:

- **roleName**: the name of the role to use as the run as role during login phase. If not specified a default of “nobody” is used.

org.jboss.security.ClientLoginModule

The ClientLoginModule is an implementation of LoginModule for use by JBoss clients for the establishment of the caller identity and credentials. This simply sets the org.jboss.security.SecurityAssociation.principal to the value of the NameCallback filled in by the CallbackHandler, and the org.jboss.security.SecurityAssociation.credential to the value of the PasswordCallback filled in by the

CallbackHandler. This is the only supported mechanism for a client to establish the current thread's caller. Both stand-alone client applications and server environments, acting as JBoss EJB clients where the security environment has not been configured to use JBossSX transparently, need to use the ClientLoginModule. Of course, you could always set the org.jboss.security.SecurityAssociation information directly, but this is considered an internal API that is subject to change without notice.

Note that this login module does not perform any authentication. It merely copies the login information provided to it into the JBoss server EJB invocation layer for subsequent authentication on the server. If you need to perform client-side authentication of users you would need to configure another login module in addition to the ClientLoginModule.

The supported login module configuration options include the following:

- **multi-threaded=true|false**, When the multi-threaded option is set to true, each login thread has its own principal and credential storage. This is useful in client environments where multiple user identities are active in separate threads. When true, each separate thread must perform its own login. When set to false the login identity and credentials are global variables that apply to all threads in the VM. The default for this option is false.
- **password-stacking=useFirstPass**, When password-stacking option is set, this module first looks for a shared username and password using "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively in the login module shared state Map. This allows a module configured prior to this one to establish a valid username and password that should be passed to JBoss. You would use this option if you want to perform client-side authentication of clients using some other login module such as the LdapLoginModule.

A sample login configuration for ClientLoginModule is the default configuration entry found in the JBoss distribution client/auth.conf file. The configuration is:

```
other {
    // Put your login modules that work without jBoss here

    // jBoss LoginModule
    org.jboss.security.ClientLoginModule required;

    // Put your login modules that need jBoss here
};
```

Writing Custom Login Modules

If the login modules bundled with the JBossSX framework do not work with your security environment, you can write your own custom login module implementation that does.

Recall from the section on the JaasSecurityManager architecture that the JaasSecurityManager expected a particular usage pattern of the Subject principals set. You need to understand the JAAS Subject class's information storage features and the expected usage of these features to be able to write a login module that works with the JaasSecurityManager. This section examines this requirement and introduces two abstract base LoginModule implementations that can help you implement your own custom login modules.

You can obtain security information associated with a Subject in six ways using the following methods:

```
java.util.Set getPrincipals()
java.util.Set getPrincipals(java.lang.Class c)
java.util.Set getPrivateCredentials()
java.util.Set getPrivateCredentials(java.lang.Class c)
java.util.Set getPublicCredentials()
java.util.Set getPublicCredentials(java.lang.Class c)
```

For Subject identities and roles, JBossSX has selected the most natural choice: the principals sets obtained via `getPrincipals()` and `getPrincipals(java.lang.Class)`. The usage pattern is as follows:

- User identities (username, social security number, employee ID, and so on) are stored as `java.security.Principal` objects in the Subject Principals set. The Principal implementation that represents the user identity must base comparisons and equality on the name of the principal. A suitable implementation is available as the `org.jboss.security.SimplePrincipal` class. Other Principal instances may be added to the Subject Principals set as needed.
- The assigned user roles are also stored in the Principals set, but they are grouped in named role sets using `java.security.acl.Group` instances. The Group interface defines a collection of Principals and/or Groups, and is a subinterface of `java.security.Principal`. Any number of role sets can be assigned to a Subject. Currently, the JBossSX framework uses two well-known role sets with the names "Roles" and "CallerPrincipal". The "Roles" Group is the collection of Principals for the named roles as known in the application domain under which the Subject has been authenticated. This role set is used by methods like the `EJBContext.isCallerInRole(String)`, which EJBs can use to see if the current caller belongs to the named application domain role. The security interceptor logic that performs method permission checks also uses this role set. The "CallerPrincipal" Group consists of the single Principal identity assigned to the user in the application domain. The `EJBContext.getCallerPrincipal()` method uses the "CallerPrincipal" to allow the application domain to map from the operation environment identity to a user identity suitable for the application. If a Subject does not have a "CallerPrincipal" Group, the application identity is the same as operational environment identity.

Support for the Subject Usage Pattern

To simplify correct implementation of the Subject usage patterns described in the preceding section, JBossSX includes two abstract login modules that handle the population of the authenticated Subject with a template pattern that enforces correct Subject usage. The most generic of the two is the `org.jboss.security.auth.spi.AbstractServerLoginModule` class. It provides a concrete implementation of the `javax.security.auth.spi.LoginModule` interface and offers abstract methods for the key tasks specific to an operation environment security infrastructure. The key details of the class are highlighted in the following class fragment. The Javadoc comments detail the responsibilities of subclasses.

```
package org.jboss.security.auth.spi;
/** This class implements the common functionality required for a
JAAS server-side LoginModule and implements the JBossSX standard
Subject usage pattern of storing identities and roles. Subclass
this module to create your own custom LoginModule and override the
login(), getRoleSets(), and getIdentity() methods.
```

```
/*
public abstract class AbstractServerLoginModule
    implements javax.security.auth.spi.LoginModule
{
    protected Subject subject;
    protected CallbackHandler callbackHandler;
    protected Map sharedState;
    protected Map options;
    protected Logger log;
    /** Flag indicating if the shared credential should be used */
    protected boolean useFirstPass;
    /** Flag indicating if the login phase succeeded. Subclasses that override
     the login method must set this to true on successful completion of login
     */
    protected boolean loginOk;

    ...
/** 
 * Initialize the login module. This stores the subject, callbackHandler
 * and sharedState and options for the login session. Subclasses should
override
 * if they need to process their own options. A call to super.initialize(...)
 * must be made in the case of an override.
 * <p>
 * The options are checked for the <em>password-stacking</em> parameter.
 * If this is set to "useFirstPass", the login identity will be taken from the
 * <code>javax.security.auth.login.name</code> value of the sharedState map,
 * and the proof of identity from the
 * <code>javax.security.auth.login.password</code> value of the sharedState
map.
 *
 * @param subject the Subject to update after a successful login.
 * @param callbackHandler the CallbackHandler that will be used to obtain the
 *          the user identity and credentials.
 * @param sharedState a Map shared between all configured login module
instances
 * @param options the parameters passed to the login module.
 */
public void initialize(Subject subject,
    CallbackHandler callbackHandler,
    Map sharedState,
    Map options)
{
    ...
}

/** Looks for javax.security.auth.login.name and
javax.security.auth.login.password
values in the sharedState map if the useFirstPass option was true and returns
true if they exist. If they do not or are null this method returns false.
```

Note that subclasses that override the login method must set the loginOk ivar to true if the login succeeds in order for the commit phase to populate the Subject. This implementation sets loginOk to true if the login() method returns true, otherwise, it sets loginOk to false.

```

        */
    public boolean login() throws LoginException
    {
        ...
    }

    /** Overridden by subclasses to return the Principal that
     * corresponds to the user primary identity.
    */
    abstract protected Principal getIdentity();

    /** Overridden by subclasses to return the Groups that
     * correspond to the role sets assigned to the user. Subclasses
     * should create at least a Group named "Roles" that contains
     * the roles assigned to the user.
     * A second common group is "CallerPrincipal," which provides
     * the application identity of the user rather than the security
     * domain identity.
    @return Group[] containing the sets of roles
    */
    abstract protected Group[] getRoleSets() throws LoginException;
}

```

One key change in JBoss-3.0.3 was the addition of the *loginOk* instance variable. This must be set to true if the login succeeds, false otherwise by any subclasses that override the login method. Failure to set this variable correctly will result in the commit method either not updating the Subject when it should, or updating the Subject when it should not. Tracking the outcome of the login phase was added to allow login module to be chained together with control flags that do not require that the login module succeed in order for the overall login to succeed.

The second abstract base login module suitable for custom login modules is the [org.jboss.security.auth.spi.UsernamePasswordLoginModule](#). The login module further simplifies custom login module implementation by enforcing a string-based username as the user identity and a char[] password as the authentication credential. It also supports the mapping of anonymous users (indicated by a null username and password) to a Principal with no roles. The key details of the class are highlighted in the following class fragment. The Javadoc comments detail the responsibilities of subclasses.

```

package org.jboss.security.auth.spi;
/** An abstract subclass of AbstractServerLoginModule that imposes
 * a an identity == String username, credentials == String password
 * view on the login process. Subclasses override the
 * getUsersPassword() and getUsersRoles() methods to return the
 * expected password and roles for the user.
*/
public abstract class UsernamePasswordLoginModule
    extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password

```

```
    are seen */
    private Principal unauthenticatedIdentity;
    /** The message digest algorithm used to hash passwords. If null then
     plain passwords will be used. */
    private String hashAlgorithm = null;
    /** The name of the charset/encoding to use when converting the password
     String to a byte array. Default is the platform's default encoding.
     */
    private String hashCharset = null;
    /** The string encoding format to use. Defaults to base64. */
    private String hashEncoding = null;

    ...

    /** Override the superclass method to look for an
     unauthenticatedIdentity property. This method first invokes
     the super version.
    @param options,
        @option unauthenticatedIdentity: the name of the principal
        to assign and authenticate when a null username and password
        are seen.
    */
    public void initialize(Subject subject,
        CallbackHandler callbackHandler,
        Map sharedState,
        Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
            options);
        // Check for unauthenticatedIdentity option.
        Object option = options.get("unauthenticatedIdentity");
        String name = (String) option;
        if( name != null )
            unauthenticatedIdentity = new SimplePrincipal(name);
    }

    ...

    /** A hook that allows subclasses to change the validation of
     the input password against the expected password. This version
     checks that neither inputPassword or expectedPassword are null
     and that inputPassword.equals(expectedPassword) is true;
     @return true if the inputPassword is valid, false otherwise.
    */
    protected boolean validatePassword(String inputPassword,
        String expectedPassword)
    {
        if( inputPassword == null || expectedPassword == null )
            return false;
        return inputPassword.equals(expectedPassword);
    }

    /** Get the expected password for the current username
     available via the getUsername() method. This is called from
     within the login() method after the CallbackHandler has
```

```
        returned the username and candidate password.  
        @return the valid password String  
        */  
        abstract protected String getUsersPassword()  
            throws LoginException;  
    }
```

The choice of subclassing the [AbstractServerLoginModule](#) versus [UsernamePasswordLoginModule](#) is simply based on whether a [String](#) based username and [String](#) credential are usable for the authentication technology you are writing the login module for. If the string based semantic is valid, then subclass [UsernamePasswordLoginModule](#), else subclass [AbstractServerLoginModule](#).

The steps you are required to perform when writing a custom login module are summarized in the following depending on which base login module class you choose. When writing a custom login module that integrates with your security infrastructure, you should start by subclassing [AbstractServerLoginModule](#) or [UsernamePasswordLoginModule](#) to ensure that your login module provides the authenticated [Principal](#) information in the form expected by the JBossSX security manager.

When subclassing the [AbstractServerLoginModule](#), you need to override the following:

- `void initialize(Subject, CallbackHandler, Map, Map);` if you have custom options to parse.
- `boolean login();` to perform the authentication activity. Be sure to set the `loginOk` instance variable to true if login succeeds, false if it fails.
- `Principal getIdentity();` to return the Principal object for the user authenticated by the `log()` step.
- `Group[] getRoleSets();` to return at least one [Group](#) named "Roles" that contains the roles assigned to the [Principal](#) authenticated during `login()`. A second common [Group](#) is named "CallerPrincipal" and provides the user's application identity rather than the security domain identity.

When subclassing the [UsernamePasswordLoginModule](#), you need to override the following:

- `void initialize(Subject, CallbackHandler, Map, Map);` if you have custom options to parse.
- `Group[] getRoleSets();` to return at least one [Group](#) named "Roles" that contains the roles assigned to the [Principal](#) authenticated during `login()`. A second common [Group](#) is named "CallerPrincipal" and provides the user's application identity rather than the security domain identity.
- `String getUsersPassword();` to return the expected password for the current user-name available via the `getUsername()` method. The `getUsersPassword()` method is called from within `login()` after the [CallbackHandler](#) returns the username and candidate password.

A Custom LoginModule Example

In this section we will develop a custom login module example. It will extend the [UsernamePasswordLoginModule](#) and obtains a user's password and role names from a JNDI lookup. The idea is that there is a JNDI context that will return a user's password if you perform a lookup on the context

using a name of the form “password/<username>” where <username> is the current user being authenticated. Similary, a lookup of the form “roles/<username>” returns the requested user’s roles.

The source code for the example is located in the src/main/org/jboss/chap8/ex2 directory of the book examples. Listing 8-10 shows the source code for the JndiUserAndPass custom login module. Note that because this extends the JBoss UsernamePasswordLoginModule, all the JndiUserAndPass does is obtain the user’s password and roles from the JNDI store. The JndiUserAndPass does not concern itself with the JAAS LoginModule operations.

LISTING 8-10. A JndiUserAndPass custom login module

```
package org.jboss.chap8.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/** An example custom login module that obtains passwords and roles for a user
from a JNDI lookup.

@author Scott.Stark@jboss.org
@version $Revision$
*/
public class JndiUserAndPass extends UsernamePasswordLoginModule
{
    /** The JNDI name to the context that handles the password/<username> lookup */
    private String userPathPrefix;
    /** The JNDI name to the context that handles the roles/<username> lookup */
    private String rolesPathPrefix;

    /** Override to obtain the userPathPrefix and rolesPathPrefix options.
     */
    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options)
    {
        super.initialize(subject, callbackHandler, sharedState, options);
        userPathPrefix = (String) options.get("userPathPrefix");
        rolesPathPrefix = (String) options.get("rolesPathPrefix");
    }

    /** Get the roles the current user belongs to by querying the
     rolesPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected Group[] getRoleSets() throws LoginException
    {
```

```
try
{
    InitialContext ctx = new InitialContext();
    String rolesPath = rolesPathPrefix + '/' + super.getUsername();
    String[] roles = (String[]) ctx.lookup(rolesPath);
    Group[] groups = {new SimpleGroup("Roles")};
    log.info("Getting roles for user=" + super.getUsername());
    for(int r = 0; r < roles.length; r++)
    {
        SimplePrincipal role = new SimplePrincipal(roles[r]);
        log.info("Found role=" + roles[r]);
        groups[0].addMember(role);
    }
    return groups;
}
catch(NamingException e)
{
    log.error("Failed to obtain groups for user=" + super.getUsername(), e);
    throw new LoginException(e.toString(true));
}
}

/** Get the password of the current user by querying the
userPathPrefix + '/' + super.getUsername() JNDI location.
*/
protected String getUsersPassword() throws LoginException
{
    try
    {
        InitialContext ctx = new InitialContext();
        String userPath = userPathPrefix + '/' + super.getUsername();
        log.info("Getting password for user=" + super.getUsername());
        String passwd = (String) ctx.lookup(userPath);
        log.info("Found password=" + passwd);
        return passwd;
    }
    catch(NamingException e)
    {
        log.error("Failed to obtain password for user=" + super.getUsername(), e);
        throw new LoginException(e.toString(true));
    }
}
}
```

The details of the JNDI store are found in the [org.jboss.chap8.ex2.service.JndiStore](#) MBean. This service binds an [ObjectFactory](#) that returns a javax.naming.Context proxy into JNDI. The proxy handles lookup operations done against it by checking the prefix of the lookup name against “password” and “roles”. When the name begins with “password”, a user’s password is being requested. When the name begins with “roles” the user’s roles are being requested. The example implementation always returns a password of “thedupe” and an array of roles names equal to {“TheDuke”, “Echo”} regardless of what the username is. You can experiment with other implementations as you wish.

The example code includes a simple session bean for testing the custom login module. To build, deploy and run the example, execute the following command from the examples directory. Make sure you have the JBoss server running. The key lines from the client are given in Listing 8-11 while the server side operation of the is shown in Listing 8-12.

LISTING 8-11. The chap8-ex2 secured client access output

```
examples 940>ant -Dchap=8 -Dex=2 run-example
Buildfile: build.xml
...
run-example2:
    [copy] Copying 1 file to G:\JBossReleases\jboss-
3.0.1RC1\server\default\deploy
    [echo] Waiting for 5 seconds for deploy...
    [java] [INFO,ExClient] Login with username=jduke, password=theduke
    [java] [INFO,ExClient] Looking up EchoBean2
    [java] [INFO,ExClient] Created Echo
    [java] [INFO,ExClient] Echo.echo('Hello') = Hello

BUILD SUCCESSFUL

Total time: 12 seconds
```

LISTING 8-12. The chap8-ex2 server side behavior of the JndiUserAndPass

```
10:22:09,828 INFO  [MainDeployer] Starting deployment of package: file:/G:/JBoss
Releases/jboss-3.0.1RC1/server/default/deploy/chap8-ex2.jar
10:22:09,921 INFO  [SecurityConfig] Creating
10:22:09,921 INFO  [SecurityConfig] Created
10:22:10,156 INFO  [EjbModule] Creating
10:22:10,187 INFO  [EjbModule] Deploying EchoBean2
10:22:10,250 INFO  [JaasSecurityManagerService] Created securityMgr=org.jboss.se
curity.plugins.JaasSecurityManager@27f9dc
10:22:10,250 INFO  [JaasSecurityManagerService] setCachePolicy, c=org.jboss.util
.TimedCachePolicy@4e28b
10:22:10,250 INFO  [JaasSecurityManagerService] Added chap8-ex2, org.jboss.secur
ity.plugins.SecurityDomainContext@354362 to map
10:22:10,265 INFO  [EjbModule] Created
10:22:10,281 INFO  [JndiStore] Start, bound security/store
10:22:10,281 INFO  [SecurityConfig] Starting
10:22:10,281 INFO  [SecurityConfig] Using JAAS AuthConfig: jar:file:/G:/JBossRel
eases/jboss-3.0.1RC1/server/default/tmp/deploy/server/default/deploy/chap8-
ex2.j
ar/60.chap8-ex2.jar-contents/chap8-ex2.sar!/META-INF/login-config.xml
10:22:10,312 INFO  [SecurityConfig] Started
10:22:10,312 INFO  [EjbModule] Starting
10:22:10,390 INFO  [EjbModule] Started
10:22:10,390 INFO  [MainDeployer] Deployed package: file:/G:/JBossReleases/jboss
-3.0.1RC1/server/default/deploy/chap8-ex2.jar
10:22:15,343 INFO  [JndiUserAndPass] Getting password for user=jduke
10:22:15,390 INFO  [JndiStore] lookup, name=password/jduke
10:22:15,390 INFO  [JndiUserAndPass] Found password=theduke
```

```
10:22:15,390 INFO [JndiStore] lookup, name=roles/jduke
10:22:15,390 INFO [JndiUserAndPass] Getting roles for user=jduke
10:22:15,390 INFO [JndiUserAndPass] Found role=TheDuke
10:22:15,390 INFO [JndiUserAndPass] Found role=Echo
```

The choice of using the `JndiUserAndPass` custom login module for the server side authentication of the user is determined by the login configuration for the example security domain. The ejb-jar META-INF/jboss.xml descriptor sets the security domain and the sar META-INF/login-config.xml descriptor defines the login module configuration. The contents of these descriptors are shown in Listing 8-13.

LISTING 8-13. The chap8-ex2 security domain and login module configuration

The chap8-ex2 jboss.xml descriptor security domain settings

```
<?xml version="1.0"?>
<jboss>
    <security-domain>java:/jaas/chap8-ex2</security-domain>
</jboss>
```

The login-config.xml configuration fragment for the chap8-ex2 application

```
<application-policy name = "chap8-ex2">
    <authentication>
        <login-module code = "org.jboss.chap8.ex2.JndiUserAndPass"
                      flag = "required">
            <module-option name = "userPathPrefix">/security/store/password</module-
option>
            <module-option name = "rolesPathPrefix">/security/store/roles</module-
option>
        </login-module>
    </authentication>
</application-policy>
```

The Secure Remote Password (SRP) Protocol

The SRP protocol is an implementation of a public key exchange handshake described in the Internet standards working group request for comments 2945(RFC2945). The RFC2945 abstract states:

This document describes a cryptographically strong network authentication mechanism known as the Secure Remote Password (SRP) protocol. This mechanism is suitable for negotiating secure connections using a user-supplied password, while eliminating the security problems traditionally associated with reusable passwords. This system also performs a secure key exchange in the process of authentication, allowing security layers (privacy and/or integrity protection) to be enabled during the session. Trusted key servers and certificate infrastructures are not required, and cli-

ents are not required to store or manage any long-term keys. SRP offers both security and deployment advantages over existing challenge-response techniques, making it an ideal drop-in replacement where secure password authentication is needed.

Note: The complete RFC2945 specification can be obtained from <http://www.rfc-editor.org/rfc.html>. Additional information on the SRP algorithm and its history can be found here: <http://www-cs-students.stanford.edu/~tjw/srp/>.

SRP is similar in concept and security to other public key exchange algorithms, such as Diffie-Hellman and RSA. SRP is based on simple string passwords in a way that does not require a clear text password to exist on the server. This is in contrast to other public key-based algorithms that require client certificates and the corresponding certificate management infrastructure.

Algorithms like Diffie-Hellman and RSA are known as public key exchange algorithms. The concept of public key algorithms is that you have two keys, one public that is available to everyone, and one that is private and known only to you. When someone wants to send encrypted information to you, then encrypt the information using your public key. Only you are able to decrypt the information using your private key. Contrast this with the more traditional shared password based encryption schemes that require the sender and receiver to know the shared password. Public key algorithms eliminate the need to share passwords. For more information on public key algorithms as well as numerous other cryptographic algorithms, see "Applied Cryptography, Second Edition" by Bruce Schneier, ISBN 0-471-11709-9.

The JBossSX framework includes an implementation of SRP that consists of the following elements:

- An implementation of the SRP handshake protocol that is independent of any particular client/server protocol
- An RMI implementation of the handshake protocol as the default client/server SRP implementation
- A client side JAAS LoginModule implementation that uses the RMI implementation for use in authenticating clients in a secure fashion
- A JMX MBean for managing the RMI server implementation. The MBean allows the RMI server implementation to be plugged into a JMX framework and externalizes the configuration of the verification information store. It also establishes an authentication cache that is bound into the JBoss server JNDI namespace.
- A server side JAAS LoginModule implementation that uses the authentication cache managed by the SRP JMX MBean.

Figure 8-10 gives a diagram of the key components involved in the JBossSX implementation of the SRP client/server framework.

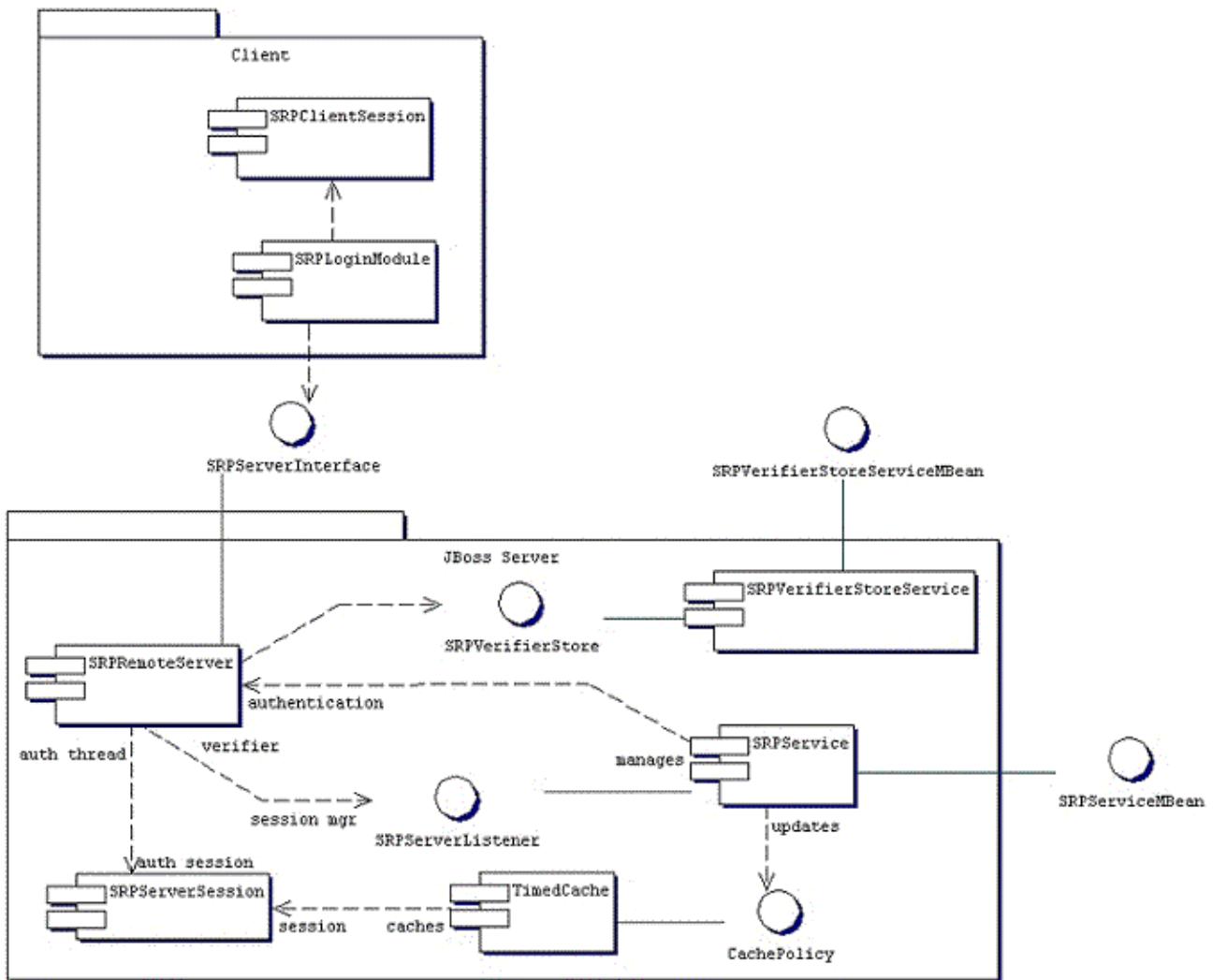


FIGURE 8-10. The JBossSX components of the SRP client-server framework.

On the client side, SRP shows up as a custom JAAS `LoginModule` implementation that communicates to the authentication server through an `org.jboss.security.srp.SRPServerInterface` proxy. A client enables authentication using SRP by creating a login configuration entry that includes the `org.jboss.security.srp.jaas.SRPLoginModule`. This module supports the following configuration options:

- **principalClassName:** This option is no longer supported. The principal class is now always `org.jboss.security.srp.jaas.SRPPrincipal`.
- **srpServerJndiName:** The JNDI name of the `SRPServerInterface` object to use for communicating with the SRP authentication server. If both `srpServerJndiName` and `srpServerRmiUrl` options are specified, the `srpServerJndiName` is tried before `srpServerRmiUrl`.
- **srpServerRmiUrl:** The RMI protocol URL string for the location of the `SRPServerInterface` proxy to use for communicating with the SRP authentication server.

- **externalRandomA**: A true/false flag indicating if the random component of the client public key A should come from the user callback. This can be used to input a strong cryptographic random number coming from a hardware token for example.
- **hasAuxChallenge**: A true/false flag indicating an that a string will be sent to the server as an additional challenge for the server to validate. If the client session supports an encryption cipher then a temporary cipher will be created using the session private key and the challenge object sent as a javax.crypto.SealedObject.
- **multipleSessions**: a true/false flag indicating if a given client may have multiple SRP login sessions active simultaneously.

The [SRPLoginModule](#) needs to be configured along with the standard [ClientLoginModule](#) to allow the SRP authentication credentials to be used for validation of access to security J2EE components. An example login configuration entry that demonstrates such a setup is:

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
        srpServerJndiName="SRPServerInterface"
        ;
    org.jboss.security.ClientLoginModule required
        password-stacking="useFirstPass"
        ;
}
```

On the JBoss server side, there are two MBeans that manage the objects that collectively make up the SRP server. The primary service is the [org.jboss.security.srp.SRPServer](#) MBean, and it is responsible for exposing an RMI accessible version of the [SRPServerInterface](#) as well as updating the SRP authentication session cache. The configurable [SRPServer](#) MBean attributes include the following:

- **JndiName**: The JNDI name from which the [SRPServerInterface](#) proxy should be available. This is the location where the [SRPServer](#) binds the serializable dynamic proxy to the [SRPServerInterface](#). If not specified it defaults to "srp/SRPServerInterface".
- **VerifierSourceJndiName**: The JNDI name of the [SRPVerifierSource](#) implementation that should be used by the SRPServer. If not set it defaults to "srp/DefaultVerifierSource".
- **AuthenticationCacheJndiName**: The JNDI name under which the authentication [org.jboss.util.CachePolicy](#) implementation to be used for caching authentication information is bound. The SRP session cache is made available for use through this binding. If not specified it defaults to "srp/AuthenticationCache".
- **ServerPort**: RMI port for the [SRPRemoteServerInterface](#). If not specified it defaults to 10099.
- **ClientSocketFactory**: An optional custom [java.rmi.server.RMIClientSocketFactory](#) implementation class name used during the export of the [SRPServerInterface](#). If not specified the default [RMIClientSocketFactory](#) is used.
- **ServerSocketFactory**: An optional custom [java.rmi.server.RMIServerSocketFactory](#) implementation class name used during the export of the [SRPServerInterface](#). If not specified the default [RMIServerSocketFactory](#) is used.
- **AuthenticationCacheTimeout**: Specifies the timed cache policy timeout in seconds. If not specified this defaults to 1800 seconds(30 minutes).

- **AuthenticationCacheResolution:** Specifies the timed cache policy resolution in seconds. This controls the interval between checks for timeouts. If not specified this defaults to 60 seconds(1 minute).

The one input setting is the VerifierSourceJndiName attribute. This is the location of the SRP password information store implementation that must be provided and made available through JNDI. The org.jboss.security.srp SRPVerifierStoreService is an example MBean service that binds an implementation of the SRPVerifierStore interface that uses a file of serialized objects as the persistent store. Although not realistic for a production environment, it does allow for testing of the SRP protocol and provides an example of the requirements for an SRPVerifierStore service. The configurable SRPVerifierStoreService MBean attributes include the following:

- **JndiName:** The JNDI name from which the SRPVerifierStore implementation should be available. If not specified it defaults to "srp/DefaultVerifierSource".
- **StoreFile:** The location of the user password verifier serialized object store file. This can be either a URL or a resource name to be found in the classpath. If not specified it defaults to "SRPVerifierStore.ser".

The SRPVerifierStoreService MBean also supports addUser and delUser operations for addition and deletion of users. The signatures are:

```
public void addUser(String username, String password) throws IOException;
public void delUser(String username) throws IOException;
```

An example configuration of these services is presented in the section “The Secure Remote Password (SRP) Protocol” on page 286.

Providing Password Information for SRP

The default implementation of the SRPVerifierStore interface is not likely to be suitable for you production security environment as it requires all password hash information to be available as a file of serialized objects. You need to provide an MBean service that provides an implementation of the SRPVerifierStore interface that integrates with your existing security information stores. The SRPVerifierStore interface is shown in .

LISTING 8-14. The SRPVerifierStore interface

```
package org.jboss.security.srp;

import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
{
    public static class VerifierInfo implements Serializable
    {
        /** The username the information applies to. Perhaps redundant but it
```

```
    makes the object self contained.  
 */  
 public String username;  
 /** The SRP password verifier hash */  
 public byte[] verifier;  
 /** The random password salt originally used to verify the password */  
 public byte[] salt;  
 /** The SRP algorithm primitive generator */  
 public byte[] g;  
 /** The algorithm safe-prime modulus */  
 public byte[] N;  
}  
  
/** Get the indicated user's password verifier information.  
 */  
public VerifierInfo getUserVerifier(String username)  
throws KeyException, IOException;  
/** Set the indicated users' password verifier information. This is equivalent  
to changing a user's password and should generally invalidate any existing  
SRP sessions and caches.  
 */  
public void setUserVerifier(String username, VerifierInfo info)  
throws IOException;  
  
/** Verify an optional auxillary challenge sent from the client to the server.  
The auxChallenge object will have been decrypted if it was sent encrypted from  
the client. An example of a auxillary challenge would be the validation of a  
hardware token (SafeWord, SecureID, iButton) that the server validates to  
further strengthen the SRP password exchange.  
 */  
public void verifyUserChallenge(String username, Object auxChallenge)  
throws SecurityException;  
}
```

The primary function of a SRPVerifierStore implementation is to provide access to the SRPVerifierStore.VerifierInfo object for a given username. The `getUserVerifier(String)` method is called by the SRPService at that start of a user SRP session to obtain the parameters needed by the SRP algorithm. The elements of the VerifierInfo objects are:

- **username:** The user's name or id used to login.
- **verifier:** This is the one-way hash of the password or PIN the user enters as proof of their identity. The org.jboss.security.Util class has a `calculateVerifier` method that performs that password hashing algorithm. The output password $H(salt \mid H(username \mid ! \mid password))$ as defined by RFC2945. Here H is the SHA secure hash function. The username is converted from a string to a `byte[]` using the UTF-8 encoding.
- **salt:** This is a random number used to increase the difficulty of a brute force dictionary attack on the verifier password database in the event that the database is compromised. It is a value that should be generated from a cryptographically strong random number algorithm when the user's existing clear-text password is hashed.
- **g:** The SRP algorithm primitive generator. In general this can be a well known fixed parameter rather than a per-user setting. The org.jboss.security.srp.SRPConf utility class provides several

settings for g including a good default which can be obtained via `SRPConf.getDefaultParams().g()`.

- **N:** The SRP algorithm safe-prime modulus . In general this can be a well known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for N including a good default which can be obtained via `SRPConf.getDefaultParams().N()`.

So, step 1 of integrating your existing password store is the creation of a hashed version of the password information. If your passwords are already stored in an irreversible hashed form, then this can only be done on a per-user basis as part of an upgrade procedure for example. Note that the `setUserVerifier(String, VerifierInfo)` method is not used by the current `SRPSerivce` and may be implemented as noop method, or even one that throws an exception stating that the store is read-only.

Step 2 is the creation of the custom `SRPVerifierStore` interface implementation that knows how to obtain the `VerifierInfo` from the store you created in step 1. The `verifyUserChallenge(String, Object)` method of the interface is only called if the client `SRPLoginModule` configuration specifies the `hasAuxChallenge` option. This can be used to integrate existing hardware token based schemes like SafeWord or Radius into the SRP algorithm.

Step 3 is the creation of an MBean that makes the step 2 implementation of the `SRPVerifierStore` interface available via JNDI, and exposes any configurable parameters you need. In addition to the default `org.jboss.security.srp.SRPVerifierStoreService` example, the SRP example presented later in this chapter provides a Java properties file based `SRPVerifierStore` implementation. Between the two examples you should have enough to integrate your security store.

Inside of the SRP algorithm

The appeal of the SRP algorithm is that it allows for mutual authentication of client and server using simple text passwords without a secure communication channel. You might be wondering how this is done. Figure 8-11 presents a sequence diagram of the authentication protocol as implemented by JBossSX.

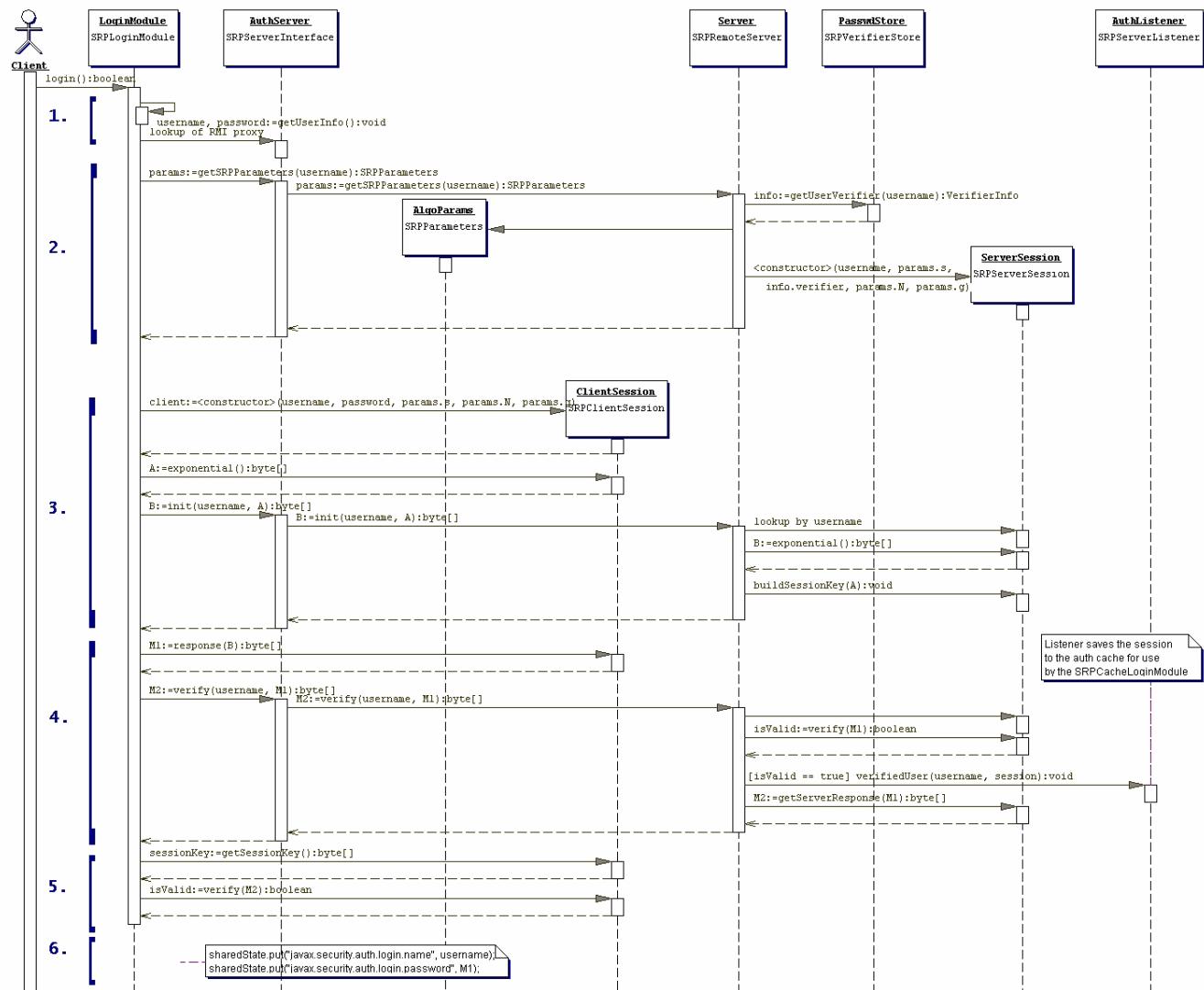


FIGURE 8-11. The SRP client-server authentication algorithm sequence diagram.

The highlights of what is taking place for the key message exchanges presented in Figure 8-11 are as follows. If you want the complete details and theory behind the algorithm, refer to the SRP references mentioned in a note earlier. There are six steps that are performed to complete authentication:

1. The client side SRPLoginModule retrieves the SRPServerInterface instance for the remote authentication server from the naming service.
2. The client side SRPLoginModule next requests the SRP parameters associated with the user-name attempting the login. There are a number of parameters involved in the SRP algorithm that must be chosen when the user password is first transformed into the verifier form used by the SRP algorithm. Rather than hard-coding the parameters (which could be done with minimal security risk), the JBossSX implementation allows a user to retrieve this information as part of the exchange protocol. The getSRPParameters(username) call retrieves the SRP parameters for the given user-name.

3. The client side SRPLoginModule begins an SRP session by creating an SRPClientSession object using the login username, clear-text password, and SRP parameters obtained from step 2. The client then creates a random number A that will be used to build the private SRP session key. The client then initializes the server side of the SRP session by invoking the SRPServerInterface.init method and passes in the username and client generated random number A. The server returns its own random number B. This step corresponds to the exchange of public keys.
4. The client side SRPLoginModule obtains the private SRP session key that has been generated as a result of the previous messages exchanges. This is saved as a private credential in the login Subject. The server challenge response M2 from step 4 is verified by invoking the SRPClientSession.verify method. If this succeeds, mutual authentication of the client to server, and server to client have been completed. The client side SRPLoginModule next creates a challenge M1 to the server by invoking SRPClientSession.response method passing the server random number B as an argument. This challenge is sent to the server via the SRPServerInterface.verify method and server's response is saved as M2. This step corresponds to an exchange of challenges. At this point the server has verified that the user is who they say they are.
5. The client side SRPLoginModule saves the login username and M1 challenge into the LoginModule sharedState Map. This is used as the Principal name and credentials by the standard JBoss ClientLoginModule. The M1 challenge is used in place of the password as proof of identity on any method invocations on J2EE components. The M1 challenge is a cryptographically strong hash associated with the SRP session. Its interception via a third party cannot be used to obtain the user's password.
6. At the end of this authentication protocol, the SRPService has been placed into the SRPService authentication cache for subsequent use by the SRPCacheLoginModule

Although SRP has many interesting properties, it is still an evolving component in the JBossSX framework and has some limitations of which you should be aware. Issues of note include the following:

- Because of how JBoss detaches the method transport protocol from the component container where authentication is performed, an unauthorized user could snoop the SRP M1 challenge and effectively use the challenge to make requests as the associated username. Custom interceptors that encrypt the challenge using the SRP session key can be used to prevent this issue.
- The SRPService maintains a cache of SRP sessions that time out after a configurable period. Once they time out, any subsequent J2EE component access will fail because there is currently no mechanism for transparently renegotiating the SRP authentication credentials. You must either set the authentication cache timeout very long (up to 2,147,483,647 seconds, or approximately 68 years), or handle re-authentication in your code on failure.
- By default there can only be one SRP session for a given username. Because the negotiated SRP session produces a private session key that can be used for encryption/decryption between the client and server, the session is effectively a stateful one. Support for multiple SRP sessions per user has been added as of JBoss-3.0.3, but you cannot encrypt data with one session key and then decrypt it with another.

To use end-to-end SRP authentication for J2EE component calls, you need to configure the security domain under which the components are secured to use the org.jboss.security.srp.jaas.SRPCacheLoginModule. The SRPCacheLoginModule has a single configuration option named `cacheJndiName` that sets the JNDI location of the SRP authentication CachePolicy instance. This must correspond to the

AuthenticationCacheJndiName attribute value of the SRPService MBean. The SRPCacheLoginModule authenticates user credentials by obtaining the client challenge from the SRPServerSession object in the authentication cache and comparing this to the challenge passed as the user credentials. Figure 8-12 illustrates the operation of the SRPCacheLoginModule.login method implementation.

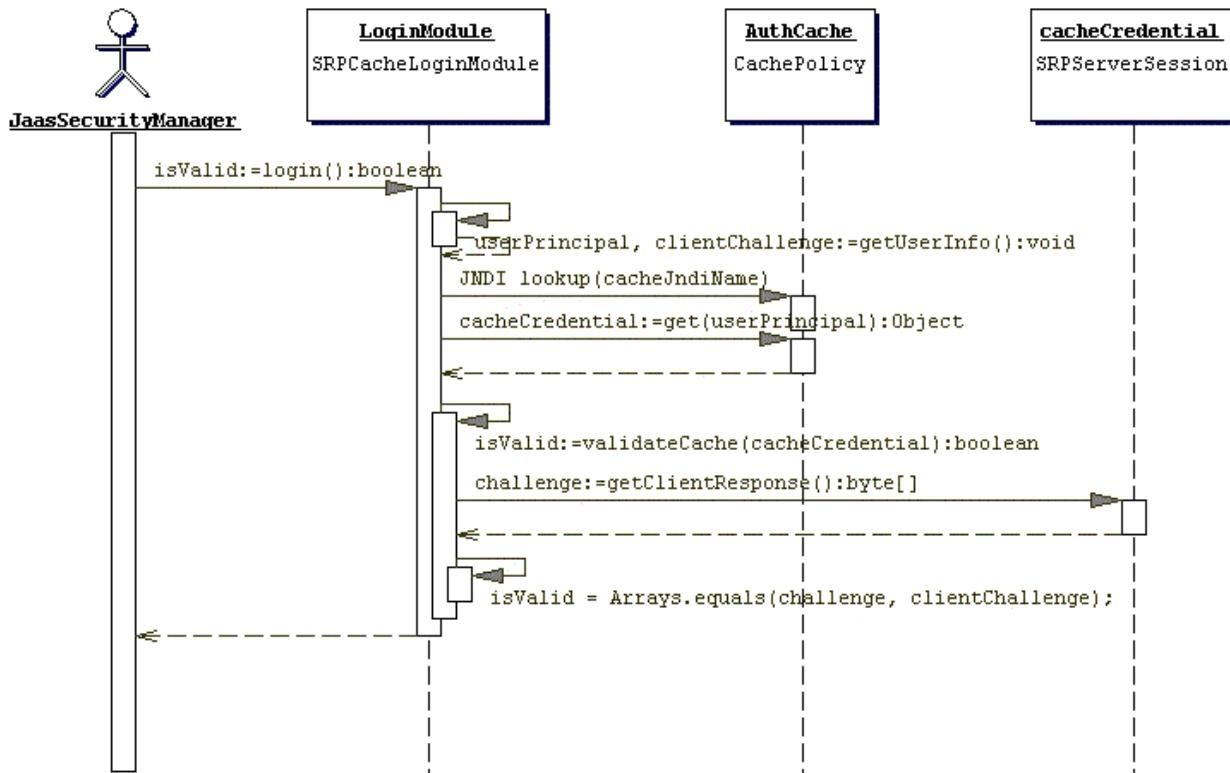


FIGURE 8-12. A sequence diagram illustrating the interaction of the SRPCacheLoginModule with the SRP session cache.

An SRP example

We have covered quite a bit of material on SRP and now its time to demonstrate SRP in practice with an example. The example demonstrates client side authentication of the user via SRP as well as subsequent secured access to a simple EJB using the SRP session challenge as the user credential. The test code deploys an ejb-jar that includes a sar for the configuration of the server side login module configuration and SRP services. As in the previous examples we will dynamically install the server side login module configuration using the SecurityConfig MBean. In this example we also use a custom implementation of the SRPVerifierStore interface that uses an in memory store that is seeded from a Java properties file rather than a serialized object store as used by the SRPVerifierStoreService. This custom service is org.jboss.chap8.ex3.service.PropertiesVerifierStore. Listing 8-15 shows the contents of the jar that contains the example EJB and SRP services.

LISTING 8-15. The chap8-ex3 jar contents

```
examples 992>java -cp output/classes ListJar output/chap8/chap8-ex3.jar
```

```
output/chap8/chap8-ex3.jar
+- META-INF/MANIFEST.MF
+- META-INF/ejb-jar.xml
+- META-INF/jboss.xml
+- org/jboss/chap8/ex3/Echo.class
+- org/jboss/chap8/ex3/EchoBean.class
+- org/jboss/chap8/ex3/EchoHome.class
+- roles.properties
+- users.properties
+- chap8-ex3.sar (archive)
| +- META-INF/MANIFEST.MF
| +- META-INF/jboss-service.xml
| +- META-INF/login-config.xml
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStore$1.class
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStore.class
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStoreMBean.class
| +- org/jboss/chap8/service/SecurityConfig.class
| +- org/jboss/chap8/service/SecurityConfigMBean.class
```

The key SRP related items in this example are the SRP MBean services configuration, and the SRP login module configurations. The jboss-service.xml descriptor of the chap8-ex3.sar is given in Listing 8-16, while Listing 8-17 gives the example client side and server side login module configurations.

LISTING 8-16. The chap8-ex3.sar jboss-service.xml descriptor for the SRP services

```
<server>
  <!-- The custom JAAS login configuration that installs
       a Configuration capable of dynamically updating the
       config settings
  -->
  <mbean code="org.jboss.chap8.service.SecurityConfig"
         name="jboss.docs.chap8:service=LoginConfig-EX3">
    <attribute name="AuthConfig">META-INF/login-config.xml</attribute>
    <attribute name="SecurityConfigName">jboss.security:name=SecurityConfig</
attribute>
  </mbean>

  <!-- The SRP service that provides the SRP RMI server and server side
      authentication cache -->
  <mbean code="org.jboss.security.srp.SRPService"
         name="jboss.docs.chap8:service=SRPService">
    <attribute name="VerifierSourceJndiName">srp-test/chap8-ex3</attribute>
    <attribute name="JndiName">srp-test/SRPServerInterface</attribute>
    <attribute name="AuthenticationCacheJndiName">srp-test/AuthenticationCache</
attribute>
    <attribute name="ServerPort">0</attribute>
    <depends>jboss.docs.chap8:service=PropertiesVerifierStore</depends>
  </mbean>

  <!-- The SRP store handler service that provides the user password verifier
      information -->
  <mbean code="org.jboss.chap8.ex3.service.PropertiesVerifierStore"
```

```
        name="jboss.docs.chap8:service=PropertiesVerifierStore">
    <attribute name="JndiName">srp-test/chap8-ex3</attribute>
</mbean>
</server>
```

LISTING 8-17. The chap8-ex3 client side and server side SRP login module configurations

```
// Client side standard JAAS configuration fragment
srp {
    org.jboss.security.srp.jaas.SRPLLoginModule required
        srpServerJndiName="srp-test/SRPServerInterface"
    ;
    org.jboss.security.ClientLoginModule required
        password-stacking="useFirstPass"
    ;
};

// Server side XMLLoginConfig configuration fragment
<application-policy name = "chap8-ex3">
    <authentication>
        <login-module code = "org.jboss.security.srp.jaas.SRPCacheLoginModule"
            flag = "required">
            <module-option name = "cacheJndiName">srp-test/AuthenticationCache</
module-option>
        </login-module>
        <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
            flag = "required">
            <module-option name = "password-stacking">useFirstPass</module-option>
        </login-module>
    </authentication>
</application-policy>
```

The example services are the ServiceConfig and the PropertiesVerifierStore and SRPSERVICE MBeans. Note that the JndiName attribute of the PropertiesVerifierStore is equal to the Verifier-SourceJndiName attribute of the SRPSERVICE, and that the SRPSERVICE depends on the PropertiesVerifierStore. This is required because the SRPSERVICE needs an implementation of the SRPVerifierStore interface for accessing user password verification information.

The client side login module configuration of Listing 8-17 makes use of the SRPLLoginModule with a srpServerJndiName option value that corresponds to the JBoss server component SRPSERVICE Jndi-Name attribute value("srp-test/SRPServerInterface"). Also needed is the ClientLoginModule configured with the password-stacking="useFirstPass" value to propagate the user authentication credentials generated by the SRPLLoginModule to the EJB invocation layer.

There are two issues to note about the server side login module configuration. First, note the cacheJndiName=srp-test/AuthenticationCache configuration option tells the SRPCacheLoginModule the location of the CachePolicy that contains the SRPServerSession for users who have authenticated against the SRPSERVICE. This value corresponds to the SRPSERVICE AuthenticationCacheJndiName attribute value. Second, the configuration includes a UsersRolesLoginModule with the password-stacking=useFirstPass configuration option. It is required to use a second login module with the SRP-

CacheLoginModule because SRP is only an authentication technology. A second login module needs to be configured that accepts the authentication credentials validated by the SRPCacheLoginModule to set the principal's roles that determines the principal's permissions. The UsersRolesLoginModule is augmenting the SRP authentication with properties file based authorization. The user's roles are coming from the roles.properties file included in the EJB jar.

Now, run the example 3 client by executing the following command from the book examples directory:

```
examples 982>ant -Dchap=8 -Dex=3 run-example
Buildfile: build.xml
...
run-example3:
    [copy] Copying 1 file to G:\JBossReleases\jboss-
3.0.1RC1\server\default\deploy
    [echo] Waiting for 5 seconds for deploy...
    [java] Logging in using the 'srp' configuration
    [java] Created Echo
    [java] Echo.echo()#1 = This is call 1
    [java] Echo.echo()#2 = This is call 2

BUILD SUCCESSFUL

Total time: 31 seconds
```

In the examples directory you will find a file called ex3-trace.log. This is a detailed trace of the client side of the SRP algorithm. The traces show step-by-step the construction of the public keys, challenges, session key and verification.

Note that the client has taken a long time to run relative to the other simple examples. The reason for this is the construction of the client's public key. This involves the creation of a cryptographically strong random number, and this process takes quite a bit of time the first time it occurs. If you were to log out and log in again within the same VM, the process would be much faster. Also note that "Echo.echo()#2" fails with an Authentication exception. The client code sleeps for 15 seconds after making the first call to demonstrate the behavior of the SRPService cache expiration. The SRPService cache policy timeout has been set to a mere 10 seconds to force this issue. As stated earlier, you need to make the cache timeout very long, or handle re-authentication on failure.

Running JBoss with a Java 2 security manager

By default the JBoss server does not start with a Java 2 security manager. If you want to restrict privileges of code using Java 2 permissions you need to configure the JBoss server to run under a security manager. This is done by configuring the Java VM options in the run.bat or run.sh scripts in the JBoss server distribution bin directory. The two required VM options are as follows:

- **java.security.manager**: This is used without any value to specify that the default security manager should be used. This is the preferred security manager. You can also pass a value to the java.security.manager option to specify a custom security manager implementation. The value must be the fully qualified class name of a subclass of java.lang.SecurityManager. This form specifies that the policy file should augment the default security policy as configured by the VM installation.

java.security.policy: This is used to specify the policy file that will augment the default security policy information for the VM. This option takes two forms:

java.security.policy=policyFileURL java.security.policy==policyFileURL

The first form specifies that the policy file should augment the default security policy as configured by the VM installation. The second form specifies that only the indicated policy file should be used. The policyFileURL value can be any URL for which a protocol handler exists, or a file path specification.

Listing 8-18 illustrates a fragment of the standard run.bat start script for Win32 that shows the addition of these two options to the command line used to start JBoss.

LISTING 8-18. The modifications to the Win32 run.bat start script to run JBoss with a Java 2 security manager.

```
...  
  
set CONFIG=%1  
@if "%CONFIG%" == "" set CONFIG=default  
set PF=..%conf%/%CONFIG%/server.policy  
set OPTS=-Djava.security.manager  
set OPTS=%OPTS% -Djava.security.policy=%PF%  
echo JBOSS_CLASSPATH=%JBOSS_CLASSPATH%  
java %JAXP% %OPTS% -classpath "%JBOSS_CLASSPATH%" org.jboss.Main %*
```

Listing 8-19 shows a fragment of the standard run.sh start script for UNIX/Linux systems that shows the addition of these two options to the command line used to start JBoss.

LISTING 8-19. The modifications to the UNIX/Linux run.sh start script to run JBoss with a Java 2 security manager.

```
...  
  
CONFIG=$1  
if [ "$CONFIG" == "" ]; then CONFIG=default; fi  
PF=..%conf%/$CONFIG/server.policy  
OPTS=-Djava.security.manager  
OPTS="$OPTS -Djava.security.policy=$PF"  
echo JBOSS_CLASSPATH=$JBOSS_CLASSPATH  
java $HOTSPOT $JAXP $OPTS -classpath $JBOSS_CLASSPATH org.jboss.Main $@
```

Both start scripts are setting the security policy file to the server.policy file located in the JBoss configuration file set directory that corresponds to the configuration name passed as the first argument to

the script. This allows one maintain a security policy per configuration file set without having to modify the start script.

Enabling Java 2 security is the easy part. The difficult part of Java 2 security is establishing the allowed permissions. If you look at the server.policy file that is contained in the default configuration file set, you'll see that it contains the following permission grant statement:

```
grant {  
    // Allow everything for now  
    permission java.security.AllPermission;  
};
```

This effectively disables security permission checking for all code as it says any code can do anything, which is not a reasonable default. What is a reasonable set of permissions is entirely up to you. To conclude this discussion, here is a little-known tidbit on debugging security policy settings. There are various debugging flag that you can set to determine how the security manager is using your security policy file as well as what policy files are contributing permissions. Running the VM as follows shows the possible debugging flag settings:

```
bin 1205>java -Djava.security.debug=help  
  
all      turn on all debugging  
access   print all checkPermission results  
jar      jar verification  
policy   loading and granting  
scl      permissions SecureClassLoader assigns
```

The following can be used with access:

```
stack    include stack trace  
domain   dumps all domains in context  
failure  before throwing exception, dump stack  
         and domain that didn't have permission
```

Running with -Djava.security.debug=all provides the most output, but the output volume is torrential. This might be a good place to start if you don't understand a given security failure at all. A less verbose setting that helps debug permission failures is to use -Djava.security.debug=access,failure. This is still relatively verbose, but not nearly as bad as the all mode as the security domain information is only displayed on access failures.

Using SSL with JBoss using JSSE

JBoss uses JSSE the Java Secure Socket Extension (JSSE) . JSSE is bundled with JBoss and it comes with JDK 1.4. For more information on JSSE see: <http://java.sun.com/products/jsse/index.html>. A simple test that you can use the JSSE as bundled with JBoss works is to run a program like the following:

```
import java.net.*;
import javax.net.ServerSocketFactory;
import javax.net.ssl.*;

public class JSSE_install_check
{
    public static void main(String[] args) throws Exception
    {
        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
        ServerSocketFactory factory =
            SSLSocketFactory.getDefault();
        SSLSocket sslSocket = (SSLSocket)
            factory.createServerSocket(12345);

        String [] cipherSuites = sslSocket.getEnabledCipherSuites();
        for(int i = 0; i < cipherSuites.length; i++)
        {
            System.out.println("Cipher Suite " + i +
                " = " + cipherSuites[i]);
        }
    }
}
```

The book examples includes a testcase for this which can be run using the following command. This will produce a lot of output as the -Djavax.net.debug=all option is passed to the VM.

```
examples 1052>java -version
java version "1.3.1_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_03-b03)
Java HotSpot(TM) Client VM (build 1.3.1_03-b03, mixed mode)
examples 1053>ant -Dchap=8 -Dex=4a run-example
Buildfile: build.xml
...
run-example4a:
[echo] Testing JSSE availability
[java] keyStore is :
[java] keyStore type is : jks
[java] init keystore
[java] init keymanager of type SunX509
[java] trustStore is:
D:\usr\local\Java\jdk1.3.1_03\jre\lib\security\cacerts
[java] trustStore type is : jks
[java] init truststore
...
[java] trigger seeding of SecureRandom
[java] done seeding SecureRandom
[java] Cipher Suite 0 = SSL_DHE_DSS_WITH_DES_CBC_SHA
[java] Cipher Suite 1 = SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
[java] Cipher Suite 2 = SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
[java] Cipher Suite 3 = SSL_RSA_WITH_RC4_128_MD5
[java] Cipher Suite 4 = SSL_RSA_WITH_RC4_128_SHA
[java] Cipher Suite 5 = SSL_RSA_WITH DES_CBC_SHA
[java] Cipher Suite 6 = SSL_RSA_WITH_3DES_EDE_CBC_SHA
[java] Cipher Suite 7 = SSL_RSA_EXPORT_WITH_RC4_40_MD5
```

The JSSE jars include the jcrt.jar, jnet.jar and jsse.jar in the JBOSS_DIST/client directory.

Once you have tested that JSSE runs, you need a public key/private key pair in the form of an X509 certificate for use by the SSL server sockets. For the purpose of this example we have created a self-signed certificate using the JDK 1.3 keytool and included the resulting keystore file in the chap8 source directory as chap8.keystore. It was created using the following command and input:

```
examples 1121>keytool -genkey -alias rmi+ssl -keyalg RSA  
-keystore chap8.keystore -validity 3650  
Enter keystore password: rmi+ssl  
What is your first and last name?  
[Unknown]: Chapter8 SSL Example  
What is the name of your organizational unit?  
[Unknown]: JBoss Book  
What is the name of your organization?  
[Unknown]: JBoss Group, LLC  
What is the name of your City or Locality?  
[Unknown]: Issaquah  
What is the name of your State or Province?  
[Unknown]: WA  
What is the two-letter country code for this unit?  
[Unknown]: US  
Is <CN=Chapter8 SSL Example, OU=JBoss Book, O="JBoss Group, LLC", L=Issaquah, ST  
[no]: yes=WA, C=US> correct?  
  
Enter key password for <rmi+ssl>  
(RETURN if same as keystore password):
```

This produces a keystore file called chap8.keystore. A keystore is a database of security keys. There are two different types of entries in a keystore:

- key entries: each entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate "chain" for the corresponding public key. The keytool and jarsigner tools only handle the later type of entry, that is private keys and their associated certificate chains.
- trusted certificate entries: each entry contains a single public key certificate belonging to another party. It is called a "trusted certificate" because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the "subject" (owner) of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

Listing the src/main/org/jboss/chap8/chap8.keystore examples file contents using the keytool shows one self-signed certificate:

```
bin 1054>keytool -list -v -keystore src/main/org/jboss/chap8/chap8.keystore  
Enter keystore password: rmi+ssl  
  
Keystore type: jks  
Keystore provider: SUN  
  
Your keystore contains 1 entry:  
  
Alias name: rmi+ssl
```

```
Creation date: Thu Nov 08 19:50:23 PST 2001
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Chapter8 SSL Example, OU=JBoss Book, O="JBoss Group, LLC",
L=Issaquah, ST=WA, C=US
Issuer: CN=Chapter8 SSL Example, OU=JBoss Book, O="JBoss Group, LLC",
L=Issaquah, ST=WA, C=US
Serial number: 3beb5271
Valid from: Thu Nov 08 19:50:09 PST 2001 until: Sun Nov 06
19:50:09 PST 2011
Certificate fingerprints:
MD5: F6:1B:2B:E9:A5:23:E7:22:B2:18:6F:3F:9F:E7:38:AE
SHA1: F2:20:50:36:97:86:52:89:71:48:A2:C3:06:C8:F9:2D:F7:79:00:36
*****
*****
```

With JSSE working and a keystore with the certificate you will use for the JBoss server, you are ready to configure JBoss to use SSL for EJB access. This is done by configuring the EJB invoker RMI socket factories. The JBossSX framework includes implementations of the `java.rmi.server.RMIServerSocketFactory` and `java.rmi.server.RMIClientSocketFactory` interfaces that enable the use of RMI over SSL encrypted sockets. The implementation classes are `org.jboss.security.ssl.RMISSLServerSocketFactory` and `org.jboss.security.ssl.RMISSLClientSocketFactory` respectively. There are two steps to enable the use of SSL for RMI access to EJBs. The first is to enable the use of a keystore as the database for the SSL server certificate, which is done by configuring an `org.jboss.security.plugins.JaasSecurityDomain` MBean. The `jboss-service.xml` descriptor in the `chap8/ex4` directory includes the `JaasSecurityDomain` definition shown in Listing 8-20.

LISTING 8-20. A sample JaasSecurityDomain config for RMI/SSL

```
<!-- The SSL domain setup -->
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="jboss.security:service=JaasSecurityDomain,domain=RMI+SSL">
  <constructor>
    <arg type="java.lang.String" value="RMI+SSL" />
  </constructor>
  <attribute name="KeyStoreURL">chap8.keystore</attribute>
  <attribute name="KeyStorePass">rmi+ssl</attribute>
</mbean>
```

The `JaasSecurityDomain` is a subclass of the standard `JaasSecurityManager` class that adds the notions of a keystore as well JSSE KeyManagerFactory and TrustManagerFactory access. It extends the basic security manager to allow support for SSL and other cryptographic operations that require security keys. This configuration simply loads the `chap8.keystore` from the example 4 MBean sar using the indicated password.

The second step is to define an EJB invoker configuration that uses the JBossSX RMI socket factories that support SSL. To do this you need to define a custom configuration for the `JRMPInvoker` we saw in Chapter “EJBs on JBoss - The EJB Container Configuration and Architecture” on page 141 , as

well as an EJB setup that makes use of this invoker. The configuration required to enable RMI over SSL access to stateless session bean is provided for you in Listing 8-21. The top of the listing shows the jboss-service.xml descriptor that defines the custom JRMPInvoker, and the bottom shows the example 4 “EchoBean4” configuration needed to use the SSL invoker. You will use this configuration in a stateless session bean example.

LISTING 8-21. The jboss-service.xml and jboss.xml configurations to enable SSL with the example 4 stateless session bean.

```
// The jboss-service.xml SSL JRMPInvoker MBean Configuration
<mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
       name="jboss:service=invoker,type=jrmp,socketType=SSL">
    <attribute name="RMIOBJECTPort">4445</attribute>
    <attribute name="RMIClientSocketFactory">
        org.jboss.security.ssl.RMISSLClientSocketFactory
    </attribute>
    <attribute name="RMIServerSocketFactory">
        org.jboss.security.ssl.RMISSLServerSocketFactory
    </attribute>
    <attribute name="SecurityDomain">java:/jaas/RMI+SSL</attribute>
    <depends>jboss.security:service=JaasSecurityDomain, domain=RMI+SSL</depends>
</mbean>

// The jboss.xml session bean configuration to use the SSL invoker
<?xml version="1.0"?>
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>EchoBean4</ejb-name>
            <configuration-name>Standard Stateless SessionBean</configuration-name>
            <home-invoker>jboss:service=invoker,type=jrmp,socketType=SSL
            </home-invoker>
            <bean-invoker>jboss:service=invoker,type=jrmp,socketType=SSL
            </bean-invoker>
        </session>
    </enterprise-beans>
</jboss>
```

The example 4 code is located under the src/main/org/jboss/chap8/ex4 directory of the book examples. This is another simple stateless session bean with an echo method that returns its input argument. It is hard to tell when SSL is in use unless it fails, so we'll run the example 4 client in two different ways to demonstrate that the EJB deployment is in fact using SSL. Start the JBoss server using the chap8 configuration and then run example 4b as follows:

```
examples 514>ant -Dchap=8 -Dex=4b run-example
Buildfile: build.xml
...
run-example4b:
    [copy] Copying 1 file to G:\JBossReleases\jboss-3.0.1RC1\server\default\deploy
    [echo] Waiting for 15 seconds for deploy...
    [java] java.rmi.MarshalException: Error marshaling transport header; nested
exception is:
```

```
[java] javax.net.ssl.SSLException: untrusted server cert chain
[java]     at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a([DashoPro-V1.2-120198])
[java]     at com.sun.net.ssl.internal.ssl.ClientHandshaker.a([DashoPro-V1.2-
120198])
[java]     at
com.sun.net.ssl.internal.ssl.ClientHandshaker.processMessage([DashoProV1.2-120198])
[java]     at com.sun.net.ssl.internal.ssl.Handshaker.process_record([DashoPro-V1.2-
120198])
[java]     at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a([DashoPro-V1.2-120198])
[java]     at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a([DashoPro-V1.2-120198])
[java]     at com.sun.net.ssl.internal.ssl.AppOutputStream.write([DashoPro-V1.2-
120198])
[java]     at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:67)
[java]     at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:125)
[java]     at java.io.DataOutputStream.flush(DataOutputStream.java:99)
[java]     at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:198)
[java]     at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:169)
[java]     at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:78)
[java]     at org.jboss.invocation.jrmp.server.JRMPInvoker_Stub.invoke(Unknown
Source)
[java]     at
org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxy.invoke(JRMPInvokerProxy.java:128)
[java]     at
org.jboss.invocation.InvokerInterceptor.invoke(InvokerInterceptor.java:108)
[java]     at
org.jboss.proxy.TransactionInterceptor.invoke(TransactionInterceptor.java:73)
[java]     at
org.jboss.proxy.SecurityInterceptor.invoke(SecurityInterceptor.java:76)
[java]     at org.jboss.proxy.ejb.HomeInterceptor.invoke(HomeInterceptor.java:198)
[java]     at org.jboss.proxy.ClientContainer.invoke(ClientContainer.java:76)
[java]     at $Proxy0.create(Unknown Source)
[java]     at org.jboss.chap8.ex4.ExClient.main(ExClient.java:31)
[java] Exception in thread "main"
[java] Java Result: 1
```

BUILD SUCCESSFUL

Total time: 26 seconds

The resulting exception is expected, and is the purpose of the 4b version of the example. Note that the exception stack trace has been edited to fit into the book format, so expect some difference. The key item to notice about the exception is it clearly shows you are using the Sun JSSE classes to communicate with the JBoss EJB container. The exception is saying that the self-signed certificate you are using as the JBoss server certificate cannot be validated as signed by any of the default certificate authorities. This is expected because the default certificate authority keystore that ships with the JSSE package only includes well known certificate authorities such as VeriSign, Thawte, and RSA Data Security. To get the EJB client to accept your self-signed certificate as valid, you need to tell the JSSE classes to use your chap8.keystore as its truststore. A truststore is just a keystore that contains public key certificates used to sign other certificates. To do this, run example 4 using -Dex=4 rather than -Dex=4b to pass the location of the correct truststore using the javax.net.ssl.trustStore system property:

```
examples 516>ant -Dchap=8 -Dex=4 run-example
Buildfile: build.xml
...
run-example4:
```

```
[copy] Copying 1 file to G:\JBossReleases\jboss-  
3.0.1RC1\server\default\deploy  
[echo] Waiting for 5 seconds for deploy...  
[java] [DEBUG,RMISSLClientSocketFactory] SSL handshakeCompleted, cipher=SSL  
_RSA_WITH_RC4_128_SHA, peerHost=172.17.66.54  
[java] 0 [Thread-0] DEBUG org.jboss.security.ssl.RMISSLClientSocketFactory  
- SSL handshakeCompleted, cipher=SSL_RSA_WITH_RC4_128_SHA,  
peerHost=172.17.66.54  
[java] Created Echo  
[java] Echo.echo()#1 = This is call 1  
  
BUILD SUCCESSFUL  
  
Total time: 17 seconds
```

This time the only indication that an SSL socket is involved is because of the "SSL handshakeCompleted" message. This is coming from the RMISSLClientSocketFactory class as a debug level log message. If you did not have the client configured to print out log4j debug level messages, there would be no direct indication that SSL was involved. If you note the run times and the load on your system CPU, there definitely is a difference. SSL, like SRP, involves the use of cryptographically strong random numbers that take time to seed the first time they are used. This shows up as high CPU utilization and start up times.

One consequence of this is that if you are running on a system that is slower than the one used to run the examples for the book, such as when running example 4b, you may see an exception similar to the following:

```
javax.naming.NameNotFoundException: EchoBean not bound  
at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer  
at sun.rmi.transport.StreamRemoteCall.executeCall  
at sun.rmi.server.UnicastRef.invoke  
at org.jnp.server.NamingServer_Stub.lookup  
at org.jnp.interfaces.NamingContext.lookup  
at org.jnp.interfaces.NamingContext.lookup  
at javax.naming.InitialContext.lookup  
at org.jboss.chap8.ex3.ExClient.main(ExClient.java:23)  
Exception in thread "main"  
Java Result: 1
```

The problem is that the JBoss server has not finished deploying the example EJB in the time the client allowed. This is due to the initial setup time of the secure random number generator used by the SSL server socket. If you see this issue, simply rerun the example again or increase the deployment wait time in the chap8 build.xml Ant script.

This chapter describes the steps for integrating a third party Web container into the JBoss application server framework. A Web container is a J2EE server component that enables access to servlets and JSP pages. Example servlet containers include Tomcat and Jetty.

Integrating a servlet container into JBoss consists of mapping web-app.xml JNDI information into the JBoss JNDI namespace using an optional jboss-web.xml descriptor as well as delegating authentication and authorization to the JBoss security layer. The [org.jboss.web.AbstractWebContainer](#) class exists to simplify these tasks. The focus of the first part of this chapter is how to integrate a Web container using the [AbstractWebContainer](#) class. The chapter concludes with a discussion on how to configure the use of secure socket layer (SSL) encryption with the JBoss/Tomcat bundle, as well as how to configure Apache with the JBoss/Tomcat bundle.

The AbstractWebContainer Class

The [org.jboss.web.AbstractWebContainer](#) class is an implementation of a template pattern for web container integration into JBoss. Web container providers wishing to integrate their container into a JBoss server should create a subclass of [AbstractWebContainer](#) and provide the web container specific setup and war deployment steps. The [AbstractWebContainer](#) provides support for parsing the standard J2EE web.xml web application deployment descriptor JNDI and security elements as well as support for parsing the JBoss specific jboss-web.xml descriptor. Parsing of these deployment descriptors is performed to generate an integrated JNDI environment and security context. We have already seen the most of the elements of the jboss-web.xml descriptor in other chapters. Figure 9-1 provides a complete view of the jboss-web.xml descriptor DTD for reference. The complete DTD with comments can be found in xxx.

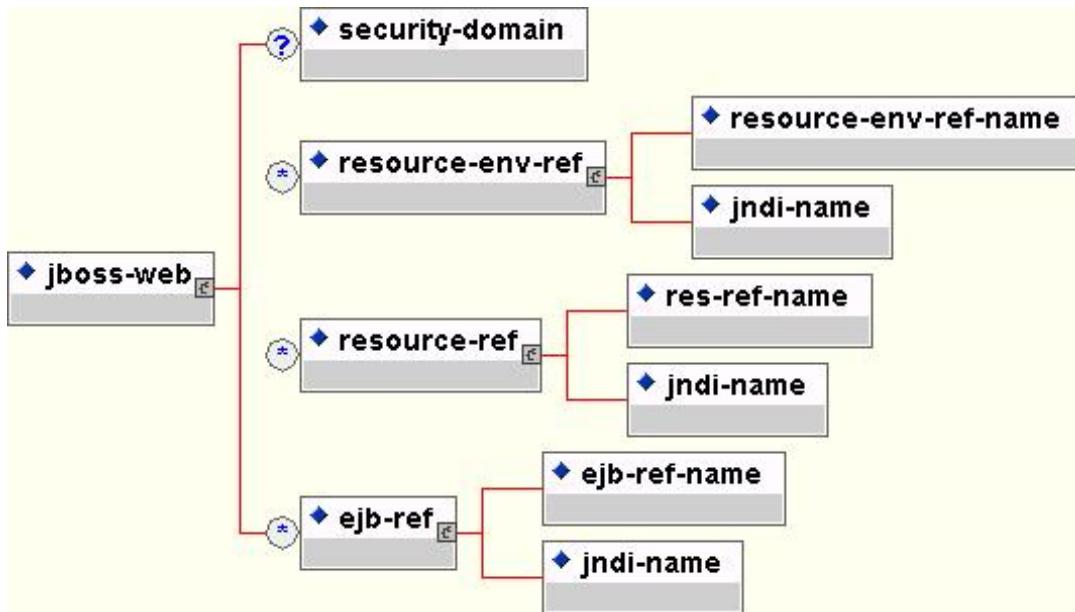


FIGURE 9-1. The complete jboss-web.xml descriptor DTD.

The two elements that have not been discussed are the context-root and virtual-host. The context-root element allows one to specify the prefix under which web application is located. This is only applicable to stand-alone web application deployment as a WAR file. Web applications included as part of an EAR must set the root using the context-root element of the EAR application.xml descriptor. The virtual-host element specifies the DNS name of the virtual host to which the web application should be deployed. The details of setting up virtual hosts for servlet contexts depends on the particular servlet container. We will look at examples of using the virtual-host element when we look at the Tomcat and Jetty servlet containers later in this chapter.

The AbstractWebContainer Contract

The AbstractWebContainer is an abstract class that implements the org.jboss.web.AbstractWebContainerMBean interface used by the JBoss J2EE deployer to delegate the task of installing war files needing to be deployed. Listing 9-1 presents some of the key AbstractWebContainer methods.

LISTING 9-1. Key methods of the AbstractWebContainer class.

```

160:public abstract class AbstractWebContainer
161:  extends SubDeployerSupport
162:  implements AbstractWebContainerMBean
163: {
```

```
164:     public static interface WebDescriptorParser
165:     {
166:         public void parseWebAppDescriptors(ClassLoader loader, WebMetaData
metaData) throws Exception;
167:     }
168:     public boolean accepts(DeploymentInfo sdi)
169:     {
170:         String warFile = sdi.url.getFile();
171:         return warFile.endsWith("war") || warFile.endsWith("war/");
172:     }
173:     public synchronized void start(DeploymentInfo di) throws
DeploymentException
174:     {
175:         Thread thread = Thread.currentThread();
176:         ClassLoader appClassLoader = thread.getContextClassLoader();
177:         try
178:         {
179:             // Create a classloader for the war to ensure a unique ENC
180:             URL[] empty = {};
181:             URLClassLoader warLoader = URLClassLoader.newInstance(empty, di.ucl);
182:             thread.setContextClassLoader(warLoader);
183:             WebDescriptorParser webAppParser = new DescriptorParser(di);
184:             String webContext = di.webContext;
185:             if( webContext != null )
186:             {
187:                 if( webContext.length() > 0 && webContext.charAt(0) != '/' )
188:                     webContext = "/" + webContext;
189:             }
190:             // Get the war URL
191:             URL warURL = di.localUrl != null ? di.localUrl : di.url;
192:             if (log.isDebugEnabled())
193:             {
194:                 log.debug("webContext: " + webContext);
195:                 log.debug("warURL: " + warURL);
196:                 log.debug("webAppParser: " + webAppParser);
197:             }
198:             // Parse the web.xml and jboss-web.xml descriptors
199:             WebMetaData metaData = parseMetaData(webContext, warURL);
200:             WebApplication warInfo = new WebApplication(metaData);
201:             performDeploy(warInfo, warURL.toString(), webAppParser);
202:             deploymentMap.put(warURL.toString(), warInfo);
203:         }
204:         finally
205:         {
206:             thread.setContextClassLoader(appClassLoader);
207:         }
208:     }
209:     protected abstract void performDeploy(WebApplication webApp, String
warUrl,
210:                                         WebDescriptorParser webAppParser) throws Exception;
211:     public synchronized void stop(DeploymentInfo di)
212:     throws DeploymentException
213:     {
214:         String warUrl = di.localUrl.toString();
215:         try
```

```

343:      {
344:          performUndeploy(warUrl);
345:          // Remove the web application ENC...
346:          deploymentMap.remove(warUrl);
347:      }
352:      catch(Exception e)
353:      {
354:          throw new DeploymentException("Error during deploy", e);
355:      }
356:  }
365:  protected abstract void performUndeploy(String warUrl) throws Exception;
366:
405:  public void setConfig(Element config)
406:  {
407:  }
416:  protected void parseWebAppDescriptors(DeploymentInfo di, ClassLoader
loader,
418:      throws Exception417:          WebMetaData metaData)
419:  {
421:      InitialContext iniCtx = new InitialContext();
422:      Context envCtx = null;
423:      ClassLoader currentLoader =
Thread.currentThread().getContextClassLoader();
424:      try
425:      {
426:          // Create a java:comp/env environment unique for the web application
434:          Thread.currentThread().setContextClassLoader(loader);
435:          envCtx = (Context) iniCtx.lookup("java:comp");
437:          // Add a link to the global transaction manager
438:          envCtx.bind("UserTransaction", new LinkRef("UserTransaction"));
440:          envCtx = envCtx.createSubcontext("env");
441:      }
442:      finally
443:      {
444:          Thread.currentThread().setContextClassLoader(currentLoader);
445:      }
446:
447:      Iterator envEntries = metaData.getEnvironmentEntries();
449:      addEnvEntries(envEntries, envCtx);
450:      Iterator resourceEnvRefs = metaData.getResourceEnvReferences();
452:      linkResourceEnvRefs(resourceEnvRefs, envCtx);
453:      Iterator resourceRefs = metaData.getResourceReferences();
455:      linkResourceRefs(resourceRefs, envCtx);
456:      Iterator ejbRefs = metaData.getEjbReferences();
458:      linkEjbRefs(ejbRefs, envCtx, di);
459:      Iterator ejbLocalRefs = metaData.getEjbLocalReferences();
461:      linkEjbLocalRefs(ejbLocalRefs, envCtx, di);
462:      String securityDomain = metaData.getSecurityDomain();
464:      linkSecurityDomain(securityDomain, envCtx);
466:  }
468:  protected void addEnvEntries(Iterator envEntries, Context envCtx)
469:      throws ClassNotFoundException, NamingException
470:  {
479:  }
480:

```

```
481:     protected void linkResourceEnvRefs(Iterator resourceEnvRefs, Context
envCtx)
482:         throws NamingException
483:     {
484:     }
510: }
512:     protected void linkResourceRefs(Iterator resourceRefs, Context envCtx)
513:         throws NamingException
514:     {
515:     }
539: }
619:     protected void linkEjbRefs(Iterator ejbRefs, Context envCtx,
DeploymentInfo di)
620:         throws NamingException
621:     {
638:     }
640:     protected void linkEjbLocalRefs(Iterator ejbRefs, Context envCtx,
DeploymentInfo di)
641:         throws NamingException
642:     {
655:     }
666:     protected void linkSecurityDomain(String securityDomain, Context envCtx)
667:         throws NamingException
668:     {
686:     }
732: public String[] getCompileClasspath(ClassLoader loader)
733: {
764: }
947: }
```

Lines 197-201 correspond to the accepts method implemented by JBoss deployers to indicate which type of deployments they accepts. The AbstractWebContainer handles the deployments of WARs as jars or unpacked directories.

Lines 15-38 correspond to the start method. This method is a template pattern method implementation. The argument to the deploy method is the WAR deployment info object. This contains the URL to the WAR, the UnifiedClassLoader for the WAR, the parent archive such as an EAR, and the J2EE application.xml context-root if the WAR is part of an EAR.

The first step of the start method is to save the current thread context ClassLoader and then create another URLClassLoader (warLoader) using the WAR UnifiedClassLoader as its parent. This warLoader is used to ensure a unique JNDI ENC (enterprise naming context) for the WAR will be created. This is done by the code on lines 277-278. Chapter 3 mentioned that the java:comp context's uniqueness was determined by the ClassLoader that created the java:comp context. The warLoader ClassLoader is set as the current thread context ClassLoader, on line 279, before the performDeploy call is made. Next, the web.xml and jboss-web.xml descriptors are parsed by calling parseMetaData on line 298. Next, the Web container-specific subclass is asked to perform the actual deployment of the WAR through the performDeploy call on line 300. The WebApplication object for this deployment is stored in the deployed application map using the warUrl as the key on line 301. The final step at line 313 is to restore the thread context ClassLoader to the one that existed at the start of the method.

Lines 330-331 give the signature for the abstract `performDeploy` method. This method is called by the `start` method and must be overridden by subclasses to perform the Web container specific deployment steps. A `WebApplication` is provided as an argument, and this contains the metadata from the `web.xml` descriptor, and the `jboss-web.xml` descriptor. The metadata contains the `context-root` value for the web module from the J2EE application.xml descriptor, or if this is a stand-alone deployment, the `jboss-web.xml` descriptor. The metadata also contains any `jboss-web.xml` descriptor `virtual-host` value. On return from `performDeploy`, the `WebApplication` must be populated with the `ClassLoader` of the servlet context for the deployment. The `warUrl` argument is the string for the URL of the Web application WAR to deploy. The `webAppParser` argument is a callback handle the subclass must use to invoke the `parseWebAppDescriptors` method to set up the Web application JNDI environment. This callback provides a hook for the subclass to establish the Web application JNDI environment before any servlets are created that are to be loaded on startup of the WAR. A subclass' `performDeploy` method implementation needs to be arranged so that it can call the `parseWebAppDescriptors` before starting any servlets that need to access JNDI for JBoss resources like EJBs, resource factories, and so on. One important setup detail that needs to be handled by a subclass implementation is to use the current thread context `ClassLoader` as the parent `ClassLoader` for any Web container-specific `ClassLoader` created. Failure to do this results in problems for Web applications that attempt to access EJBs or JBoss resources through the JNDI ENC.

Lines 338-356 correspond to the `stop` method. This is a template pattern method implementation. Line 344 of this method calls the subclass `performUndeploy` method to perform the container-specific undeployment steps. Next, at line 346, the `warUrl` is unregistered from the deployment map. The `warUrl` argument is the string URL of the WAR as originally passed to the `performDeploy` method.

Line 365 gives the signature of the abstract `performUndeploy` method. This method is called as part of the `stop` method template as shown on line 344. A call to `performUndeploy` asks the subclass to perform the Web container-specific undeployment steps.

Lines 405-407 correspond to the `setConfig` method. This method is a stub method that subclasses can override if they want to support an arbitrary extended configuration beyond that which is possible through MBean attributes. The `config` argument is the parent DOM element for an arbitrary hierarchy given by the child element of the `Config` attribute in the `mbean` element specification of the `jboss-service.xml` descriptor of the web container service. You'll see an example use of this method and config value when you look at the MBean that supports embedding Tomcat into JBoss.

Lines 416- 466 correspond to the `parseWebAppDescriptors` method. This is invoked from within the subclass `performDeploy` method when it invokes the `webAppParser.parseWebAppDescriptors` callback to setup Web application ENC (`java:comp/env`) `env-entry`, `resource-env-ref`, `resource-ref`, `local-ejb-ref` and `ejb-ref` element values declared in the `web.xml` descriptor. The creation of the `env-entry` values does not require a `jboss-web.xml` descriptor. The creation of the `resource-env-ref`, `resource-ref`, and `ejb-ref` elements does require a `jboss-web.xml` descriptor for the JNDI name of the deployed resources/EJBs. Because the ENC context is private to the Web application, the Web application `ClassLoader` is used to identify the ENC. The `loader` argument is the `ClassLoader` for the Web application, and may not be null. The `metaData` argument is the `WebMetaData` argument passed to the subclass `performDeploy` method. The implementation of the `parseWebAppDescriptors` uses the metadata information from the WAR deployment descriptors and then creates the JNDI ENC bindings by calling methods shown on lines 447-464.

The addEnvEntries method on lines 468-479 creates the java:comp/env Web application env-entry bindings that were specified in the web.xml descriptor.

The linkResourceEnvRefs method on lines 481-510 maps the java:comp/env/xxx Web application JNDI ENC resource-env-ref web.xml descriptor elements onto the deployed JNDI names using the mappings specified in the jboss-web.xml descriptor.

The linkResourceRefs method on lines 512-539 maps the java:comp/env/xxx Web application JNDI ENC resource-ref web.xml descriptor elements onto the deployed JNDI names using the mappings specified in the jboss-web.xml descriptor.

The linkEjbRefs method on lines 619-638 maps the java:comp/env/ejb Web application JNDI ENC ejb-ref web.xml descriptor elements onto the deployed JNDI names using the mappings specified in the jboss-web.xml descriptor.

The linkEjbLocalRefs method on lines 640-655 maps the java:comp/env/ejb Web application JNDI ENC ejb-local-ref web.xml descriptor elements onto the deployed JNDI names using the ejb-link mappings specified in the web.xml descriptor.

The linkSecurityDomain method on lines 666-686 creates a java:comp/env/security context that contains a securityMgr binding pointing to the AuthenticationManager implementation and a realmMapping binding pointing to the RealmMapping implementation that is associated with the security domain for the Web application. Also created is a subject binding that provides dynamic access to the authenticated Subject associated with the request thread. If the jboss-web.xml descriptor contained a security-domain element, the bindings are javax.naming.LinkRefs to the JNDI name specified by the security-domain element, or subcontexts of this name. If there was no security-domain element, the bindings are to org.jboss.security.plugins.NullSecurityManager instance that simply allows all authentication and authorization checks.

Lines 732-764 correspond to the getCompileClasspath method. This is a utility method available for Web containers to generate a classpath that walks up the ClassLoader chain starting at the given loader and queries each ClassLoader for the URLs it serves to build a complete classpath of URL strings. This is needed by some JSP compiler implementations (Jasper for one) that expect to be given a complete classpath for compilation.

Creating an AbstractWebContainer Subclass

To integrate a web container into JBoss you need to create a subclass of AbstractWebContainer and implement the required performDeploy(WebApplication, String, WebDescriptorParser) and performUndeploy(String) methods as described in the preceding section. The following additional integration points should be considered as well.

Use the Thread Context Class Loader

Although this issue was noted in the performDeploy method description, we'll repeat it here since it is such a critical detail. During the setup of a WAR container, the current thread context ClassLoader

must be used as the parent ClassLoader for any web container specific ClassLoader that is created. Failure to do this will result in problems for web applications that attempt to access EJBs or JBoss resources through the JNDI ENC.

Integrate Logging Using log4j

JBoss uses the Apache log4j logging API as its internal logging API. For a web container to integrate well with JBoss it needs to provide a mapping between the web container logging abstraction to the log4j API. As a subclass of AbstractWebContainer, your integration class has access to the log4j interface via the super.log instance variable or equivalently, the superclass getLog method. This is an instance of the org.jboss.logging.Logger class that wraps the log4j category. The name of the log4j category is the name of the container subclass.

Delegate web container authentication and authorization to JBossSX

Ideally both web application and EJB authentication and authorization are handled by the same security manager. To enable this for your web container you must hook into the JBoss security layer. This typically requires a request interceptor that maps from the web container security callouts to the JBoss security API. Integration with the JBossSX security framework is based on the establishment of a “java:comp/env/security” context as described in the linkSecurityDomain method comments in the previous section. The security context provides access to the JBossSX security manager interface implementations associated with the web application for use by subclass request interceptors. An outline of the steps for authenticating a user using the security context is presented in Listing 9-2 in quasi pseudo-code. Listing 9-3 provides the equivalent process for the authorization of a user.

LISTING 9-2. A pseudo-code description of authenticating a user via the JBossSX API and the java:comp/env/security JNDI context.

```
// Get the username and password from the request context...
HttpServletRequest request = ...;
String username = getUsername(request);
String password = getPassword(request);
// Get the JBoss security manager from the ENC context
InitialContext iniCtx = new InitialContext();
AuthenticationManager securityMgr = (AuthenticationManager)
    iniCtx.lookup("java:comp/env/security/securityMgr");
SimplePrincipal principal = new SimplePrincipal(username);
if( securityMgr.isValid(principal, password) )
{
    // Indicate the user is allowed access to the web content...
    // Propagate the user info to JBoss for any calls into made by the servlet
    SecurityAssociation.setPrincipal(principal);
    SecurityAssociation.setCredential(password.toCharArray());
}
else
{
    // Deny access...
}
```

LISTING 9-3. A pseudo-code description of authorization a user via the JBossSX API and the java:comp/env/security JNDI context.

```
// Get the username & required roles from the request context...
HttpServletRequest request = ...;
String username = getUsername(request);
String[] roles = getContentTypeRoles(request);
// Get the JBoss security manager from the ENC context
InitialContext iniCtx = new InitialContext();
RealmMapping securityMgr = (RealmMapping)
    iniCtx.lookup("java:comp/env/security/realmMapping");
SimplePrincipal principal = new SimplePrincipal(username);
Set requiredRoles = new HashSet(java.util.Arrays.asList(roles));
if( securityMgr.doesUserHaveRole(principal, requiredRoles) )
{
    // Indicate user has the required roles for the web content...
}
else
{
    // Deny access...
}
```

JBoss/Tomcat-4.x bundle notes

In this section we'll discuss configuration issues specific to the JBoss/Tomcat-4.x integration bundle. The Tomcat-4.x release, which is also known by the name Catalina, is the latest Apache Java servlet container. It supports the Servlet 2.3 and JSP 1.2 specifications. The JBoss/Tomcat integration layer is controlled by the JBoss MBean service configuration. The MBean used to embed the Tomcat-4.x series of web containers is the [org.jboss.web.catalina.EmbeddedCatalinaServiceSX](#) service, and it is a subclass of the [AbstractWebContainer](#) class. Its configurable attributes include:

- **CatalinaHome**, sets the value to use for the catalina.home System property. This is used to point to a catalina distribution outside of the jboss structure. If not specified this will be determined based on the location of the jar containing the org.apache.catalina.startup.Embedded class assuming a standard catalina distribution structure.
- **CatalinaBase**, sets the value to use for the catalina.base System property. This is used to resolve relative paths. If not specified the CatalinaHome attribute value will be used.
- **Java2ClassLoaderCompliance**, enables the standard Java2 parent delegation class loading model rather than the servlet 2.3 load from war first model. This is true by default as loading from wars that include client jars with classes used by EJBs causes class loading conflicts. If you enable the servlet 2.3 class loading model by setting this flag to false, you will need to organize your deployment package to avoid duplicate classes in the deployment.
- **DeleteWorkDirs**: set the delete work directories on undeployment flag. By default catalina does not delete its working directories when a context is stopped and this can cause jsp pages in

redeployments to not be recompiled if the timestamp of the file in the war has not been updated. This defaults to true.

- **SnapshotMode**: Set the snapshot mode in a clustered environment. This must be one of “instant” or “interval”. In instant mode changes to a clustered session are instantly propagated whenever a modification is made. In interval mode all modifications are collected over the SnapshotInterval attribute value and periodically propagated.
- **SnapshotInterval**: Set the snapshot interval in ms for the “interval” snapshot mode. The default is 1000 or 1 second.
- **Config**, an attribute that provides support for extended configuration using constructs from the standard Tomcat server.xml file to specify additional connectors, and so on. Note that this is the only mechanism for configuring the embedded Tomcat servlet container as none of the Tomcat configuration files such as the conf/server.xml file are used. An outline of the configuration DTD that is currently supported is given in Figure 9-2, and the elements are described in the following section.

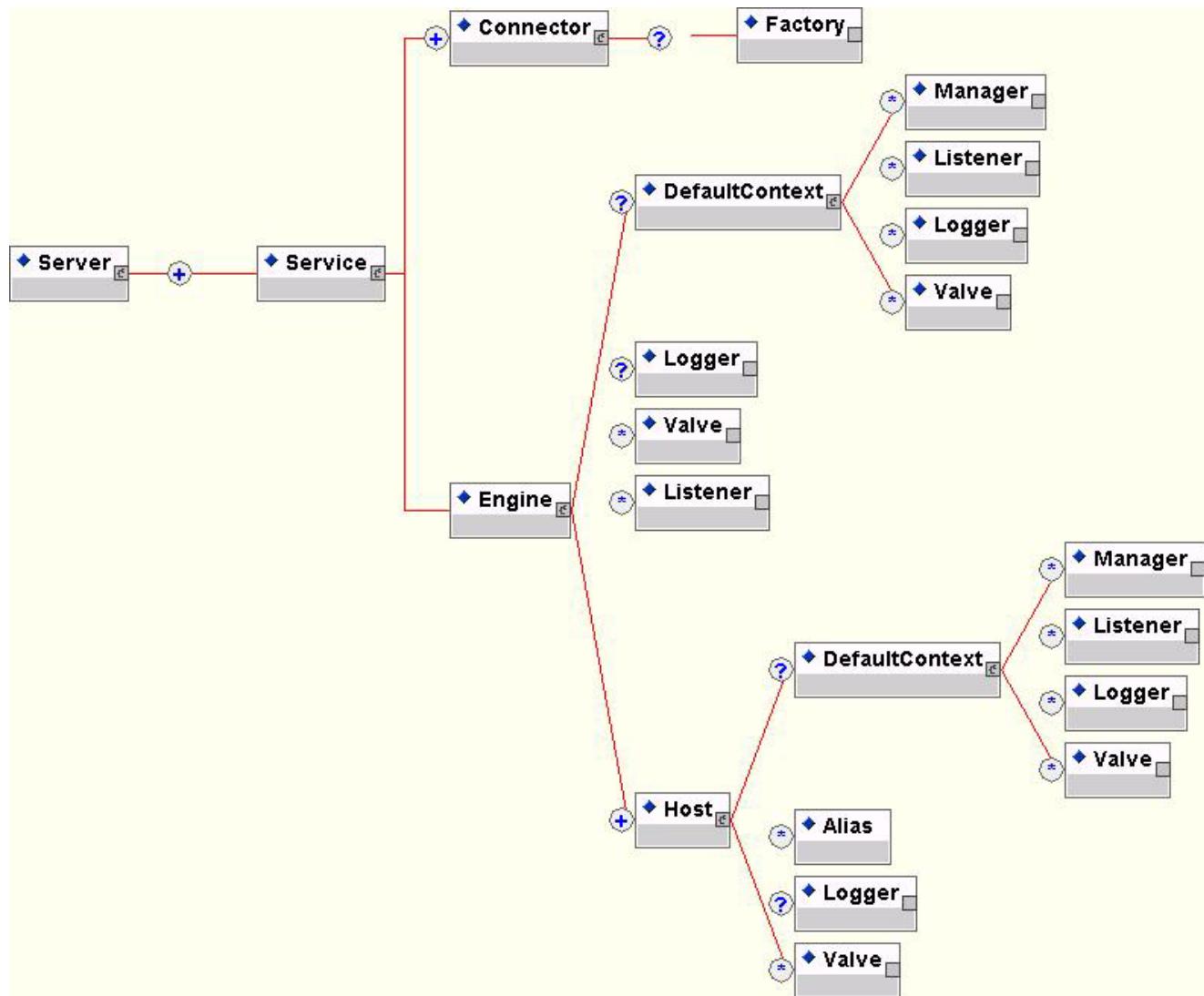


FIGURE 9-2. An overview of the Tomcat-4.0.4 configuration DTD supported by the EmbeddedCatalinaServiceSX Config attribute.

The integration of Tomcat with JBoss depends on the tomcat4-service.xml descriptor found in the deploy directories of the default and all configuration file sets. You will only find this descriptor in the bundled release of JBoss/Tomcat. This bundle also include the full Tomcat distribution as the JBOSS_DIST/catalina directory. Currently this is the jakarta-tomcat-4.0.4-LE-jdk14 distribution.

The Embedded Tomcat Configuration Elements

This section provides an overview of the Tomcat configuration elements that may appear as child elements of the EmbeddedCatalinaSX Config attribute.

Server

The Server element is the root element of the Tomcat servlet container configuration. There are no attributes of this element that are supported by the embedded service.

Service

A Service is a container of one or more Connectors and a single Engine. The only supported attribute is:

- **name**, a unique name by which the service is known.

Connector

A Connector element configures a transport mechanism that allows clients to send requests and receive responses from the Service it is associated with. Connectors forward requests to the Service Engine and return the results to the requesting client. There are currently three connector implementations, HTTP, AJP and Warp. All connectors support these attributes:

- **className**, the fully qualified name of the class of the connector implementation. The class must implement the org.apache.catalina.Connector interface. The embedded service defaults to the org.apache.catalina.connector.http.HttpConnector, which is the HTTP connector implementation.
- **enableLookups**, a flag that enables DNS resolution of the client hostname as accessed via the ServletRequest.getRemoteHost method. This flag defaults to false.
- **redirectPort**, the port to which non-SSL requests will be redirected when a request for content secured under a transport confidentiality or integrity constraint is received. This defaults to the standard https port of 443.
- **secure**, sets the ServletRequest.isSecure method value flag to indicate whether or not the transport channel is secure. This flag defaults to false.**scheme**, sets the protocol name as accessed by the ServletRequest.getScheme method. The scheme defaults to http.

THE HTTP CONNECTOR

The HTTP connector is an HTTP 1.1 protocol connector that allows Tomcat to function as a stand-alone web server. The key attributes specific to this connector are:

- **port**, the listening port number on which connections will be accepted. This defaults to 8080.
- **address**, the IP address of the interface the connector listening port will be bound to. This defaults to all available interfaces.
- **connectionTimeout**, the time in milliseconds the connector will wait for data in any given read. This value is passed as the Socket.setSoTimeout value. This defaults to 60000 or 1 minute.

Additional attribute descriptions may be found in the Tomcat website document: <http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/http11.html>

THE AJP CONNECTOR

The AJP connector supports versions 1.3 and 1.4 of the Apache AJP protocols and allows Tomcat to handle requests from an Apache web server. The key attributes specific to this connector are:

- **className**, must be set to AJP connection implementation class org.apache.ajp.tomcat4.Ajp13Connector.
- **port**, the listening port number on which connections will be accepted. This defaults to 8009.
- **address**, the IP address of the interface the connector listening port will be bound to. This defaults to all available interfaces.
- **connectionTimeout**, the time in milliseconds the connector will wait for data in any given read. This value is passed as the Socket.setSoTimeout value. This defaults to -1, or never timeout.

Additional attribute descriptions along with the required Apache configuration setup may be found in the Tomcat website document <http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/ajp.html>. We will also go through an example of configuring an AJP connector and an Apache web server later in this chapter.

THE WARP CONNECTOR

The Warp connector supports the Apache WARP protocol and allows Tomcat to handle requests from an Apache web server. Only limited success has been achieved with configuring this connector and the required mod_webapp Apache module is not yet available or easily built on all platforms.

Because of this we don't really support this connector. If you want to try it out, a good starting point is the following howto available on the Sun Dot-Com Builder site: http://dcb.sun.com/practices/howtos/tomcat_apache.jsp. If you get the Warp connector to work well, post message to the JBoss developer list at jboss-development@lists.sourceforge.net with the details.

Engine

Each Service must have a single Engine configuration. An Engine handles the requests submitted to a Service via the configured connectors. The child elements supported by the embedded service include Host, Logger, DefaultContext, Valve and Listener. The supported attributes include:

- **className**, the fully qualified class name of the `org.apache.catalina.Engine` interface implementation to use. If not specified this defaults to `org.apache.catalina.core.StandardEngine`.
- **defaultHost**, the name of a `Host` configured under the `Engine` that will handle requests with host names that do not match a `Host` configuration.
- **name**, a logical name to assign the `Engine`. It will be used in log messages produced by the `Engine`.

Additional information on the `Engine` element may be found in the Tomcat website document <http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/engine.html>.

Host

A `Host` element represents a virtual host configuration. It is a container for web applications with a specified DNS hostname. The child elements supported by the embedded service include `Alias`, `Logger`, `DefaultContext`, `Valve` and `Listener`. The supported attributes include:

- **className**, the fully qualified class name of the `org.apache.catalina.Host` interface implementation to use. If not specified this defaults to `org.apache.catalina.core.StandardHost`.
- **name**, the DNS name of the virtual host. At least one `Host` element must be configured with a name that corresponds to the `defaultHost` value of the containing `Engine`.

Additional information on the `Host` element may be found in the Tomcat website document <http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/host.html>.

ALIAS

The `Alias` element is an optional child element of the `Host` element. Each `Alias` content specifies an alternate DNS name for the enclosing `Host`.

DefaultContext

The `DefaultContext` element is a configuration template for web application contexts. It may be defined at the `Engine` or `Host` level. The child elements supported by the embedded service include `WrapperLifecycle`, `InstanceListener`, `WrapperListener`, and `Manager`. The supported attributes include:

- **className**, the fully qualified class name of the `org.apache.catalina.core.DefaultContext` implementation. This defaults to `org.apache.catalina.core.DefaultContext` and if overridden must be a subclass of `DefaultContext`.
- **cookies**, a flag indicating if sessions will be tracked using cookies. The default is true.
- **crossContext**, A flag indicating if the `ServletContext.getContext(String path)` method should return contexts for other web applications deployed in the calling web application's virtual host. The default is false.

MANAGER

The Manager element is an optional child of the DefaultContext configuration that defines a session manager. The supported attributes include:

- **className**, the fully qualified class name of the org.apache.catalina.Manager interface implementation. This defaults to org.apache.catalina.session.StandardManager.

Logger

The Logger element specifies a logging configuration for Engine, Hosts, and DefaultContexts. The supported attributes include:

- **className**, the fully qualified class name of the org.apache.catalina.Logger interface implementation. This defaults to org.jboss.web.catalina.Log4jLogger, and should be used for integration with JBoss server log4j system.

Valve

A Valve element configures a request pipeline element. A Valve is an implementation of the org.apache.catalina.Valve interface, and several standard Valves are available for use. The most commonly used Valve allows one to log access requests. Its supported attributes include:

- **className**, the fully qualified class name of the org.apache.catalina.Valve interface implementation. This must be org.jboss.web.catalina.valves.AccessLogValue.
- **directory**, the directory path into which the access log files will be created.
- **pattern**, a pattern specifier that defines the format of the log messages. This defaults to “common”.
- **prefix**, the prefix to add to each log file name. This defaults to “access_log”.
- **suffix**, the suffix to add to each log file name. This default to the empty string “” meaning that no suffix will be added.

Additional information on the Valve element and the available valve implementations may be found in the Tomcat website document <http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/valve.html>.

Listener

A Listener element configures a component life-cycle listener. You add a life-cycle listener using a Listener element with a **className** attribute giving the fully qualified name of the org.apache.catalina.LifecycleListener interface along with any additional properties supported by the listener implementation.

Using SSL with the JBoss/Tomcat bundle

There are two ways one can configure HTTP over SSL for the embedded Tomcat servlet container. If you want to only allow access over SSL encrypted connections then you can configure the primary connector using the EmbeddedCatalinaServiceSX **SecurityDomain** attribute. You set this to the JNDI name of the org.jboss.security.SecurityDomain implementation that JSSE should obtain the SSL Key-

Store from. This requires establishing a SecurityDomain using the org.jboss.security.plugins.JaasSecurityDomain MBean. These two steps are similar to the procedure we used in Chapter 8 to enable RMI with SSL encryption. A tomcat4-service.xml configuration file that illustrates the setup of SSL via this approach is given in Listing 9-4. This configuration includes the same JaasSecurityDomain setup as Chapter 8, but since the descriptor is not being deployed as part of a SAR that includes the chap8.keystore, you need to copy the chap8.keystore to the server/default/conf directory.

LISTING 9-4. The JaasSecurityDomain and EmbeddedCatalinaSX MBean configurations for setting up Tomcat-4.x to use SSL as its primary connector protocol.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- An example tomcat config that only uses SSL connectors.
-->
<!DOCTYPE server [
    <!ENTITY catalina.home ".../catalina">
]>

<!-- The service configuration for the embedded Tomcat4 web container
-->
<server>

    <classpath codebase="file:&catalina.home;/common/lib/" archives="*"/>
    <classpath codebase="file:&catalina.home;/server/lib/" archives="*"/>
    <classpath codebase="file:&catalina.home;/bin/" archives="*"/>
    <classpath codebase="file:&catalina.home;/lib/" archives="*"/>
    <classpath codebase=". " archives="tomcat4-service.jar"/>

    <!-- The SSL domain setup -->
    <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
        name="Security:name=JaasSecurityDomain,domain=RMI+SSL">
        <constructor>
            <arg type="java.lang.String" value="RMI+SSL"/>
        </constructor>
        <attribute name="KeyStoreURL">chap8.keystore</attribute>
        <attribute name="KeyStorePass">rmi+ssl</attribute>
    </mbean>

    <!-- The embedded Tomcat-4.x(Catalina) service configuration -->
    <mbean code="org.jboss.web.catalina.EmbeddedCatalinaServiceSX"
        name="DefaultDomain:service=EmbeddedCatalinaSX">
        <attribute name="CatalinaHome">&catalina.home;</attribute>
        <attribute name="Config">
            <Server>
                <Service name = "JBoss-Tomcat">
                    <Engine name="MainEngine" defaultHost="localhost">
                        <Logger className = "org.jboss.web.catalina.Log4jLogger"
                            verbosityLevel = "trace" category =
                            "org.jboss.web.localhost.Engine"/>
                        <Host name="localhost">
                            <Valve className = "org.apache.catalina.valves.AccessLogValve"
                                prefix = "localhost_access" suffix = ".log"
                                pattern = "common" directory = ".../server/default/log" />
                        </Host>
                </Service>
            </Server>
        </attribute>
    </mbean>
```

```
        <DefaultContext cookies = "true" crossContext = "true" override =
"true" />
    </Host>
</Engine>

<!-- SSL/TLS Connector configuration using the SSL domain keystore -->
<Connector className =
"org.apache.catalina.connector.http.HttpConnector"
    port = "443" scheme = "https" secure = "true">
    <Factory className =
"org.jboss.web.catalina.security.SSLServerSocketFactory"
        securityDomainName = "java:/jaas/RMI+SSL" clientAuth = "false"
        protocol = "TLS" />
</Connector>
</Service>
</Server>
</attribute>
</mbean>

</server>
```

A quick test of this config can be made by accessing the JMX console web application using this URL <https://localhost/jmx-console/index.jsp>.

*Note: if you're running on a *nix system (Linux, Solaris, OS X) that only allows root to open ports below 1024 you will need to change the port number above to something like 8443.*

Alternatively, if one wants to support both access using non-SSL and SSL, you can do this by adding a Connector configuration to the EmbeddedCatalinaSX MBean. This can be done using a JBoss specific connector socket factory that allows one to obtain the JSSE server certificate information from a JBossSX SecurityDomain. A tomcat4-service.xml configuration file that illustrates such a setup of SSL is given in Listing 9-5.

LISTING 9-5. The JaasSecurityDoman and EmbeddedCatalinaSX MBean configurations for setting up Tomcat-4.x to use both non-SSL and SSL enabled HTTP connectors.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- An example tomcat config that only uses both SSL and non-SSL connectors.
-->
<!DOCTYPE server [
    <!ENTITY catalina.home "&../catalina;">
]>

<!-- The service configuration for the embedded Tomcat4 web container
-->
<server>

    <classpath codebase="file:&catalina.home;/common/lib/" archives="*"/>
    <classpath codebase="file:&catalina.home;/server/lib/" archives="*"/>
    <classpath codebase="file:&catalina.home;/bin/" archives="*"/>
    <classpath codebase="file:&catalina.home;/lib/" archives="*"/>
```

```
<classpath codebase=". " archives="tomcat4-service.jar"/>

<!-- The SSL domain setup -->
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="Security:name=JaasSecurityDomain,domain=RMI+SSL">
  <constructor>
    <arg type="java.lang.String" value="RMI+SSL"/>
  </constructor>
  <attribute name="KeyStoreURL">chap8.keystore</attribute>
  <attribute name="KeyStorePass">rmi+ssl</attribute>
</mbean>

<!-- The embedded Tomcat-4.x(Catalina) service configuration -->
<mbean code="org.jboss.web.catalina.EmbeddedCatalinaServiceSX"
       name="DefaultDomain:service=EmbeddedCatalinaSX">
  <attribute name="CatalinaHome">&catalina.home;</attribute>
  <attribute name="Config">
    <Server>
      <Service name = "JBoss-Tomcat">
        <Engine name="MainEngine" defaultHost="localhost">
          <Logger className = "org.jboss.web.catalina.Log4jLogger"
                 verbosityLevel = "trace" category =
"org.jboss.web.localhost.Engine"/>
          <Host name="localhost">
            <Valve className = "org.apache.catalina.valves.AccessLogValve"
                  prefix = "localhost_access" suffix = ".log"
                  pattern = "common" directory = "../server/default/log" />
            <DefaultContext cookies = "true" crossContext = "true" override =
"true" />
          </Host>
        </Engine>
      <!-- HTTP Connector configuration -->
      <Connector className =
"org.apache.catalina.connector.http.HttpConnector"
          port = "8080" redirectPort = "443"/>
      <!-- SSL/TLS Connector configuration using the SSL domain keystore -->
      <Connector className =
"org.apache.catalina.connector.http.HttpConnector"
          port = "443" scheme = "https" secure = "true">
        <Factory className =
"org.jboss.web.catalina.security.SSLServerSocketFactory"
          securityDomainName = "java:/jaas/RMI+SSL" clientAuth = "false"
          protocol = "TLS"/>
      </Connector>
    </Service>
  </Server>
  </attribute>
</mbean>

</server>
```

Both approaches work so which you choose is a matter of preference. Note that if you try to test this configuration using the self-signed certificate from the Chapter 8 chap8.keystore and attempt to

access content over an https connection, your browser should display a warning dialog indicating that it does not trust the certificate authority that signed the certificate of the server you are connecting to. For example, when the first configuration example was tested, IE 5.5 showed the initial security alert dialog listed in Figure 9-3. Figure 9-4 shows the server certificate details. This is the expected behavior as anyone can generate a self-signed certificate with any information they want, and a web browser should warn you when such a secure site is encountered.



FIGURE 9-3. The Internet Explorer 5.5 security alert dialog.

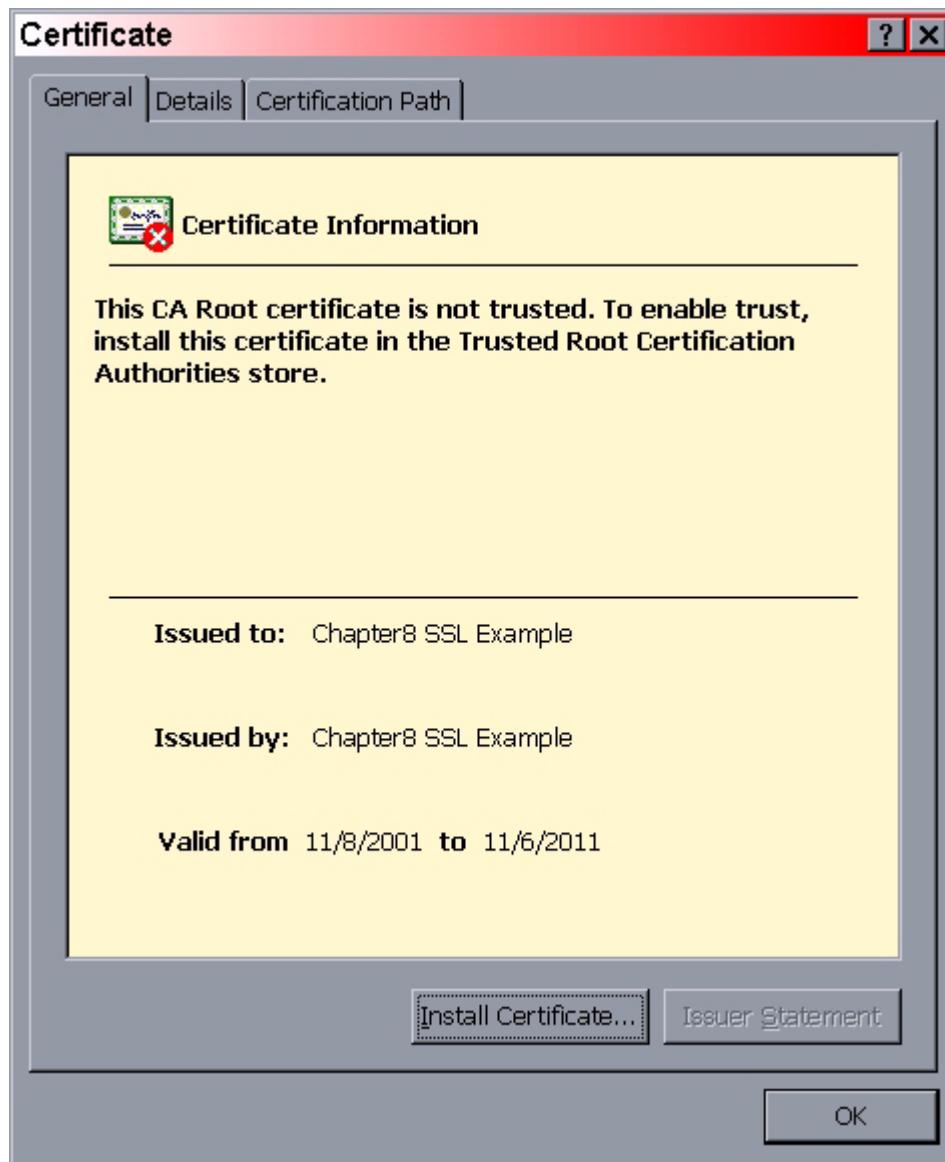


FIGURE 9-4. The Internet Explorer 5.5 SSL certificate details dialog.

Setting up Virtual Hosts with the JBoss/Tomcat-4.x bundle

As of the 2.4.5 release, support for virtual hosts has been added to the servlet container layer. Virtual hosts allow you to group web applications according to the various DNS names by which the machine running JBoss is known. As an example, consider the `tomcat4-service.xml` configuration file given in Listing 9-6. This configuration defines a default host named `localhost` and a second host named `banshee.starkinternational.com`. The `banshee.starkinternational.com` also has the alias `www.starkinternational.com` associated with it.

LISTING 9-6. An example virtual host configuration.

```

<!-- The embedded Tomcat-4.x(Catalina) service configuration -->
<mbean code="org.jboss.web.catalina.EmbeddedCatalinaServiceSX"
       name="DefaultDomain:service=EmbeddedCatalinaSX">
  <attribute name="Config">
    <Server>
      <Service name = "JBoss-Tomcat">
        <Engine name="MainEngine" defaultHost="localhost">
          <Logger className = "org.jboss.web.catalina.Log4jLogger"
                  verbosityLevel = "debug" category = "org.jboss.web.CatalinaEngine"/>
          <DefaultContext cookies = "true" crossContext = "true" override =
"true" />
          <Host name="localhost">
            <Logger className = "org.jboss.web.catalina.Log4jLogger"
                    verbosityLevel = "debug" category =
"org.jboss.web.Host=localhost"/>
            <Valve className = "org.apache.catalina.valves.AccessLogValve"
                  prefix = "localhost_access" suffix = ".log"
                  pattern = "common" directory = "../server/default/log" />
          </Host>
          <Host name="banshee.starkinternational.com">
            <Alias>www.starkinternational.com</Alias>
            <Logger className = "org.jboss.web.catalina.Log4jLogger"
                    verbosityLevel = "debug" category = "org.jboss.web.Host=www"/>
            <Valve className = "org.apache.catalina.valves.AccessLogValve"
                  prefix = "www_access" suffix = ".log"
                  pattern = "common" directory = "../server/default/log" />
          </Host>
        </Engine>

        <!-- A HTTP Connector on port 8080 -->
        <Connector className =
"org.apache.catalina.connector.http.HttpConnector"
          port = "8080" minProcessors = "3" maxProcessors = "10" enableLookups
= "true"
          acceptCount = "10" connectionTimeout = "60000"/>
      </Service>
    </Server>
  </attribute>
</mbean>

```

When a WAR is deployed, it will be by default associated with the virtual host whose name matches the defaultHost attribute of the containing Engine. To deploy a WAR to a specific virtual host you need to use the jboss-web.xml descriptor and the virtual-host element. For example, to deploy a WAR to the virtual host www.starkinternational.com virtual host alias, the following jboss-web.xml descriptor would be need to be included in the WAR WEB-INF directory. This demonstrates that an alias of the virtual host can be used in addition to the Host name attribute value.

LISTING 9-7. An example jboss-web.xml descriptor for deploying a WAR to the www.starkinternational.com virtual host

```

<jboss-web>
  <context-root></context-root>
  <virtual-host>www.starkinternational.com</virtual-host>

```

```
</jboss-web>
```

When such a WAR is deployed, the server console shows that the WAR is in fact deployed to the www.starkinternational.com virtual host as seen by the “Host=www” category name in the log statements.

LISTING 9-8. Output from the www.starkinternational.com Host component when the Listing 9-7 WAR is deployed.

```
13:11:52,948 INFO [MainDeployer] Starting deployment of package: file:/tmp/jboss-3.0.1RC1/server/default/deploy/chap9-ex1.war
13:11:52,980 INFO [EmbeddedCatalinaServiceSX] deploy, ctxPath=, warUrl=file:/tmp/jboss-3.0.1RC1/server/default/tmp/deploy/server/default/deploy/chap9-ex1.war/78.chap9-ex1.war
13:11:52,982 INFO [EmbeddedCatalinaServiceSX] ClusteredHTTPSessionService not found
13:11:53,018 INFO [Host=www] WebappLoader[]: Deploying class repositories to work directory /tmp/jboss-3.0.1RC1/catalina/work/MainEngine/banshee.starkinternational.com/_/
13:11:53,074 INFO [Host=www] StandardManager[]: Seeding of random number generator has been completed
13:11:53,073 INFO [Host=www] StandardManager[]: Seeding random number generator class java.security.SecureRandom
13:11:53,564 INFO [Host=www] ContextConfig[]: Added certificates -> request attribute Valve
13:11:53,670 INFO [EmbeddedCatalinaServiceSX] Using Java2 parent classloader delegation: true
13:11:53,671 INFO [Host=www] StandardWrapper[:default]: Loading container servlet default
13:11:53,672 INFO [Host=www] default: init
13:11:53,673 INFO [Host=www] StandardWrapper[:invoker]: Loading container servlet invoker
13:11:53,673 INFO [Host=www] invoker: init
13:11:53,674 INFO [Host=www] jsp: init
13:11:53,803 INFO [MainDeployer] Deployed package: file:/tmp/jboss-3.0.1RC1/server/default/deploy/chap9-ex1.war
```

Using Apache with the JBoss/Tomcat-4.x bundle

To enable the use of Apache as a front-end web server that delegates servlet requests to a JBoss/Tomcat bundle, you need to configure an appropriate connector in the EmbeddedCatalinaSX MBean definition. For example, to configure the use of the Ajpv13 protocol connector with the Apache mod_jk module, you would use a configuration like that given in Listing 9-9.

LISTING 9-9. An example EmbeddedCatalinaSX MBean configuration that supports integration with Apache using the Ajpv13 protocol connector.

```
<server>
  <!-- The embedded Tomcat-4.x(Catalina) service configuration -->
  <mbean code="org.jboss.web.catalina.EmbeddedCatalinaServiceSX"
        name="DefaultDomain:service=EmbeddedCatalinaSX">
    <attribute name="Config">
```

```
<Server>
  <Service name = "JBoss-Tomcat">
    <Engine name="MainEngine" defaultHost="localhost">
      <Logger className = "org.jboss.web.catalina.Log4jLogger"
        verbosityLevel = "trace" category =
"org.jboss.web.localhost.Engine"/>
        <Host name="localhost">
          <Valve className = "org.apache.catalina.valves.AccessLogValve"
            prefix = "localhost_access" suffix = ".log"
            pattern = "common" directory = "../server/default/log" />
          <DefaultContext cookies = "true" crossContext = "true" override =
"true" />
        </Host>
      </Engine>

      <!-- AJP13 Connector configuration -->
      <Connector className="org.apache.ajp.tomcat4.Ajp13Connector"
        port="8009" minProcessors="5" maxProcessors="75"
        acceptCount="10" />
    </Service>
  </Server>
</attribute>
</mbean>
</server>
```

The configuration of the Apache side proceeds as it normally would as bundling Tomcat inside of JBoss does not affect the how Apache interacts with Tomcat. For example, a fragment of an httpd.conf configuration to test the Listing 9-9 setup with a WAR deployed with a context root of “/jbosstest” might look like:

```
...
LoadModule      jk_module      libexec/mod_jk.so
AddModule      mod_jk.c

<IfModule mod_jk.c>
  JkWorkersFile  /tmp/workers.properties
  JkLogFile      /tmp/mod_jk.log
  JkLogLevel     debug
  JkMount        /jbosstest/* ajp13
</IfModule>
```

Other Apache to Tomcat configurations would follow the same pattern. All that would change is the Connector element definition that is placed into the EmbeddedCatalinaSX MBean configuration.

Using Clustering

As of the JBoss 3.0.1 release, there is support for clustering in the Tomcat embedded service. The steps to setup clustering of Tomcat embedded containers is:

1. If you are using a load balancer, make sure that your setup uses sticky sessions. This means that if a user that starts a session on node A, all subsequent requests are forwarded to node A as

long node A is up and running. For configuration of the Apache webserver sticky sessions see <http://www.ubeans.com/tomcat/index.html> for details.

2. Make sure that cluster-service.xml and jbossha-httpsession.sar are in your configuration file set deploy directory, e.g. {JBOSS_HOME}/server/default/deploy. The cluster-service.xml is not included in the default configuration, but can be found in JBOSS_DIST/server/all/deploy. The jbossha-httpsession.sar can be found in the JBOSS_DIST/docs/examples/clustering directory. You also need the javagroups20.jar. This can be found in the JBOSS_DIST/server/all/lib directory.
3. Start JBoss to check if your setup works. Look at the JMX management console (<http://localhost:8080/jmx-console/>). Find the MBean jboss:service=ClusteredHttpSession. The "StateString" must be "Started". If it is "Stopped" look in the server's logfile.
4. To enable clustering of your web applications you must mark them as distributable using the servlet 2.3 web.xml descriptor. For example:

```
<?xml version="1.0"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <distributable/>
...
</web-app>
```

5. Deploy your war as usual and it should not be clustered.

JBoss/Jetty-4.0.0 Bundle Notes

Jetty is a pure Java web server and servlet container compliant to the HTTP1.1, Servlet 2.3 & JSP 1.2 specifications developed by **Mort Bay Consulting** (<http://www.mortbay.com>). It has been designed to be fast, lightweight, extensible, and embeddable. This section discusses the embedding of Jetty within JBoss, but for more general information on Jetty, visit the **Jetty website** (<http://jetty.mort-bay.org>).

Integration with JBoss

Jetty is fully integrated with the JBoss environment in terms of:

- In-JVM optimized calls. The overhead of RMI is avoided when the servlet and EJB containers are run in the same JVM.
- Implementing a web container service. The Jetty integration extends the [org.jboss.web.AbstraceWebContainer](#) class to enable Jetty to conform to the standard JBoss web container service interface. This allows the Jetty Service to be stopped and restarted, to hot-deploy webapps and for those webapps to be able to reference EJBs, resources and other objects in the J2EE JNDI environment.

- Logging. Debug and informational log output from the Jetty Service is adapted to the standard JBoss logging service.
- Security. The Jetty integration classes adapt the servlet security environment to the JBoss security environment. This allows webapps performing basic or form based authentication to transparently access the JBossSX framework.
- JMX. As a compliant JBoss service, Jetty can be controlled from the jmx-console web application available on port 8080. Jetty makes available each of its constituent components as mbeans allowing detailed management of configuration, debugging and statistics gathering. Additionally, Jetty creates an mbean for every deployed web application context, allowing individual contexts to be stopped and (re)started without undeploying the webapp itself.
- Clustered Sessions. The clustered HTTP Session service can be used to provide distributed sessions.

Deployment

Jetty is packaged as a service archive file called `jbossweb.sar`. It deploys automatically with JBoss with a default configuration:

1. it will listen on port 8080 for HTTP requests (note that as no demonstration webapp is provided, hitting `localhost:8080/` will result in your receiving a "404 NotFound")
2. the HTTP request log is written to the standard JBoss log directory as files with names of the form `yyyy_mm_dd.request.log` which rollover daily
3. output from Jetty such as debug and informational messages are directed to the standard JBoss log

Configuration

The default configuration can be modified by editing the Jetty configuration file found inside the sar as `jbossweb.sar/META-INF/jboss-service.xml`. JBoss will reload and restart Jetty with its new configuration when you save the `jboss-service.xml` file. Alternatively, for non-permanent configuration changes, you can use the JMX Agent on port 8082.

The default Jetty `jboss-service.xml` file looks like:

LISTING 9-10. Standard Jetty service configuration file `jboss-service.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <!-- ====== -->
  <!-- Web Container -->
  <!-- ====== -->

  <!--
    | Be sure to check that the configuration values are valid for your
    | environment.
  -->
```

```
-->

<mbean code="org.jboss.jetty.JettyService" name="jboss.web:service=Jetty">

<!-- ===== -->
<!-- Uncomment the following line ONLY if you want to provide a custom -->
<!-- webdefault.xml file in place of the standard one. Place your -->
<!-- file in the src/etc directory to have it automatically included -->
<!-- in the build. -->
<!-- ===== -->

<!--
<attribute name="WebDefault">webdefault.xml</attribute>
-->

<!-- ===== -->
<!-- If true, .war files are unpacked to a temporary directory. This -->
<!-- is useful with JSPs. -->
<!-- ===== -->

<attribute name="UnpackWars">true</attribute>

<!-- ===== -->
<!-- If true, Jetty first delegates loading a class to the webapp's -->
<!-- parent class loader (a la Java 2). If false, Jetty follows the -->
<!-- Servlet 2.3 specification, and tries the webapp's own loader -->
<!-- first (for "non-system" classes) -->
<!-- ===== -->

<attribute name="Java2ClassLoaderCompliance">true</attribute>

<!-- ===== -->
<!-- Configuring Jetty. The XML fragment contained in the -->
<!-- name="ConfigurationElement" attribute is a Jetty-style -->
<!-- configuration specification. It is used to configure Jetty with -->
<!-- a listener on port 8080, and a HTTP request log location. -->
<!-- The placement here of other Jetty XML configuration statements -->
<!-- for deploying webapps etc is not encouraged: if you REALLY NEED -->
<!-- something extra, place it in WEB-INF/jetty-web.xml files -->
<!-- ===== -->

<attribute name="ConfigurationElement">
    <Configure class="org.mortbay.jetty.Server">

        <!-- ===== -->
        <!-- Add the listener -->
        <!-- ===== -->
        <Call name="addListener">
            <Arg>
                <New class="org.mortbay.http.SocketListener">
                    <Set name="Port"><SystemProperty name="jetty.port" default="8080"/>
            </Set>
            <Set name="MinThreads">5</Set>
            <Set name="MaxThreads">255</Set>
            <Set name="MaxIdleTimeMs">30000</Set>
        </Call>
    </Configure>
</attribute>
```

```

        <Set name="MaxReadTimeMs">10000</Set>
        <Set name="MaxStopTimeMs">5000</Set>
        <Set name="LowResourcePersistTimeMs">5000</Set>
        </New>
    </Arg>
</Call>

<!-- ===== -->
<!-- Add the HTTP request log -->
<!-- ===== -->
<Set name="RequestLog">
    <New class="org.mortbay.http.NCSARequestLog">
        <Arg><SystemProperty name="jboss.server.home.dir"/><SystemProperty
name="jetty.log" default="/log"/>/yyyy_mm_dd.request.log
        </Arg>
        <Set name="retainDays">90</Set>
        <Set name="append">true</Set>
        <Set name="extended">true</Set>
        <Set name="LogTimeZone">GMT</Set>
    </New>
</Set>

<!-- ===== -->
<!-- Uncomment and set at least the Keystore, Password and -->
<!-- KeyPassword fields to configure an SSL listener -->
<!-- ===== -->
<!-- -->
<Call name="addListener">
    <Arg>
        <New class="org.mortbay.http.SunJsseListener">
            <Set name="Port">8443</Set>
            <Set name="MinThreads">5</Set>
            <Set name="MaxThreads">255</Set>
            <Set name="MaxIdleTimeMs">30000</Set>
            <Set name="MaxReadTimeMs">10000</Set>
            <Set name="MaxStopTimeMs">5000</Set>
            <Set name="LowResourcePersistTimeMs">2000</Set>
            <Set name="Keystore"><SystemProperty name="jetty.home"
default=". "/>/etc/demokeystore</Set>
            <Set name="Password">dummy</Set>
            <Set name="KeyPassword">dummy</Set>
        </New>
    </Arg>
</Call>
-->
</Configure>
</attribute>

<!-- ===== -->
<!-- Options for distributed session management are: -->
<!--     org.jboss.jetty.session.CoarseDistributedStore -->
<!--     org.jboss.jetty.session.ClusteredStore -->
<!-- ===== -->

```

```
<attribute  
name="HttpSessionStorageStrategy">org.jboss.jetty.session.ClusteredStore</  
attribute>  
  
<!-- ===== -->  
<!-- Options for synchronizing distributed sessions: -->  
<!-- never/idle/request/<num-seconds> -->  
<!-- ===== -->  
  
<attribute name="HttpSessionSnapshotFrequency">never</attribute>  
  
<!-- ===== -->  
<!-- Options for the notification of HttpSessionActivationListeners -->  
<!-- around snapshotting are: -->  
<!-- neither -->  
<!-- activate -->  
<!-- passivate -->  
<!-- both -->  
<!-- ===== -->  
  
<attribute name="HttpSessionSnapshotNotificationPolicy">neither</attribute>  
  
<!-- ===== -->  
<!-- If you require JAAS authentication, configure the name of the -->  
<!-- attribute in which you expect to find the JAAS active subject: -->  
<!-- -->  
<!-- Commenting out this configuration will disable JAAS support -->  
<!-- ===== -->  
  
<attribute name="SubjectAttributeName">j_subject</attribute>  
</mbean>  
<!-- ===== -->  
<!-- ===== -->  
  
</server>
```

UNPACKING WARS ON DEPLOYMENT

By default, Jetty will unpack your war as it is deployed. This is because JSP compilers typically can only compile unpacked classes. To change this behaviour, set the following property:

```
<attribute name="UnpackWars">false</attribute>
```

CLASSLOADING BEHAVIOUR

By default, Jetty follows the Java 2 specification for class loading. That is, when loading a class, Jetty first delegates to the webapp's parent class loader. This should be the norm within JBoss to take advantage of the unified class loading mechanism. However, it is possible to force Jetty to follow the Servlet 2.3 class loading specification, whereby Jetty first tries the webapp's own loader when loading "non-system" classes. If you are sure you need this kind of behavior, set the following:

```
<attribute name="Java2ClassLoaderCompliance">false</attribute>
```

CHANGING THE DEFAULT HTTP LISTENER PORT

By default, Jetty listens on port 8080. To change this, modify the Port property of the addListener element:

```
<Set name="Port"><SystemProperty name="jetty.port" default="9090"/></Set>
```

CHANGING OTHER HTTP LISTENER PORT ATTRIBUTES

The jboss-service.xml file specifies several extra attributes for the operation of Jetty which you may find useful to customise to your environment:

LISTING 9-11. Jetty listener port attributes

```
<Call name="addListener">
  <Arg>
    <New class="org.mortbay.http.SocketListener">
      <Set name="Port"><SystemProperty name="jetty.port" default="8080"/></Set>
      <Set name="Address">localhost</Set>
      <Set name="MinThreads">5</Set>
      <Set name="MaxThreads">255</Set>
      <Set name="MaxIdleTimeMs">30000</Set>
      <Set name="MaxReadTimeMs">10000</Set>
      <Set name="MaxStopTimeMs">5000</Set>
      <Set name="LowResourcePersistTimeMs">5000</Set>
    </New>
  </Arg>
</Call>
```

Attribute	Description
MinThreads	The minimum number of threads allowed.
MaxThreads	The maximum number of threads allowed.
MaxIdleTimeMs	Time in MS that a thread can be idle before it may expire.
MaxReadTimeMs	The maximum time in milliseconds that a read can be idle.
MaxStopTimeMs	Time in MS that a thread is allowed to run when stopping.
LowResourcePersistTimeMs	Time in ms to persist idle connections if low on resources.
IntegralPort	Port to redirect to for integral connections specified in a security constraint.
IntegralScheme	Protocol to use for integral redirections.
ConfidentialPort	Port to redirect to for confidential connections. 0 if not supported.
ConfidentialScheme	Protocol to use for confidential redirections.
LingerTimeSecs	The maximum time in seconds that a connection lingers during close handshaking.

USING SSL

The jboss-service.xml file includes a commented out example of how to set up Jetty for SSL:

```
<!-- Uncomment and set at least the Keystore, Password and      -->
<!-- KeyPassword fields to configure an SSL listener           -->
```

```
<!-- ===== -->
<!--
<Call name="addListener">
  <Arg>
    <New class="org.mortbay.http.SunJsseListener">
      <Set name="Port">8443</Set>
      <Set name="MinThreads">5</Set>
      <Set name="MaxThreads">255</Set>
      <Set name="MaxIdleTimeMs">30000</Set>
      <Set name="MaxReadTimeMs">10000</Set>
      <Set name="MaxStopTimeMs">5000</Set>
      <Set name="LowResourcePersistTimeMs">2000</Set>
      <Set name="Keystore">
        <SystemProperty name="jetty.home" default=". "/>/etc/demokeystore</Set>
          <Set name="Password">dummy</Set>
          <Set name="KeyPassword">dummy</Set>
        </New>
      </Arg>
    </Call>
  -->
```

Additional tips on the use of this SSL setup can be found on the Jetty website at <http://jetty.mort-bay.org/jetty/doc/SslListener.html>.

As of JBoss-3.0.2, an alternate SSL listener exists that integrates with the JaasSecurityDomain MBean used by the Tomcat, and RMI/SSL configurations. Instead of supplying the keystore location and passwords, you specify the name of the security domain associated with the JaasSecurityDomain MBean. Listing 9-12 shows an example of how to configure an SSL listener that uses the same key-store configured in the RMI/SSL example configuration of Listing 8-20.

LISTING 9-12. Using the SecurityDomainListener configure SSL for Jetty

```
<Call name="addListener">
  <Arg>
    <New class="org.jboss.jetty.http.SecurityDomainListener">
      <Set name="Address">thestore.dom.com</Set>
      <Set name="Port">8443</Set>
      <Set name="MinThreads">5</Set>
      <Set name="MaxThreads">255</Set>
      <Set name="MaxIdleTimeMs">30000</Set>
      <Set name="MaxReadTimeMs">10000</Set>
      <Set name="MaxStopTimeMs">5000</Set>
      <Set name="LowResourcePersistTimeMs">2000</Set>
      <Set name="SecurityDomain">java:/jaas/RMI+SSL</Set>
    </New>
  </Arg>
</Call>
```

USING JAAS

JAAS support is configurable across the JettyService instance via specifying the JAAS name of the attribute in which the active subject is transported:

```
<attribute name="SubjectAttributeName">j_subject</attribute>
```

USING DISTRIBUTED HTTPSESSIONS

An `HttpSession` is an object used in a webapp to store conversational state between requests. It is configured in the webapp by specifying 'distributable' in it's `WEB-INF/web.xml`.

The J2EE specification requires that a 'distributable' application may be 'migrated' between nodes of a cluster - i.e. taken down on one node and brought up on another. Extant `HttpSessions` must continue to be available to the new webapp instance. Many appservers extend this functionality from simply allowing migration to providing failover i.e. if a webapp is not undeployed from its node cleanly (e.g. the node crashes, hangs, becomes overloaded) its `HttpSessions` are still made available to other instances of the same webapp within the cluster.

This extension is problematic since J2EE requires that on being undeployed, a distributed webapp should notify `HttpSession` attributes implementing `HttpSessionActivationListener` before passivation/distributing them. When the webapp has been re-deployed and it re-activates an `HttpSession`, the same attributes must be notified again. If, because of the reuse of this functionality to provide fail-over, attributes do not receive passivation events on one node before receipt of activation events on another, an asymmetry - which would not happen on a fully compliant appserver - occurs.

The Jetty integration allows the user to specify whether this extended behavior (called 'snapshotting') is required and, if so, exactly what combination of events attributes should expect.

Configuring Session Distribution

In order to use distributed http sessions, you need to perform the following series of steps:

1. Edit the `jbossweb.sar/META-INF/jboss-service.xml` file:
1. Ensure the following property is set (as it is by default):

```
<attribute name="HttpSessionStorageStrategy">
    org.jboss.jetty.session.ClusteredStore
</attribute>
```

2. Set the snapshot frequency which affects the synchronization of distributed sessions:

```
<attribute name="HttpSessionSnapshotFrequency">never</attribute>.
```

Options for the value are:

- never
- idle
- request
- <number of seconds>

3. Set the snapshot notification policy, which will affect when `HttpSessionActivationListeners` are notified:

```
<attribute name="HttpSessionSnapshotNotificationPolicy">neither</attribute>
```

Options for the value are:

- never

- activate
- passivate
- both

Other Jetty Configuration Tips

Deploying a war to context '/'

Deploying a webapp called `foo.war` will result in it being deployed at context `/foo`. To deploy it instead to the root context, choose one of the following mechanisms:

1. The standard J2EE way: wrap your `.war` in an `.ear` and in the `.ear`'s `META-INF/application.xml` you can specify the required context.
2. The proprietary JBoss extension: put a `jboss-web.xml` into your `.war`'s `WEB-INF` directory and specify the context root in that.
3. Tomcat style: call the file `ROOT.war` and deploy it.

Using virtual hosts

This is supported as of JBoss2.4.5 and higher via a proprietary extension mechanism.

To define a virtual host, add a line of the following form to your webapp's `WEB-INF/jboss-web.xml` file (and set up your DNS to route requests for this hostname):

```
<virtual-host>myvirtualhost</virtual-host>
```

You can also specify a context path in the `WEB-INF/jboss-web.xml` file like so:

```
<context-root>/mycontextpath</context-root>
```

You should be careful as a context path specification in a `META-INF/application.xml` file will take precedence over the `WEB-INF/jboss-web.xml` specification.

RUNNING ON PORT 80

As port 80 is a privileged port, it is usually better to set up a mapping from it to a non-privileged port (such as 8080) where the HTTP server is running. The set-up required is operating system specific. For a how-to for Unix systems, see <http://jetty.mortbay.org/jetty/doc/User80.html>.

RUNNING WITH APACHE FRONT-ENDING JETTY

It is not necessary to configure Apache to use Jetty, as Jetty is a fully featured HTTP server. However, if you have a special requirement for Apache, you can layer it in front of Jetty. Instructions for doing this can be found at <http://jetty.mortbay.com/jetty/doc/JettyWithApache.html>

This chapter discusses useful MBean services that are not discussed elsewhere as they are utility services not necessary for running JBoss.

System Properties Management

The management of system properties can be done using the [org.jboss.varia.property.SystemPropertiesService](#) MBean. It supports setting of the VM global property values just as [java.lang.System.setProperty](#) method and the -Dproperty=value VM command line arguments do.

Its configurable attributes include:

- **Properties**: a specification of multiple property name=value pairs using the [java.util.Properties.load\(java.io.InputStream\)](#) method format. Each property=value statement is given on a separate line within the body of the Properties attribute element.
- **URLList**: a comma separated list of URL strings from which to load properties file formatted content. If a component in the list is a relative path rather than a URL it will be treated as a file path relative to the <jboss-dist>/server/<config> directory. For example, a component of “conf/local.properties” would be treated as a file URL that points to the <jboss-dist>/server/default/conf/local.properties file when running with the “default” configuration file set.

Both attributes are illustrated in Listing 10-1.

LISTING 10-1. An example SystemPropertiesService jboss-service descriptor

```
<server>
<mbean code="org.jboss.varia.property.SystemPropertiesService"
```

```
name="jboss.util:type=Service,name=SystemProperties">

<!-- Load properties from each of the given comma seperated URLs -->
<attribute name="URLList">
    http://somehost/some-location.properties,
    ./conf/somelocal.properties
</attribute>

<!-- Set properties using the properties file style. -->
<attribute name="Properties">
    property1=This is the value of my property
    property2=This is the value of my other property
</attribute>

</mbean>
</server>
```

Property Editor Management

Support for managing [java.bean.PropertyEditor](#) instances is available through the [org.jboss.varia.property.PropertyEditorManagerService](#) MBean. This is a simple service that helps define [PropertyEditors](#) using the [java.bean.PropertyEditorManager](#) class. As of JBoss-3.0.3, this service is used in the main jboss-service.xml file to preload the custom JBoss [PropertyEditor](#) implementations. This is necessary for some JDK1.3.0 VMs that will only load PropertyEditors from the system classpath.

Its supported attributes include:

- **BootstrapEditors:** This is a listing of property_editor_class=editor_value_type_class pairs defining the [PropertyEditor](#) to type mappings that should be preloaded into the [PropertyEditorManager](#) class using its `registerEditor(Class targetType, Class editorClass)` method. The value type of this attribute is a string so that it may be set from a string without requiring a custom [PropertyEditor](#).
- **Editors:** This serves the same function as the BootstrapEditors attribute, but its type is a [java.util.Properties](#) class, and so setting this from a string value requires a custom [PropertyEditor](#) for [Properties](#). In situations where custom [PropertyEditors](#) can be loaded from the thread context class loader, this may be used instead of the BootstrapEditors attribute.
- **EditorSearchPath:** This attribute allows one to set the [PropertyEditorManager](#) editor packages search path.

Scheduling Tasks

Java includes a simple timer based capability through the `java.util.Timer` and `java.util.TimerTask` utility classes. JMX also includes a mechanism for scheduling JMX notifications at a given time with an optional repeat interval as the `javax.management.timer.TimerMBean` agent service.

JBoss includes two variations of the JMX timer service in the `org.jboss.varia.scheduler.Scheduler` and `org.jboss.varia.scheduler.ScheduleManager` MBeans. Both MBeans rely on the JMX timer service for the basic scheduling. They extend the behavior of the timer service as described in the following sections.

org.jboss.varia.scheduler.Scheduler

The Scheduler differs from the TimerMBean in that the Scheduler directly invokes a callback on an instance of a user defined class, or an operation of a user specified MBean.

- **InitialStartDate:** Date when the initial call is scheduled. It can be either:
 - NOW: date will be the current time plus 1 seconds
 - A number representing the milliseconds since 1/1/1970
 - Date as String able to be parsed by SimpleDateFormat with default format pattern “M/d/yy h:mm a”. If the date is in the past the Scheduler will search a start date in the future with respect to the initial repetitions and the period between calls. This means that when you restart the MBean (restarting JBoss etc.) it will start at the next scheduled time. When no start date is available in the future the Scheduler will not start.
- For example, if you start your Schedulable everyday at Noon and you restart your JBoss server then it will start at the next Noon (the same if started before Noon or the next day if start after Noon).
- **InitialRepetitions:** The number of times the scheduler will invoke the target’s callback. If -1 then the callback will be repeated until the server is stopped.
- **StartAtStartup:** A flag that determines if the `Scheduler` will start when it receives its `startService` life cycle notification. If true the `Scheduler` starts on its startup. If false, an explicit `startSchedule` operation must be invoked on the `Scheduler` to begin.
- **SchedulePeriod:** The interval between scheduled calls in milliseconds. This value must be bigger than 0.
- **SchedulableClass:** The fully qualified class name of the `org.jboss.varia.scheduler.Schedulable` interface implementation that is to be used by the `Scheduler`. The `SchedulableArguments` and `SchedulableArgumentTypes` must be populated to correspond to the constructor of the `Schedulable` implementation.
- **SchedulableArguments:** A comma separated list of arguments for the `Schedulable` implementation class constructor. Only primitive data types, `String` and classes with a constructor that accepts a `String` as its sole argument are supported.
- **SchedulableArgumentTypes:** A comma separated list of argument types for the `Schedulable` implementation class constructor. This will be used to find the correct constructor via reflection.

Only primitive data types, `String` and classes with a constructor that accepts a `String` as its sole argument are supported.

- **SchedulableMBean**: Specifies the fully qualified JMX `ObjectName` name of the schedulable MBean to be called. If the MBean is not available it will not be called but the remaining repetitions will be decremented. When using SchedulableMBean the SchedulableMBeanMethod must also be specified.
- **SchedulableMBeanMethod**: Specifies the operation name to be called on the schedulable MBean. It can optionally be followed by an opening bracket, a comma separated list of parameter keywords, and a closing bracket. The supported parameter keywords include:
 - `NOTIFICATION` which will be replaced by the timers notification instance (`javax.management.Notification`)
 - `DATE` which will be replaced by the date of the notification call (`java.util.Date`)
 - `REPETITIONS` which will be replaced by the number of remaining repetitions (`long`)
 - `SCHEDULER_NAME` which will be replaced by the `ObjectName` of the Scheduler (`javax.management.ObjectName`)
 - Any fully qualified class name which the Scheduler will set to null. This allows

A given `Scheduler` instance only support a single schedulable instance. If you need to configure multiple scheduled events you would use multiple `Scheduler` instances, each with a unique `ObjectName`. 10-2 gives an example of configuring a `Scheduler` to call a `Schedulable` implementation as well as a configuration for calling a MBean.

LISTING 10-2. An example Scheduler jboss-service descriptor

```
<server>

  <classpath codebase="lib" archives="scheduler-plugin.jar,scheduler-plugin-
example.jar"/>

  <!-- An example Scheduler configuration to call a custom Schedulable
      interface implementation
  -->
  <mbean code="org.jboss.varia.scheduler.Scheduler"
    name="jboss.util:service=Scheduler,type=SchedulableExample">
    <attribute name="StartAtStartup">true</attribute>
    <attribute
name="SchedulableClass">org.jboss.varia.scheduler.example.SchedulableExample
    </attribute>
    <attribute name="SchedulableArguments">Schedulabe Test,12345</attribute>
    <attribute name="SchedulableArgumentTypes">java.lang.String,int</attribute>
    <attribute name="InitialStartDate">0</attribute>
    <attribute name="SchedulePeriod">10000</attribute>
    <attribute name="InitialRepetitions">-1</attribute>
  </mbean>

  <!-- An example MBean and a Scheduler configuration to invoke its
      hit(Notification, Date, long, ObjectName, String) operation
  -->
  <mbean code="org.jboss.varia.scheduler.example.SchedulableMBeanExample">
```

```
    name=" jboss.util:name=SchedulableMBeanExample">
  </mbean>
<mbean code="org.jboss.varia.scheduler.Scheduler"
  name=" jboss.util:service=Scheduler,type=SchedulableMBeanExample">
  <attribute name="StartAtStartup">true</attribute>
  <attribute name="SchedulableMBean">jboss.util:name=SchedulableMBeanExample</
attribute>
  <attribute name="SchedulableMBeanMethod">hit(NOTIFICATION, DATE, REPETITIONS,
SCHEDULER_NAME, java.lang.String)</attribute>
  <attribute name="InitialStartDate">NOW</attribute>
  <attribute name="SchedulePeriod">10000</attribute>
  <attribute name="InitialRepetitions">10</attribute>
</mbean>
</server>
```


The JBoss Group and Our LGPL License

About The JBoss Group

JBoss Group LLC, is an Atlanta-based professional services company, created by Marc Fleury, founder and lead developer of the JBoss J2EE-based Open Source web application server. JBoss Group brings together core JBoss developers to provide services such as training, support and consulting, as well as management of the JBoss software and services affiliate programs. These commercial activities subsidize the development of the free core JBoss server. For additional information on the JBoss Group see the JBoss site <http://www.jboss.org/services/services.jsp>.

The GNU Lesser General Public License (LGPL)

The JBoss source code is licensed under the LGPL (see <http://www.gnu.org/copyleft/lesser.txt>). This includes all code in the org.jboss.* package namespace. Listing 11-1 gives the complete text of the LGPL license.

LISTING 11-1. The GNU lesser general public license text

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a

restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of non-free programs, enabling many more people to use the whole GNUfree software. For example, permission to use the GNU C Library in operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the

copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on

the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any

patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or
License as published by the Free Software Foundation; either
under the terms of the GNU Lesser General Public
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the
library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

This appendix provides the common JBoss DTDs for reference. Note that all DTDs used by JBoss are included in the <jboss-dist>/docs/dtds directory so check there for the latest version for a given release.

The jboss_3_0.dtd

```
<?xml version='1.0' encoding='UTF-8' ?>

<!--Generated by XML Authority-->

<!--
This is the XML DTD for the JBoss 3.0 EJB deployment descriptor.
The DOCTYPE is:
  <!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 3.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_0.dtd">

$Id: jboss_3_0.dtd,v 1.5.2.4 2002/08/24 00:39:47 starksm Exp $
$Revision: 1.5.2.4 $

Overview of the architecture of jboss.xml

<jboss>

  <enforce-ejb-restrictions />
  <security-domain />
  <unauthenticated-principal />

  <enterprise-beans>
```

```
<entity>
  <ejb-name />
  <jndi-name />
  <local-jndi-name />
  <read-only>
  <home-invoker>
  <bean-invoker>
  <configuration-name>
  <security-proxy>
  <ejb-ref>
    <resource-ref>
      <res-ref-name />
      <resource-name />
    </resource-ref>
    <resource-env-ref>
      <clustered />
      <cluster-config>
    </entity>

  <session>
    <ejb-name />
    <jndi-name />
    <local-jndi-name />
    <home-invoker>
    <bean-invoker>
    <configuration-name>
    <security-proxy>
    <ejb-ref>
      <resource-ref>
        <res-ref-name />
        <resource-name />
      </resource-ref>
      <resource-env-ref>
        <clustered />
        <cluster-config>
    </session>

  <message-driven>
    <ejb-name>
    <destination-jndi-name>
    <mdb-user>
    <mdb-passwd>
    <mdb-client-id>
    <mdb-subscription-id>
    <configuration-name>
    <security-proxy>
    <ejb-ref>
      <resource-ref>
      <resource-env-ref>
    </message-driven>

  </enterprise-beans>

<resource-managers>
```

```
<resource-manager>
  <res-name />
  <res-jndi-name />
</resource-manager>

<resource-manager>
  <res-name />
  <res-url />
</resource-manager>

</resource-managers>

<container-configurations>

  <container-configuration>
    <container-name />
    <container-invoker />
    <container-interceptors />
    <client-interceptors />
    <instance-pool />
    <instance-cache />
    <persistence-manager />
    <transaction-manager />
    <web-class-loader />
    <locking-policy />
    <container-invoker-conf />
    <container-cache-conf />
    <container-pool-conf />
    <commit-option />
    <optiond-refresh-rate />
    <security-domain/>
  </container-configuration>

</container-configurations>

</jboss>
-->
<!--
The jboss element is the root element of the jboss.xml file. It
contains all the information used by jboss but not described in the
ejb-jar.xml file. All of it is optional.

1- the application assembler can define custom container configurations
for the beans. Standard configurations are provided in standardjboss.xml
2- the deployer can override the jndi names under which the beans are
deployed
3- the deployer can specify runtime jndi names for resource managers.

-->
<!ELEMENT jboss (enforce-ejb-restrictions? , security-domain? , unauthenticated-
principal? , enterprise-beans? , resource-managers? , container-configurations?)>

<!--
```

The enforce-ejb-restrictions element tells the container to enforce ejb1.1 restrictions

It must be one of the following :

```
<enforce-ejb-restrictions>true</enforce-ejb-restrictions>
<enforce-ejb-restrictions>false</enforce-ejb-restrictions>
```

Used in: jboss

-->

```
<!ELEMENT enforce-ejb-restrictions (#PCDATA)>
```

<!--

The security-domain element specifies the JNDI name of the security manager that implements the EJBSecurityManager and RealmMapping for the domain. When specified at the jboss level it specifies the security domain for all j2ee components in the deployment unit.

One can override the global security-domain at the container level using the security-domain element at the container-configuration level.

Used in: jboss, container-configuration

-->

```
<!ELEMENT security-domain (#PCDATA)>
```

<!--

The unauthenticated-principal element specifies the name of the principal that will be returned by the EJBContext.getCallerPrincipal() method if there is no authenticated user. This Principal has no roles or privaledges to call any other beans.

-->

```
<!ELEMENT unauthenticated-principal (#PCDATA)>
```

<!--

The enterprise-beans element contains additional information about the beans. These informations, such as jndi names, resource managers and container configurations, are specific to jboss and not described in ejb-jar.xml.

jboss will provide a standard behaviour if no enterprise-beans element is found, see container-configurations, jndi-name and resource-managers for defaults.

Used in: jboss

-->

```
<!ELEMENT enterprise-beans (session | entity | message-driven)+>
```

<!--

The entity element holds information specific to jboss and not declared in ejb-jar.xml about an entity bean, such as jndi name, container configuration, and resource managers. (see tags for details)

The bean should already be declared in ejb-jar.xml, with the same ejb-name.

Used in: enterprise-beans

-->

```
<!ELEMENT entity (ejb-name , jndi-name? , local-jndi-name? , read-only? , home-
invoker? , bean-invoker? ,
configuration-name? , security-proxy? , ejb-ref* , resource-ref* , resource-env-
ref* ,
clustered? , cluster-config?)>

<!--
The session element holds information specific to jboss and not declared
in ejb-jar.xml about a session bean, such as jndi name, container
configuration, and resource managers. (see tags for details)
The bean should already be declared in ejb-jar.xml, with the same
ejb-name.

Used in: enterprise-beans
-->
<!ELEMENT session (ejb-name , jndi-name? , local-jndi-name? , read-only? , home-
invoker? ,
configuration-name? , security-proxy? , ejb-ref* , resource-ref* , resource-env-
ref* ,
clustered? , cluster-config?)>

<!--
The message-driven element holds information specific to jboss and not declared
in ejb-jar.xml about a message-driven bean, such as container
configuration and resources.
The bean should already be declared in ejb-jar.xml, with the same
ejb-name.

Used in: enterprise-beans
-->
<!ELEMENT message-driven (ejb-name , destination-jndi-name , mdb-user? , mdb-
passwd? , mdb-client-id? ,
mdb-subscription-id? , configuration-name? , security-proxy? , ejb-ref* ,
resource-ref* ,
resource-env-ref*)>

<!--
The ejb-name element gives the name of the bean, it must correspond to
an ejb-name element in ejb-jar.xml

Used in: entity, session, and message-driven
-->
<!ELEMENT ejb-name (#PCDATA)>

<!--
The jndi-name element gives the actual jndi name under which
the bean will be deployed when used in the entity, session and
message-driven elements. If it is not provided jboss will assume
"jndi-name" = "ejb-name"

When used in the ejb-ref, resource-ref, resource-env-ref elements
this specifies the jndi name to which the reference should link.

Used in: entity, session and message-driven
ejb-ref, resource-ref, resource-env-ref
```

```

-->
<!ELEMENT jndi-name (#PCDATA)>

<!--
   The JNDI name under with the local home interface should be bound

      Used in: entity and session
-->
<!ELEMENT local-jndi-name (#PCDATA)>

<!--
   The read-only element flags an entity bean as read only.
   The bean will never be ejbStored. Defaults to false.
   It must be one of the following :
      <read-only>true</read-only>
      <read-only>false</read-only>

      Used in: entity
-->
<!ELEMENT read-only (#PCDATA)>

<!--
   The home-invoker and bean-invoker elements define the Invoker MBean to use
   for Home Proxies and Bean Proxies. When specified at the container configuration
   level this specifies the default invoker for all beans using the container
   configuration.

      Used in: container-configuration, entity and session
-->
<!ELEMENT home-invoker (#PCDATA)>

<!ELEMENT bean-invoker (#PCDATA)>

<!--
   The configuration-name element gives the name of the container
   configuration for this bean. It must match one of the container-name
   tags in the container-configurations section, or one of the standard
   configurations. If no element is provided, jboss will automatically use the
   right standard configuration, see container-configurations.

   Note: unlike earlier releases, this element may not be specified as an
         empty element to achieve the same effect as not specifying the element.

      Used in: entity, session, and message-driven
-->
<!ELEMENT configuration-name (#PCDATA)>

<!ELEMENT destination-jndi-name (#PCDATA)>

<!ELEMENT mdb-user (#PCDATA)>

<!ELEMENT mdb-passwd (#PCDATA)>

<!ELEMENT mdb-client-id (#PCDATA)>
```

```
<!ELEMENT mdb-subscription-id (#PCDATA)>

<!-- The security-proxy gives the class name of the security proxy implementation.
     This may be an instance of org.jboss.security.SecurityProxy, or an
     just an object that implements methods in the home or remote interface
     of an EJB without implementing any common interface.

     Used in: entity, session, and message-driven
-->
<!ELEMENT security-proxy (#PCDATA)>

<!--
     The ejb-ref element is used to give the jndi-name of an external
     ejb reference. In the case of an external ejb reference, you don't
     provide a ejb-link element in ejb-jar.xml, but you provide a jndi-name
     in jboss.xml

     Used in: entity, session, and message-driven
-->
<!ELEMENT ejb-ref (ejb-ref-name , jndi-name)>

<!--
     The ejb-ref-name element is the name of the ejb reference as given in
     ejb-jar.xml.

     Used in: ejb-ref
-->
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
     The resource-env-ref element gives a mapping between the "code name"
     of a env resource (res-ref-name, provided by the Bean Developer) and
     its deployed JNDI name.

     Used in: session, entity, message-driven
-->
<!ELEMENT resource-env-ref (resource-env-ref-name , jndi-name)>

<!--
     The resource-env-ref-name element gives the "code name" of a resource. It is
     provided by the Bean Developer.

     Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--
     The clustered element indicates if this bean will run in a cluster of JBoss
     instances.

     It is provided by the deployer. If not, jboss will assume clustered = False
     Possible values: "True", "False" (default)

     Used in: entity and session
-->
<!ELEMENT clustered (#PCDATA)>
```

```

<!--
The cluster-config element allows to specify cluster specific settings.
WARNING: session-state-manager-jndi-name is only for SFSB.

Used in: session, entity
-->
<!ELEMENT cluster-config (partition-name? , home-load-balance-policy? , bean-
load-balance-policy? ,
session-state-manager-jndi-name?)>

<!--
The partition-name element indicates the name of the HAPartition to be used
by the container to exchange clustering information. This is a name and *not*
a JNDI name. Given name will be prefixed by "/HASessionState/" by the container
to get
the actual JNDI name of the HAPartition. If not, jboss will assume partition-
name = "DefaultPartition".

Used in: entity and session (in clustered-config element)
-->
<!ELEMENT partition-name (#PCDATA)>

<!--
The home-load-balance-policy element indicates the java class name to be used
to load balance calls in the home proxy.
If not, jboss will assume home-load-balance-policy =
"org.jboss.ha.framework.interfaces.RoundRobin".

Used in: entity and session (in clustered-config element)
-->
<!ELEMENT home-load-balance-policy (#PCDATA)>

<!--
The bean-load-balance-policy element indicates the java class name to be used
to load balance calls in the bean proxy.
If not, jboss will assume :
- for EB and SFSB : bean-load-balance-policy =
"org.jboss.ha.framework.interfaces.RoundRobin"
- for SLSB           : bean-load-balance-policy =
"org.jboss.ha.framework.interfaces.RoundRobin"

Used in: entity and session (in clustered-config element)
-->
<!ELEMENT bean-load-balance-policy (#PCDATA)>

<!--
The session-state-manager-jndi-name element indicates the name of the
HASessionState to be used
by the container as a backend for state session management in the cluster.
This *is* a JNDI name (not like the partition-name element).
If not, jboss will assume partition-name = "/HASessionState/Default".

Used in: session (in clustered-config element)
-->

```

```
<!ELEMENT session-state-manager-jndi-name (#PCDATA)>

<!--
The resource-ref element gives a mapping between the "code name"
of a resource (res-ref-name, provided by the Bean Developer) and
its "xml name" (resource-name, provided by the Application Assembler).
If no resource-ref is provided, jboss will assume that
"xml-name" = "code name"

See resource-managers.

Used in: entity, session, and message-driven
-->
<!ELEMENT resource-ref (res-ref-name , (resource-name | jndi-name | res-url))>

<!--
The res-ref-name element gives the "code name" of a resource. It is
provided by the Bean Developer. See resource-managers for the actual
configuration of the resource.

Used in: resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>

<!--
The resource-name element gives the "xml name" of the resource. It is
provided by the Application Assembler. See resource-managers for the
actual configuration of the resource.

Used in: resource-ref
-->
<!ELEMENT resource-name (#PCDATA)>

<!--
The resource-managers element is used to declare resource managers.

A resource has 3 names:
- the "code name" is the name used in the code of the bean, supplied by
  the Bean Developer in the resource-ref section of the ejb-jar.xml file

- the "xml name" is an intermediary name used by the Application Assembler
  to identify resources in the XML file.

- the "runtime jndi name" is the actual jndi-name or url of the deployed
  resource, it is supplied by the Deployer.

The mapping between the "code name" and the "xml name" is given
in the resource-ref section for the bean. If not, jboss will assume that
"xml name" = "code name".

The mapping between the "xml name" and the "runtime jndi name" is given in
a resource-manager section. If not, and if the datasource is of type
javax.sql.DataSource, jboss will look for a javax.sql.DataSource in the jndi
tree.
```

```

Used in: jboss
--->
<!ELEMENT resource-managers (resource-manager*)>

<!--
The resource-manager element is used to provide a mapping between the
"xml name" of a resource (res-name) and its "runtime jndi name"
(res-jndi-name or res-url according to the type of the resource).
If it is not provided, and if the type of the resource is
javax.sql.DataSource, jboss will look for a javax.sql.DataSource in the
jndi tree.

See resource-managers.

Used in: resource-managers
--->
<!ELEMENT resource-manager (res-name , (res-jndi-name | res-url))>

<!--
The res-name element gives the "xml name" of a resource, it is provided
by the Application Assembler. See resource-managers.

Used in: resource-manager
--->
<!ELEMENT res-name (#PCDATA)>

<!--
The res-jndi-name element is the "deployed jndi name" of a resource, it
is provided by the Deployer. See resource-managers.

Used in: resource-manager
--->
<!ELEMENT res-jndi-name (#PCDATA)>

<!--
The res-url element is the "runtime jndi name" as a url of the resource.
It is provided by the Deployer. See resource-managers.

Used in: resource-manager
--->
<!ELEMENT res-url (#PCDATA)>

<!--
The container-configurations element declares the different possible
container configurations that the beans can use. standardjboss.xml
provides 15 standard configurations with the following container-names:
- Standard CMP 2.x EntityBean
- Standard CMP EntityBean
- Clustered CMP 2.x EntityBean
- Clustered CMP EntityBean
- IIOP CMP 2.x EntityBean
- IIOP CMP EntityBean
- jdk 1.2.2 CMP EntityBean
- Standard Stateless SessionBean
- Clustered Stateless SessionBean

```

- IIOP Stateless SessionBean
- jdk 1.2.2 Stateless SessionBean
- Standard Stateful SessionBean
- Clustered Stateful SessionBean
- IIOP Stateful SessionBean
- jdk 1.2.2 Stateful SessionBean
- Standard BMP EntityBean
- Clustered BMP EntityBean
- IIOP BMP EntityBean
- jdk 1.2.2 BMP EntityBean
- Standard message Driven Bean

The standard configurations will automatically be used if no custom configuration is specified.

The jdk 1.2.2 configurations are defined for backwards compatibility.

The application assembler can define advanced custom configurations here.

Used in: jboss

-->

```
<!ELEMENT container-configurations (container-configuration*)>
```

<!--

The container-configuration element describes a configuration for the container.

The different plugins to use are declared here, as well as their configurations. The configuration-class attribute is no longer used.

Used in: container-configurations

-->

```
<!ELEMENT container-configuration (container-name , call-logging? , container-
invoker? ,
home-invoker? , bean-invoker? , container-interceptors? , client-interceptors? ,
instance-pool? ,
instance-cache? , persistence-manager? , transaction-manager? , web-class-loader?
, locking-policy? ,
container-invoker-conf? , container-cache-conf? , container-pool-conf? , commit-
option? ,
optiond-refresh-rate? , security-domain?, cluster-config?)>
```

<!-- The extends attribute gives the container-name value of the configuration the container-configuration is extending. This allows one to specify an extension configuration without having to reiterate all of the other duplicate configuration info.

```
<container-configuration extends="Standard Stateful SessionBean">
    <container-name>Secure Stateless SessionBean</container-name>
    <security-domain>java:/jaas/other</security-domain>
</container-configuration>
```

-->

```
<!ATTLIST container-configuration extends CDATA  #IMPLIED>
```

```

<!--
The container-name element gives the name of the configuration being
defined. Beans may refer to this name in their configuration-name tag.

Used in: container-configuration
-->
<!ELEMENT container-name (#PCDATA)>

<!--
The call-logging element tells if the container must log every method
invocation for this bean or not. Its value must be true or false.

Used in: container-configuration
-->
<!ELEMENT call-logging (#PCDATA)>

<!--
The container-invoker element gives the class name of the container
invoker jboss must use for in this configuration. This class must
implement the org.jboss.ejb.ContainerInvoker interface. The default is
org.jboss.proxy.ejb.ProxyFactory for entity and session beans and
org.jboss.ejb.plugins.jms.JMSContainerInvoker for message driven beans.
Containers supporting clustering use org.jboss.proxy.ejb.ProxyFactoryHA.

Used in: container-configuration
-->
<!ELEMENT container-invoker (#PCDATA)>

<!--
The container-interceptors element gives the chain of Interceptors
(instances of org.jboss.ejb.Interceptor) that are associated with the
container.
The declared order of the interceptor elements corresponds to the order of the
interceptor chain.

Used in: container-configuration
-->
<!ELEMENT container-interceptors (interceptor+)>

<!--
The client-interceptors defines the home and bean client side interceptor chain

Used in: container-configuration
-->
<!ELEMENT client-interceptors (home , bean)>

<!--
The home element gives the chain of interceptors
(instances of org.jboss.proxy.Interceptor) that are associated with the home
proxy and operate in the client VM. The declared order of the interceptor
elements corresponds to the order of the interceptor chain.

Used in: client-interceptors
-->
<!ELEMENT home (interceptor+)>

```

```
<!--
The bean element gives the chain of interceptors
instances of org.jboss.proxy.Interceptor) that are associated with the remote
proxy and operate in the client VM. The declared order of the interceptor
elements corresponds to the order of the interceptor chain.

Used in: client-interceptors
-->
<!ELEMENT bean (interceptor+)>

<!--
The interceptor element specifies an instance of org.jboss.ejb.Interceptor
that is to be added to the container interceptor stack.

Used in: container-interceptors
-->
<!ELEMENT interceptor (#PCDATA)>

<!--
The transaction attribute is used to indicate what type of container its
interceptor applies to. It is an enumerated value that can take on one of: Bean,
Container or Both. A value of Bean indicates that the interceptor should only be
added to a container for bean-managed transaction.
A value of Container indicates that the interceptor should only be added to a
container for container-managed transactions.
A value of Both indicates that the interceptor should be added to all
containers. This is the default value if the transaction attribute is not
explicitly given.
-->
<!ATTLIST interceptor transaction      (Bean | Container | Both )  "Both">

<!--
The metricsEnabled attributes is used to indicate if the interceptor
should only be included when the org.jboss.ejb.ContainerFactory metricsEnabled
flag is set to true. The allowed values are true and false with false being the
default if metricsEnabled is not explicitly given.
-->
<!ATTLIST interceptor metricsEnabled  (true | false )  "false">

<!--
The instance-pool element gives the class name of the instance pool
jboss must use for in this configuration. This class must implement
the org.jboss.ejb.InstancePool interface. The defaults are:
- org.jboss.ejb.plugins.EntityInstancePool for entity beans
- org.jboss.ejb.plugins.StatelessSessionInstancePool for stateless
session beans.
- no pool is used for stateful session beans

Used in: container-configuration
-->
<!ELEMENT instance-pool (#PCDATA)>

<!--
The instance-cache element gives the class name of the instance cache
```

jboss must use for in this configuration. This class must implement the org.jboss.ejb.InstanceCache interface. The defaults are:

- org.jboss.ejb.plugins.NoPassivationEntityInstanceCache for entity beans
- org.jboss.ejb.plugins.NoPassivationStatefulSessionInstanceCache for stateful session beans.
- no cache is used for stateless session beans

Used in: container-configuration

```
-->
<!ELEMENT instance-cache (#PCDATA)>
```

<!--

The persistence-manager element gives the class name of the persistence manager / persistence store jboss must use for in this configuration. This class must implement:

- org.jboss.ejb.EntityPersistenceStore for CMP Entity Beans (default is org.jboss.ejb.plugins.jaws.JAWSPersistenceManager)
- org.jboss.ejb.EntityPersistenceManager for BMP entity beans (default is org.jboss.ejb.plugins.BMPPersistenceManager)
- org.jboss.ejb.StatefulSessionPersistenceManager for stateless session beans.
- no persistence-manager is used for stateless session beans

Used in: container-configuration

```
-->
<!ELEMENT persistence-manager (#PCDATA)>
```

<!--

The locking-policy element gives the class name of the EJB lock implementation JBoss must use for in this configuration. This class must implement the org.jboss.ejb.BeanLock interface. The default is org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock.

Used in: container-configuration

```
-->
<!ELEMENT locking-policy (#PCDATA)>
```

<!--

The transaction-manager element gives the class name of the transaction manager jboss must use for in this configuration. This class must implement the javax.transaction.TransactionManager interface. The default is org.jboss.tm.TxManager.

Used in: container-configuration

```
-->
<!ELEMENT transaction-manager (#PCDATA)>
```

<!--

The web-class-loader element gives the class name of the web classloader jboss must use for in this configuration. This class must be a subclass of org.jboss.web.WebClassLoader. The default is org.jboss.web.WebClassLoader.

Used in: container-configuration

```
-->
<!ELEMENT web-class-loader (#PCDATA)>
```

```
<!--
The container-invoker-conf element holds configuration data for the
container invoker.
jboss does not read directly the subtree for this element: instead,
it is passed to the container invoker instance (if it implements
org.jboss.metadata.XmlLoadable) for it to load its parameters.

The Optimized tag described here only relates to the default container
invokers, ProxyFactory, ProxyFactoryHA and JMSContainerInvoker.

Used in: container-configuration
-->
<!ELEMENT container-invoker-conf (JMSProviderAdapterJNDI? ,
ServerSessionPoolFactoryJNDI? , MaximumSize? , MaxMessages? , MDBConfig?)>

<!--
Used in: container-invoker-conf for JMSContainerInvoker
-->
<!ELEMENT JMSProviderAdapterJNDI (#PCDATA)>

<!--
Used in: container-invoker-conf for JMSContainerInvoker
-->
<!ELEMENT ServerSessionPoolFactoryJNDI (#PCDATA)>

<!--
Used in: container-invoker-conf for JMSContainerInvoker
-->
<!ELEMENT MaximumSize (#PCDATA)>

<!--
Used in: container-invoker-conf for JMSContainerInvoker
-->
<!ELEMENT MDBConfig (ReconnectIntervalSec , DLQConfig?)>

<!--
Used in: MDBConfig
-->
<!ELEMENT ReconnectIntervalSec (#PCDATA)>

<!--
Used in: MDBConfig
-->
<!ELEMENT DLQConfig (DestinationQueue , MaxTimesRedelivered , TimeToLive)>

<!--
Used in: DLQConfig
-->
<!ELEMENT DestinationQueue (#PCDATA)>

<!--
Used in: DLQConfig
-->
<!ELEMENT MaxTimesRedelivered (#PCDATA)>
```

```

<!--
Used in: DLQConfig
-->
<!ELEMENT TimeToLive (#PCDATA)>

<!--
The container-cache-conf element holds dynamic configuration data
for the instance cache.
jboss does not read directly the subtree for this element: instead,
it is passed to the instance cache instance (if it implements
org.jboss.metadata.XmlLoadable) for it to load its parameters.

Used in: container-configuration
-->
<!ELEMENT container-cache-conf (cache-policy? , cache-policy-conf?)>

<!--
The implementation class for the cache policy, which controls
when instances will be passivated, etc.

Used in: container-cache-conf
-->
<!ELEMENT cache-policy (#PCDATA)>

<!--
The configuration settings for the selected cache policy. This
is currently only valid for the LRU cache.
When the cache is the LRU one for the stateful container, the elements
remover-period and max-bean-life specifies the period of the remover
task that removes stateful beans (that normally have been passivated)
that have age greater than the specified max-bean-life element.

Used in: container-cache-conf (when cache-policy is the LRU cache)
-->
<!ELEMENT cache-policy-conf (min-capacity , max-capacity , remover-period? , max-
bean-life? , overager-period , max-bean-age , resizer-period , max-cache-miss-
period , min-cache-miss-period , cache-load-factor)>

<!--
The minimum capacity of this cache
-->
<!ELEMENT min-capacity (#PCDATA)>

<!--
The maximum capacity of this cache
-->
<!ELEMENT max-capacity (#PCDATA)>

<!--
The period of the overager's runs
-->
<!ELEMENT overager-period (#PCDATA)>

<!--

```

```
    The period of the remover's runs
-->
<!ELEMENT remover-period (#PCDATA)>

<!--
The max-bean-life specifies the period of the remover
task that removes stateful beans (that normally have been passivated)
that have age greater than the specified max-bean-life element.
-->
<!ELEMENT max-bean-life (#PCDATA)>

<!--
The period of the resizer's runs
-->
<!ELEMENT resizer-period (#PCDATA)>

<!--
The age after which a bean is automatically passivated
-->
<!ELEMENT max-bean-age (#PCDATA)>

<!--
Shrink cache capacity if there is a cache miss every or more
this member's value
-->
<!ELEMENT max-cache-miss-period (#PCDATA)>

<!--
Enlarge cache capacity if there is a cache miss every or less
this member's value
-->
<!ELEMENT min-cache-miss-period (#PCDATA)>

<!--
The resizer will always try to keep the cache capacity so that
the cache is this member's value loaded of cached objects
-->
<!ELEMENT cache-load-factor (#PCDATA)>

<!--
The container-pool-conf element holds configuration data for the
instance pool.
jboss does not read directly the subtree for this element: instead,
it is passed to the instance pool instance (if it implements
org.jboss.metadata.XmlLoadable) for it to load its parameters.

The default instance pools, EntityInstancePool and
StatelessSessionInstancePool, both accept the following configuration.

Used in: container-configuration
-->
<!ELEMENT container-pool-conf (MaximumSize , feeder-policy , feeder-policy-conf)>

<!--
The capacity of the Pool. The pool feeder will feed the pool with new
```

instances, the pool size being limited by this value. For pools where reclaim is possible, the pool will also be feed when the instance is free to be reused.

This is not an hard limit, if instances are needed when the pool is at its MaximumSize, new instances will be created following the demand.

Used in: container-pool-conf

-->

```
<!ELEMENT MaximumSize (#PCDATA)>
```

<!--

This element is only valid if the instance pool is a subclass of AbstractInstancePool.

The feeder-policy element gives the Class that implements InstancePoolFeeder and is responsible to feed the pool with new instances of bean.

If not present, no thread are started and the pool will have a size of 1.

TimedInstancePoolFeeder is the first implementation available.

Used in: container-pool-conf for AbstractInstancePool subclasses

-->

```
<!ELEMENT feeder-policy (#PCDATA)>
```

<!--

This element describes properties that the InstancePoolFeeder implementation will read to configure itself (XmlLoadable).

Note: the 3 attributes are hardcoded here for TimedInstancePoolFeeder.

Used in: container-pool-conf for InstancePoolFeeder implementations

-->

```
<!ELEMENT feeder-policy-conf (increment, period)>
```

<!--

The pool feeder will feed the pool with this number of new instances at a regular period.

Used in: feeder-policy-conf

-->

```
<!ELEMENT increment (#PCDATA)>
```

<!--

The interval of time (in milliseconds) the pool feeder look if the pool has come to its maximum size (capacity) and if not, will feed it with a particular number of new instances (increment).

Used in: feeder-policy-conf

-->

```
<!ELEMENT period (#PCDATA)>
```

<!--

This option is only used for entity container configurations.

The commit-option element tells the container which option to use for transactions.

Its value must be A, B C, or D.

- option A: the entity instance has exclusive access to the database. The instance stays ready after a transaction.
- option B: the entity instance does not have exclusive access to the database. The state is loaded before the next transaction.
- option C: same as B, except the container does not keep the instance after commit:
 - a passivate is immediately performed after the commit.
- option D: a lazy update. default is every 30 secs.
 - can be updated with <optiond-refresh-rate>

See ejb1.1 specification for details (p118).

Used in: container-configuration
-->
<!ELEMENT commit-option (#PCDATA)>

<!--
This element is used to specify the refresh rate of commit option d
-->
<!ELEMENT optiond-refresh-rate (#PCDATA)>

The jbosscmp-jdbc_3_0.dtd DTD

```
<?xml version='1.0' encoding='UTF-8' ?>

<!--
This is the XML DTD for the jbosscmp-jdbc deployment descriptor.
<!DOCTYPE jbosscmp-jdbc PUBLIC
  "-//JBoss//DTD JBOSSCMP-JDBC 3.0//EN"
  "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_0.dtd">
-->

<!--
The root element of JBossCMP-JDBC configuration files.
-->
<!ELEMENT jbosscmp-jdbc (defaults?, enterprise-beans?, relationships?,
  dependent-value-classes?, type-mappings?)>

<!--
The optional defaults elements contains the default values for
entities, and relationships.
-->
<!ELEMENT defaults ((datasource, datasource-mapping)?, create-table?,
  remove-table?, read-only?, read-time-out?, row-locking?,
```

```

pk-constraint?, fk-constraint?, preferred-relation-mapping?,
read-ahead?, list-cache-max?, fetch-size?)>

<!--
The optional datasource element contains the jndi-name used to lookup
the datasource. All database connections used by an entity or relation table are
obtained from the datasource.
-->
<!ELEMENT datasource (#PCDATA)>

<!--
The optional datasource-mapping element contains the name of the type mapping
that will be used for this datasource.
-->
<!ELEMENT datasource-mapping (#PCDATA)>

<!--
Should the persistence manager attempt to create tables if they are
not present?

The create-table element must be one of the two following:
<create-table>true</create-table>
<create-table>false</create-table>
-->
<!ELEMENT create-table (#PCDATA)>

<!--
Should the persistence manager attempt to remove tables during shutdown?

The remove-table element must be one of the two following:
<remove-table>true</remove-table>
<remove-table>false</remove-table>
-->
<!ELEMENT remove-table (#PCDATA)>

<!--
Is the entity or cmp-field read-only?

The read-only element must be one of the two following:
<read-only>true</read-only>
<read-only>false</read-only>
-->
<!ELEMENT read-only (#PCDATA)>

<!--
Specifies the amount of time that a read-only field is considered
valid (milliseconds).
-->
<!ELEMENT read-time-out (#PCDATA)>
```

```
<!--
Should select statements the SELECT ... FOR UPDATE syntax?

The row-locking element must be one of the two following:
<row-locking>true</row-locking>
<row-locking>false</row-locking>
-->
<!ELEMENT row-locking (#PCDATA)>

<!--
Should a foreign key constraint be added for this relationship role?

The fk-constraint element must be one of the two following:
<fk-constraint>true</fk-constraint>
<fk-constraint>false</fk-constraint>
-->
<!ELEMENT fk-constraint (#PCDATA)>

<!--
Should a primary key constraint be added when creating tables?

The pk-constraint element must be one of the two following:
<pk-constraint>true</pk-constraint>
<pk-constraint>false</pk-constraint>
-->
<!ELEMENT pk-constraint (#PCDATA)>

<!--
Specifies the preferred mapping style for relationships.

The preferred-relation-mapping element must be one of the two following:
<preferred-relation-mapping>foreign-key</preferred-relation-mapping>
<preferred-relation-mapping>relation-table</preferred-relation-mapping>
-->
<!ELEMENT preferred-relation-mapping (#PCDATA)>

<!--
Specifies the read ahead strategy.

<read-ahead>
  <strategy>on-load</strategy>
  <page-size>255</page-size>
  <eager-load-group>*</eager-load-group>
</read-ahead>
-->
<!ELEMENT read-ahead (strategy, page-size?, eager-load-group?)>

<!--
Specifies the strategy used to read-ahead data in queries.
```

The strategy element must be one of the two following:

```

<strategy>none</strategy>
<strategy>on-load</strategy>
<strategy>on-find</strategy>
-->
<!ELEMENT strategy (#PCDATA)>

<!--
Specifies the number of entities that will be read in a single
read-ahead load query.
-->
<!ELEMENT page-size (#PCDATA)>

<!--
Specifies the number of simultaneous queries that can be tracked by
the cache for an entity.
-->
<!ELEMENT list-cache-max (#PCDATA)>

<!--
Specifies the number of entities to read in one round-trip to
the underlying datastore.
-->
<!ELEMENT fetch-size (#PCDATA)>

<!--
The enterprise-beans element contains the entity elements that will
be configured.
-->
<!ELEMENT enterprise-beans (entity+)>

<!--
The entity element contains the configuration of an entity
-->
<!ELEMENT entity (ejb-name, (datasource, datasource-mapping)?, create-table?,
remove-table?, read-only?, read-time-out?, row-locking?,
pk-constraint?, read-ahead?, list-cache-max?, fetch-size?, table-name?,
cmp-field*, load-groups?, eager-load-group?, lazy-load-groups?,
query*)>

<!--
Name of the entity being configured. This must match an entity declared
in the ejb-jar.xml file.
-->
<!ELEMENT ejb-name (#PCDATA)>

<!--
This is the name of the table that will hold data for this entity.
Each entity instance will be stored in one row of this table.

```

```
-->
<!ELEMENT table-name (#PCDATA)>

<!--
The cmp-field element contains the configuration of a cmp-field.
-->
<!ELEMENT cmp-field (field-name, read-only?, read-time-out?,
                     column-name?, not-null?, ((jdbc-type, sql-type) | (property+))?)>

<!--
Name of the cmp-field being configured. This must match a cmp-field
declared for this entity in the ejb-jar.xml file.
-->
<!ELEMENT field-name (#PCDATA)>

<!--
The name of the column that will hold the data for this field.
-->
<!ELEMENT column-name (#PCDATA)>

<!--
If present the field will not allow a field value.
-->
<!ELEMENT not-null EMPTY>

<!--
This is the JDBC type that is used when setting parameters in a JDBC
PreparedStatement or loading data from a JDBC ResultSet for this
cmp-field. The valid types are defined in java.sql.Types.
-->
<!ELEMENT jdbc-type (#PCDATA)>

<!--
This is the SQL type that is used in create table statements for
this field. Valid sql-types are only limited by your database vendor.
-->
<!ELEMENT sql-type (#PCDATA)>

<!--
The property element contains the configuration of a dependent
value class property of a cmp-field that is the type of a dependent
value class.
-->
<!ELEMENT property (property-name, column-name?, not-null?,
                    (jdbc-type, sql-type)?)>

<!--
```

Name of the property being configured. In a dependent-value-class element this must match a JavaBean property of the class. In a cmp-field element this must match a flattened property of the dependent-value-class field type.

```
-->
<!ELEMENT property-name (#PCDATA)>
```

<!--
Contains the named load groups.-->

```
<!ELEMENT load-groups (load-group+)>
```

<!--
A named group of fields that will be loaded together.-->

```
<!ELEMENT load-group (description?, load-group-name, field-name+)>
```

<!--
Contains the name of a load group.-->

```
<!ELEMENT load-group-name (#PCDATA)>
```

<!--
Contains the name of the load group that will eager loaded for this entity.-->

```
<!ELEMENT eager-load-group (#PCDATA)>
```

<!--
Contains the names of the groups that will be lazy loaded together.-->

```
<!ELEMENT lazy-load-groups (load-group-name+)>
```

<!--
Descriptive text.-->

```
<!ELEMENT description (#PCDATA)>
```

<!--
The query element contains the configuration of a query.-->

```
<!ELEMENT query (description?, query-method,
    (jboss-ql | dynamic-ql | declared-sql)?, read-ahead?)>
```

<!--
The query method that being configured. This must match a query-method declared for this entity in the ejb-jar.xml file.-->

```
<!ELEMENT query-method (method-name, method-params)>
```

```
<!--
The name of the query method that is being configured.
-->
<!ELEMENT method-name (#PCDATA)>

<!--
The method-parameters contains the parameters of the method that is
being configured. Method parameters must be in the same order as the
method and have the same type.
-->
<!ELEMENT method-params (method-param*)>

<!--
The java class of one parameter for a query.

An example follows:
<method-param>java.lang.String</method-param>
-->
<!ELEMENT method-param (#PCDATA)>

<!--
JBossQL query. JBossQL is a superset of EJB-QL.
-->
<!ELEMENT jboss-ql (#PCDATA)>

<!--
Dynamic JBossQL query. The JBossQL is passed to the query and compiled
on the fly.
-->
<!ELEMENT dynamic-ql EMPTY>

<!--
Explicitly declared sql fragments.
-->
<!ELEMENT declared-sql (select?, from?, where?, order?, other?)>

<!--
Delcares what is to be selected. A finder may only have the distinct element.
-->
<!ELEMENT select (distinct?, (ejb-name, field-name?)?, alias?)>

<!--
Delared additional SQL to append to the generated from clause.

Example: <from>, FullAddressEJB as a</from>
-->
<!ELEMENT from (#PCDATA)>
```

```

<!--
If the empty distinct element is present, the SELECT DISTINCT
syntax will be used. This syntax is used by default for ejbSelect
methods that return a java.util.Set.
-->
<!ELEMENT distinct EMPTY>

<!--
Declares the where clause for the query.

Example: <where>TITLE={0} OR ARTIST={0} OR TYPE={0} OR NOTES={0}</where>
-->
<!ELEMENT where (#PCDATA)>

<!--
Declares the order clause for the query.

Example: <order>TITLE</order>
-->
<!ELEMENT order (#PCDATA)>

<!--
Declares the other sql that is appended to the end of a query.

Example: <other>LIMIT 100 OFFSET 200</other>
-->
<!ELEMENT other (#PCDATA)>

<!--
Declare the alias to use for the main select table.

Example: <alias>my_table</alias>
-->
<!ELEMENT alias (#PCDATA)>

<!--
The relationships element contains the ejb-relation elements that will
be configured.
-->
<!ELEMENT relationships (ejb-relation+)>

<!--
The ejb-relation element contains the configuration of an
ejb-relation.
-->
<!ELEMENT ejb-relation (ejb-relation-name, read-only?, read-time-out?,
(foreign-key-mapping | relation-table-mapping)?,
(ejb-relationship-role, ejb-relationship-role)?)>

```

```
<!--
Name of the ejb-relation being configured. This must match an
ejb-relation declared in the ejb-jar.xml file.
-->
<!ELEMENT ejb-relation-name (#PCDATA)>

<!--
Specifies that the ejb-relation should be mapped with foreign-keys.
This mapping is not available for many-to-many relationships.
-->
<!ELEMENT foreign-key-mapping EMPTY>

<!--
Specifies that the ejb-relation should be mapped with a relation-table.
-->
<!ELEMENT relation-table-mapping (table-name?,
    (datasource, datasource-mapping)?, create-table?, remove-table?,
    row-locking?, pk-constraint?)>

<!--
The ejb-relationship-role element contains the configuration of an
ejb-relationship-role.
-->
<!ELEMENT ejb-relationship-role (ejb-relationship-role-name,
    fk-constraint?, key-fields?, read-ahead?)>

<!--
Name of the ejb-relationship-role being configured. This must match
an ejb-relationship-role declared for this ejb-relation in the
ejb-jar.xml file.
-->
<!ELEMENT ejb-relationship-role-name (#PCDATA)>

<!--
Contains the key fields. The interperation of the key fields depends
on the mapping style of the relationship.
-->
<!ELEMENT key-fields (key-field*)>

<!--
The key-field element declared the configuration of a key field.
The field-name element must match the field-name of one of the
primary key fields of the this entity.
-->
<!ELEMENT key-field (field-name,
    ((column-name, (jdbc-type, sql-type)?) | (property*)) )
  )>
```

```

<!--
Contains the known dependent value classes.
-->
<!ELEMENT dependent-value-classes (dependent-value-class*)>

<!--
The dependent-value-class element contains the configuration of a
dependent value class.
-->
<!ELEMENT dependent-value-class (description?, class, property+)>

<!--
Name of the java class to which the dependent value class configuration
applies.
-->
<!ELEMENT class (#PCDATA)>

<!--
The type-mappings element contains the java to sql mappings.
-->
<!ELEMENT type-mappings (type-mapping+)>

<!--
The type-mapping element contains a named java to sql mapping.
This includes both type mapping and function mapping.
-->
<!ELEMENT type-mapping (name, row-locking-template, pk-constraint-template,
    fk-constraint-template, alias-header-prefix, alias-header-suffix,
    alias-max-length, subquery-supported, true-mapping, false-mapping,
    function-mapping*, mapping+)>

<!--
Name of the type-mapping.
-->
<!ELEMENT name (#PCDATA)>

<!--
This is the template used to create a row lock on the selected rows. The
arguments supplied are as follows:
1. Select clause
2. From clasue; the order of the tables is currently not guarenteed
3. Where clause

If row locking is not supported in select statement this element should be
empty. The most common form of row locking is select for update as in the
example that follows:

SELECT ?1 FROM ?2 WHERE ?3 FOR UPDATE
-->
<!ELEMENT row-locking-template (#PCDATA)>

```

<!--

This is the template used to create a primary key constraint in the create table statement. The arguments supplied are as follows:

1. Primary key constraint name; which is always pk_{table-name}
2. Comma separated list of primary key column names

If a primary key constraint clause is not supported in a create table statement this element should be empty. The most common form of a primary key constraint follows:

```
CONSTRAINT ?1 PRIMARY KEY (?2)
-->
<!ELEMENT pk-constraint-template (#PCDATA)>
```

<!--

This is the template used to create a foreign key constraint in sepperate statement. The arguments supplied are as follows:

1. Table name
2. Foreign key constraint name; which is always fk_{table-name}_{cmr-field-name}
3. Comma sepperated list of foreign key column names
4. References table name
5. Comma sepperated list of the referenced primary key column names

If the datasource does not support foreign key constraints this element should be empty. The most common form of a foreign key constraint follows:

```
ALTER TABLE ?1 ADD CONSTRAINT ?2 FOREIGN KEY (?3) REFERENCES ?4 (?5)
-->
<!ELEMENT fk-constraint-template (#PCDATA)>
```

<!--

An alias header is prepended to a generated table alias by the EJB-QL compiler to prevent name collisions. An alias header is constructed as folows:

```
alias-header-prefix + int_counter + alias-header-suffix
-->
<!ELEMENT alias-header-prefix (#PCDATA)>
<!ELEMENT alias-header-suffix (#PCDATA)>
<!ELEMENT alias-max-length (#PCDATA)>
```

<!--

Does this type-mapping support subqueries. Some EJB-QL opperators are mapped to exists subqueries. If subquery is false the EJB-QL compiler will use a left join and is null.

The subquery-supported element must be one of the two following:

```
<create-table>true</create-table>
<create-table>false</create-table>
-->
<!ELEMENT subquery-supported (#PCDATA)>
```

<!--

The true and false mappings are the mappings for true and false in EJB-QL

```

queries.

-->
<!ELEMENT true-mapping (#PCDATA)>
<!ELEMENT false-mapping (#PCDATA)>

<!--
Specifies the mapping from a java type to a jdbc and a sql type.
-->
<!ELEMENT mapping (java-type, jdbc-type, sql-type)>

<!--
Specifies the java class type to be mapped.
-->
<!ELEMENT java-type (#PCDATA)>

<!--
Specifies the mapping from an EJB-QL function to a sql function.
-->
<!ELEMENT function-mapping (function-name, function-sql)>

<!--
The name of the function to be mapped.
-->
<!ELEMENT function-name (#PCDATA)>

<!--
The sql to which the function is mapped. The sql can contain
parameters specified with a question mark followed by the base one
parameter number. For example, function mapping for concat in Oracle
follows:

<function-mapping>
  <function-name>concat</function-name>
  <function-sql>(?1 || ?2)</function-sql>
</function-mapping>
-->
<!ELEMENT function-sql (#PCDATA)>

```

The jboss-web_3_0.dtd DTD

```

<?xml version='1.0' encoding='UTF-8' ?>

<!-- The JBoss specific elements used to integrate the servlet 2.3 web.xml
elements into a JBoss deployment. This version applies to the JBoss 3.x
releases.

DOCTYPE jboss-web
  PUBLIC "-//JBoss//DTD Web Application 2.3//EN"
  "http://www.jboss.org/j2ee/dtd/jboss-web_3_0.dtd"
-->
```

```
<!-- The jboss-web element is the root element.
-->
<!ELEMENT jboss-web (security-domain?, context-root?, virtual-host?,
resource-env-ref*, resource-ref* , ejb-ref*)>

<!-- The context-root element specifies the context root of a web
application. This is normally specified at the ear level using the standard
J2EE application.xml descriptor, but it may be given here for standalone wars.
This should not override the application.xml level specification.
-->
<!ELEMENT context-root (#PCDATA)>

<!-- The security-domain element allows one to specify a module wide
security manager domain. It specifies the JNDI name of the security
manager that implements the org.jboss.security.AuthenticationManager and
org.jboss.security.RealmMapping interfaces for the domain.
-->
<!ELEMENT security-domain (#PCDATA)>

<!-- The virtual-host element allows one to specify which virtual host the war
should be deployed to. Example, to specify that a war should be deployed to the
www.jboss-store.org virtual host add the following virtual-host element:
    <virtual-host>www.jboss-store.org</virtual-host>
-->
<!ELEMENT virtual-host (#PCDATA)>

<!--The resource-env-ref element maps from the servlet ENC relative name
of the resource-env-ref to the deployment environment JNDI name of
the administered object resource.
Example:
<resource-env-ref>
    <resource-env-ref-name>jms/NewsTopic</resource-env-ref-name>
    <jndi-name>topic/NewsTopic</jndi-name>
</resource-env-ref>
-->
<!ELEMENT resource-env-ref (resource-env-ref-name , jndi-name)>

<!-- The resource-env-ref-name specifies the name of the web.xml
resource-env-ref-name element which this mapping applies.
-->
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--The resource-ref element maps from the servlet ENC relative name
of the resource-ref to the deployment environment JNDI name of
the resource manager connection factory.
Example:
<resource-ref>
    <res-ref-name>jdbc/TheDataSource</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
</resource-ref>
-->
<!ELEMENT resource-ref (res-ref-name , jndi-name)>

<!-- The res-ref-name specifies the name of the web.xml res-ref-name element
```

which this mapping applies.

```
-->
<!ELEMENT res-ref-name (#PCDATA)>
```

<!-- The ejb-ref element maps from the servlet ENC relative name of the ejb reference to the deployment environment JNDI name of the bean.

Example:

```
<ejb-ref>
  <ejb-ref-name>ejb/Bean0</ejb-ref-name>
  <jndi-name>deployed/ejbs/Bean0</jndi-name>
</ejb-ref>
```

```
-->
<!ELEMENT ejb-ref (ejb-ref-name , jndi-name)>
```

<!-- The ejb-ref-name element gives the ENC relative name used in the web.xml ejb-ref-name element.

Used in: ejb-ref

```
-->
<!ELEMENT ejb-ref-name (#PCDATA)>
```

<!-- The jndi-name element specifies the JNDI name of the deployed object to which the servlet ENC binding will link to via a JNDI LinkRef.

Used in: resource-ref, resource-env-ref, ejb-ref

```
-->
<!ELEMENT jndi-name (#PCDATA)>
```

The security_config.dtd DTD

```
<?xml version='1.0' encoding='UTF-8' ?>

<!--Generated by XML Authority-->

<!-- This is the XML DTD for the JBoss 3.0 security policy configuration.
The DOCTYPE is:
<!DOCTYPE policy PUBLIC
  "-//JBoss//DTD JBOSS Security Config 3.0//EN"
  "http://www.jboss.org/j2ee/dtd/security_config.dtd">

$Id: security_config.dtd,v 1.1.2.2 2002/06/27 19:21:28 starks Exp $
$Revision: 1.1.2.2 $
```

The outline of the application-policy is:

```
<policy>
  <application-policy name="security-domain-name">
    <authentication>
      <login-module code="login.module1.class.name" flag="control_flag">
```

```
<module-option name = "option1-name">option1-value</module-option>
<module-option name = "option2-name">option2-value</module-option>
...
</login-module>

<login-module code="login.module2.class.name" flag="control_flag">
...
</login-module>
...
</authentication>
</application-policy>
</policy>
-->
<!-- The root element of the security policy configuration --&gt;
&lt;!ELEMENT policy (application-policy+)&gt;

<!-- An application-policy defines the security configuration for an application
domain.
Currently this consists of only the login module configurations specified in the
authentication --&gt;
&lt;!ELEMENT application-policy (authentication)&gt;

<!-- The application-policy name attribute gives the name of the security domain.
--&gt;
&lt;!ATTLIST application-policy name CDATA #REQUIRED&gt;

<!-- The authentication element contains the login module stack configuration.
Each
login module configuration is specified using a login-module element.
--&gt;
&lt;!ELEMENT authentication (login-module+)&gt;

<!-- The login-module element defines a JAAS login module configuration entry.
Each
entry must have a code and flag attribute along with zero or more login module
options
specified via the module-option element.
--&gt;
&lt;!ELEMENT login-module (module-option*)&gt;

<!-- The flag attribute controls how a login module participates in the overall
authentication procedure.
Required      - The LoginModule is required to succeed.
                  If it succeeds or fails, authentication still continues
                  to proceed down the LoginModule list.

Requisite     - The LoginModule is required to succeed.
                  If it succeeds, authentication continues down the
                  LoginModule list. If it fails,
                  control immediately returns to the application
                  (authentication does not proceed down the
                  LoginModule list).

Sufficient    - The LoginModule is not required to
                  succeed. If it does succeed, control immediately</pre>
```

returns to the application (authentication does not proceed down the LoginModule list).
If it fails, authentication continues down the LoginModule list.

- Optional - The LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.

The overall authentication succeeds only if all required and requisite LoginModules succeed. If a sufficient LoginModule is configured and succeeds, then only the required and requisite LoginModules prior to that sufficient LoginModule need to have succeeded for the overall authentication to succeed. If no required or requisite LoginModules are configured for an application, then at least one sufficient or optional LoginModule must succeed.

-->

```
<!ATTLIST login-module flag (required | requisite | sufficient | optional ) #REQUIRED>
```

```
<!-- The code attribute gives the fully qualified class name of the javax.security.auth.spi.LoginModule interface implementation for the login module.
```

-->

```
<!ATTLIST login-module code CDATA #REQUIRED>
```

```
<!-- A module option defines a name, value pair of strings that are passed to a LoginModule when it is initialized during the login procedure. The name attribute defines the option name while the element value is the option string value.
```

-->

```
<!ELEMENT module-option (#PCDATA)>
```

```
<!-- The name attribute specifies the name of the login module option.
```

-->

```
<!ATTLIST module-option name CDATA #REQUIRED>
```

The book comes with the source code for the examples discussed in the book. The examples are included with the book archive. When you unzip the JBossBook_30x.zip archive this creates an AdminDevel directory that contains an examples subdirectory. This is the examples directory referred to by the book.

The only customization needed before the examples may be used it to set the location of the JBoss server distribution. This may be done by editing the examples/build.xml file and changing the **jboss.dist** property value. This is shown in bold below:

```
<project name="JBossBook 3.0 examples" default="build-all" basedir=".">>

    <!-- Allow override from local properties file -->
    <property file=".ant.properties" />
    <!-- Override with your JBoss/Web server bundle dist location -->
    <property name="jboss.dist" value="G:/JBossReleases/jboss-3.0.1RC1"/>
```

or by creating a ‘.ant.properties’ file in the examples directory that contains a definition for the **jboss.dist** property. For example:

```
jboss.dist=D:/usr/JBoss3.0/jboss-all/build/output/jboss-3.0.1RC1
```


Index



A

- AbstractWebContainer, 307
 - Subclassing, 313
- Apache
 - and AJP connector, 327
 - and Tomcat, 327
- ApplicationDeadlockException, 177, 178
- Authentication, 246
- Authentication and authorization, 246
- Authorization, 246

B

- BMP, 182

C

- Catalina See Tomcat-4.x
- Class loading
 - Scoped, 48
- Class versioning, 48
- ClassLoaders
 - Architecture, 46
- ClientLoginModule, 276
- cluster, 182
- CMP, 175, 182
- commit-option, 174

D

- Database
 - Example configurations, 234
- DatabaseServerLoginModule, 274
- Deadlock
 - detection, 177
- deadlock, 176
- Deployment
 - Dependencies, 67, 79

- Ordering, 79
- Descriptions, 35
- dirty reads, 175
- Dynamic MBeans, 43
 - example, 80
- Dynamic proxies, 141

E

- EJB
 - Container cache configuration, 159
 - Container commit option configuration, 161
 - Container configuration, 156
 - Container interceptor configuration, 158
 - Container locking policy configuration, 161
 - Container persistence configuration, 160
 - Container plugin framework, 162
 - DTD validation, 151
 - Instance pool configuration, 158
 - local references See ejb-local-ref
 - method permissions, 242
 - references See ejb-ref
 - Verifier, 151
- ejb-jar.xml
 - ENC elements, 100
 - Security elements, 237
- ejb-local-ref, 109
- ejb-ref, 106
 - and JBoss descriptors, 108
- EmbeddedCatalinaServiceSX, 315
- ENC, 98
 - and UserTransactions, 139
- ENC See Also JNDI Application Component Environment
- env-entry, 105

F

- Flushing authentication info, 265

H

Hot deployment, 78

I

IdentityLoginModule, 269

Instance Per Transaction, 180

Interceptors

Client sid, 141

Invoker

Clustered RMI/JRMP, 149

In VM, 146

RMI/HTTP, 148

RMI/JRMP, 147

J

JAAS

Authentication, 248

Introduction to, 246

Login code, 248

LoginModule, 250

Principal, 247

Subject, 247

JBoss

and JMX, 59

Enabling declarative security, 246

Installing the binary, 22

license, 345

Security model, 251

JBoss Group

About, 345

JBoss Messaging

Client jars, 186

Default destinations, 186

JBossCX

Architecture, 223

JBossNS

Architecture, 115

JBossSecurity, 259

Architecture, 259

jboss-service.xml

DTD, 61

JBossSX

Custom security proxy, 255

Login modules, 269

MBeans, 264

Subject usage pattern, 278

JBossTX

Adapting a Transaction Manager, 137

Internals, 137

jboss-web.xml

context-root, 308

DTD Graphic, 307

ENC elements, 104

virtual-host, 308

jboss.xml, 141, 152, 155

bean-invoker, 146

client-interceptors, 143

commit-option, 161

Container configuration, 156

container-configuration element, 175

container-interceptors, 158

container-invoker, 157

ENC elements, 103

instance-cache, 159

instance-pool, 158

locking-policy, 161

row-locking, 182

security-domain, 162

JCA, 219

Common Client Interface, 220

Overview, 219

Sample adaptor, 227

JDBC

Example DataSource configurations, 234

Jetty, 329

and Apache, 337

and virtual hosts, 337

Clustering, 336

Configuration, 330

Setting up SSL, 334

JMS

Connection factory names, 186

Destination management, 214

Examples, 185

JMX

Connecting using RMI, 52

console application, 49

MBeans, 43

securing the console application, 50

JNDI

Application Component Environment, 98

ENC conventions, 99

ENC See JNDI Application Component Environment

ExternalContext MBean, 125

InitialContext Factory, 117

JBoss jndi.properties settings, 117

Logging in with, 119

MBeans, 125

NamingAlias MBean, 127

NamingService MBean, 115

Over HTTP, 120

Overview, 95

Securing, 122

Securing and read-only, 124

Viewer MBean, 128

JSSE

jars, 302

JBoss and SSL, 300

JTA

Default MBean, 138

UserTransaction, 139

XidFactory MBean, 138

J2EE

declarative security overview, 237

K

keystore, 302

L

LdapLoginModule, 271

LGPL, 345

Log Files

location, 25

Login module

introduction, 250

Login modules

Writing custom login modules, 277

Login modules See JAAS

M

MBean

and using EJBs, 80

attributes and PropertyEditor, 62

JBoss Services, 60

JBoss services See MBean Services

org.jboss.deployment.SARDeployer, 61

org.jboss.deployment.scanner.URLDeploymentScanner, 78

org.jboss.ejb.EJBDeployer, 151

org.jboss.invocation.http.server.HttpInvoker, 148

org.jboss.invocation.jrmp.server JRMPInvoker, 147

org.jboss.invocation.jrmp.server.JRMPInvokerHA, 149

org.jboss.logging.Log4jService, 78

org.jboss.mq.il.jvm.JVMServerILService, 208

org.jboss.mq.il.oil.OILServerILService, 209

org.jboss.mq.il.rmi.RMIServerILService, 209

org.jboss.mq.il.ul.UILServerILService, 210

org.jboss.mq.pm.file.CacheStore, 212

org.jboss.mq.pm.file.PersistenceManager, 212

org.jboss.mq.pm.jdbc2.PersistenceManager, 213

org.jboss.mq.pm.rollinglogged.PersistenceManager, 213

org.jboss.mq.security.SecurityManager, 211

org.jboss.mq.server.jmx.DestinationManager, 211

org.jboss.mq.server.jmx.InterceptorLoader, 210

org.jboss.mq.server.jmx.Invoker, 210

org.jboss.mq.server.jmx.Queue, 214

org.jboss.mq.server.jmx.Topic, 214

org.jboss.mq.server.MessageCache, 212

org.jboss.mq.sm.file.DynamicStateManager, 212

org.jboss.naming.ExternalContext, 125

org.jboss.naming.JNDIView, 128

org.jboss.naming.NamingAlias, 127

org.jboss.naming.NamingService, 115

org.jboss.resource.connectionmanager.BaseConnection
Manager2, 224

org.jboss.resource.connectionmanager.CachedConnectio
nManager, 226

org.jboss.resource.connectionmanager.JBossManagedC
onnectionPool, 226

org.jboss.resource.connectionmanager.LocalTxConnecti
onManager, 224

org.jboss.resource.connectionmanager.NoTxConnection
Manager, 224

org.jboss.resource.connectionmanager.RARDeployment
, 224

org.jboss.resource.connectionmanager.XATxConnection
Manager, 224

INDEX

- org.jboss.resource.RARDeployer, 224
- org.jboss.security.auth.login.XMLLoginConfig, 266
- org.jboss.security.plugins.JaasSecurityDomain, 266
- org.jboss.security.plugins.JaasSecurityManagerService, 265
 - org.jboss.security.plugins.SecurityConfig, 268
 - org.jboss.security.srp SRPVerifierStoreService, 290
 - org.jboss.security.srp.SRPService, 289
 - org.jboss.system.ServiceController, 65
 - org.jboss.tm.TransactionManagerService, 138
 - org.jboss.tm.usertx.server.ClientUserTransactionService, 139
 - org.jboss.tm.XidFactory, 138
 - org.jboss.varia.property.SystemPropertiesService, 339
 - org.jboss.varia.scheduler.Scheduler, 341
 - org.jboss.web.catalina.EmbeddedCatalinaServiceSX, 315
 - org.jboss.web.WebService, 78
 - Specifying dependencies, 67
 - MBean Services, 60, 64
 - Deployment descriptor DTD, 61
 - MBeans
 - Inspecting dependency status, 69
 - method permission, 242
 - Model MBeans, 43
- O**
 - Open MBeans, 43
- P**
 - Passivation
 - timeout setting, 159
 - Properties
 - Managing, 339
 - System, 339
 - ProxyLoginModule, 276
- R**
 - read-only, 179, 180
 - repeatable reads, 180
 - Resource adaptors See JBossCX
 - resource-env-ref, 113
 - and JBoss descriptors, 114
 - resource-ref, 111
 - and JBoss descriptors, 112
- RFC2945 See SRP
- RMI
 - HTTP example config, 149
 - JRMP compressed socket example, 147
 - Over SSL, 300
- rollback, 174, 177
- RunAsLoginModule, 276

S

- SAR
 - definition, 61
- Scheduling, 341
- Security
 - Enabling in JBoss, 246, 254
 - Security Manager
 - Running with, 298
 - security-constraint, 245
 - security-identity, 240
 - security-role, 241
 - security-role-ref, 239
- Servlet Containers
 - Integrating, 307
- SRP, 286
 - Algorithm, 292
 - Example, 295
 - Integrating your security data, 290
 - JBossSX features, 287
 - JBossSX implementation, 287
 - login modules, 288
 - Sample login config, 289
 - SRPLoginModule options, 288
- SRPLoginModule, 288
- SSL
 - and EJBs, 300
 - and JaasSecurityDomain, 266
 - and Tomcat-4.x, 320
 - JSSE, 300
- Standard MBeans, 43
- standardjboss.xml, 141, 152, 153, 155
- Startup
 - Process, 59

T

Timers, 341
Tomcat
 Service descriptor, 317
tomcat4-service.xml, 317
Tomcat-4.x, 315
 and Apache, 327
 and virtual hosts, 325
 Clustering, 328
 Configuring, 317
 Setting up SSL, 320
Transaction
 Overview, 133
transaction, 174, 175

U

UsersRolesLoginModule, 270
UserTransaction
 Support, 139

V

virtual-host See jboss-web.xml

W

web.xml
 ENC elements, 101
 Security elements, 237