

Edition

1

Date: 2002-05-28, 8:55:26 AM

SCOTT STARK, MARC FLEURY

The JBoss Group

JBoss Administration and Development

SCOTT STARK, MARC FLEURY, AND THE JBOSS GROUP

JBoss Administration and Development

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

Table of Content

PREFACE	1
FORWARD.....	1
ABOUT THE AUTHORS	2
DEDICATION.....	3
ACKNOWLEDGMENTS	3
0. INTRODUCTION TO JBOSS	1
ABOUT OPEN SOURCE	1
ABOUT J2EE	2
<i>The J2EE APIs</i>	2
<i>Why Open Source for J2EE?</i>	3
<i>Who Uses J2EE Technology</i>	3
ABOUT JBOSS	4
<i>JBoss: A Full J2EE Implementation with JMX</i>	4
JBossServer	5
JBossMQ	6
JBossTX.....	7
JBossCMP	7
JBossSX.....	9
JBossCX	9
Web Servers.....	10
WHAT THIS BOOK COVERS	10
1. INSTALLING AND BUILDING THE JBOSS SERVER	11
GETTING THE BINARY	11
INSTALLING THE BINARY PACKAGE	12
<i>Directory Structure</i>	12
CONFIGURATION FILES.....	14
<i>auth.conf</i>	15
<i>jboss.conf</i>	15
<i>jboss.jcml</i>	16
<i>jboss.properties</i>	16
<i>jbossmq-state.xml</i>	16
<i>jndi.properties</i>	16
<i>log4j.properties</i>	17
<i>mail.properties</i>	17
<i>standardjaws.xml</i>	17
<i>standardjboss.xml</i>	17
TESTING THE INSTALLATION	17
BUILDING THE SERVER FROM SOURCE CODE	18
<i>Accessing the JBoss CVS Repositories at SourceForge</i>	18
<i>Understanding CVS</i>	18
<i>Anonymous CVS Access</i>	19
<i>Obtaining a CVS Client</i>	19
<i>Understanding the JBoss CVS Modules</i>	19

<i>Building the JBoss-2.4.5 Distribution Using the CVS Source Code</i>	20
<i>Building the JBoss-2.4.5/Tomcat-4.0.3 Integrated Bundle Using the CVS Source Code</i>	26
SUMMARY.....	27
2. JBOSS SERVER ARCHITECTURE OVERVIEW	28
JMX.....	28
<i>An Introduction to JMX</i>	29
<i>Instrumentation Level</i>	30
<i>Agent Level</i>	31
<i>Distributed Services Level</i>	31
<i>JMX Component Overview</i>	32
Managed Beans or MBeans.....	32
Notification Model.....	33
MBean Metadata Classes.....	34
MBean Server.....	34
Agent Services.....	35
The dynamic loading M-Let service.....	36
The MLet Configuration File.....	36
<i>JMX Summary</i>	38
JBOSS AND JMX.....	38
<i>Writing JBoss MBean services</i>	41
The ConfigurationService MBean.....	41
The Service life-cycle interface.....	42
The ServiceControl MBean.....	43
The init method.....	43
The start method.....	43
The stop method.....	44
The destroy method.....	44
Writing JBoss MBean services.....	44
A simple custom MBean example.....	45
<i>The Core JBoss MBeans</i>	50
The bootstrap MBeans, jboss.conf.....	51
org.jboss.logging.Log4jService.....	52
org.jboss.util.ClassPathExtension.....	53
org.jboss.util.Info.....	53
org.jboss.configuration.ConfigurationService.....	53
org.jboss.util.Shutdown.....	54
org.jboss.util.ServiceControl.....	54
The standard MBean services, jboss.jcml.....	54
org.jboss.web.WebService.....	62
org.jboss.ejb.ContainerFactory.....	62
org.jboss.deployment.J2eeDeployer.....	63
org.jboss.ejb.AutoDeployer.....	63
org.jboss.jmx.server.RMIConnectorService.....	64
com.sun.jdmk.comm.HtmlAdaptorServer.....	64
org.jboss.mail.MailService.....	65
org.jboss.util.Scheduler.....	65
<i>JBoss and JMX Summary</i>	66
THE EJB CONTAINER ARCHITECTURE.....	66
<i>EJBOject and EJBHome</i>	66
Virtual EJBOject - the big picture.....	67
Dynamic proxies.....	67
EJB proxy types.....	68
From client to server.....	68
Advantages.....	69
<i>ContainerInvoker – the container transport handler</i>	69
The JRMPContainerInvoker.....	70
ContainerRemote Interface—Two Forms of Invoke Methods.....	71
<i>Handling the method invocations</i>	72
Other ContainerInvoker duties.....	72
<i>The EJB Container</i>	73
ContainerFactory MBean.....	73

Container configuration information.....	75
Verifying EJB deployments.....	78
Deploying EJBs into containers.....	79
Inside the EJB org.jboss.ejb.Container class.....	79
Container Plug-in Framework.....	80
org.jboss.ejb.ContainerPlugin.....	80
org.jboss.ejb.Interceptor.....	81
org.jboss.ejb.InstancePool.....	82
org.jboss.ejb.InstanceCache.....	83
org.jboss.ejb.EntityPersistenceManager.....	84
org.jboss.ejb.StatefulSessionPersistenceManager.....	89
Tracing the call through container.....	90
SUMMARY.....	93
3. JBOSSNS - THE JBOSS NAMING SERVICE.....	94
AN OVERVIEW OF JNDI.....	94
<i>The JNDI API</i>	95
Names.....	95
Contexts.....	96
Obtaining a Context using InitialContext.....	96
J2EE AND JNDI – THE APPLICATION COMPONENT ENVIRONMENT.....	97
<i>ENC Usage Conventions</i>	99
The ejb-jar.xml ENC Elements.....	99
The web.xml ENC Elements.....	101
The jboss.xml ENC Elements.....	104
The jboss-web.xml ENC Elements.....	104
Environment Entries.....	105
EJB References.....	107
EJB References with jboss.xml and jboss-web.xml.....	109
EJB Local References.....	110
Resource Manager Connection Factory References.....	112
Resource Manager Connection Factory References with jboss.xml and jboss-web.xml.....	114
Resource Environment References.....	115
Resource Environment References and jboss.xml, jboss-web.xml.....	116
THE JBOSSNS ARCHITECTURE.....	116
<i>The JBossNS InitialContext Factory</i>	119
ADDITIONAL NAMING MBEANS.....	121
<i>org.jboss.naming.ExternalContext MBean</i>	121
<i>The org.jboss.naming.NamingAlias MBean</i>	123
<i>The org.jboss.naming.JNDIView MBean</i>	124
SUMMARY.....	127
4. JBOSSMQ - THE JBOSS MESSAGING SERVICE.....	128
AN OVERVIEW OF JMS.....	128
<i>The JMS Architecture</i>	129
Message Destinations and Connection Factories.....	129
Messages.....	130
The JMS provider.....	130
<i>The JMS API</i>	131
Connection factories and message destinations interfaces.....	132
Messages.....	133
A PTP Example.....	134
JMS and J2EE.....	137
Message driven beans.....	137
An MDB example.....	138
AN OVERVIEW OF THE JBOSSMQ ARCHITECTURE.....	147
<i>JBossMQ Application Server Facilities</i>	151
CONFIGURING JBOSSMQ.....	153
<i>org.jboss.mq.server.JBossMQService MBean</i>	153
<i>org.jboss.mq.server.StateManager MBean</i>	154
<i>org.jboss.mq.pm.rollinglogged.PersistenceManager MBean</i>	155
<i>org.jboss.mq.pm.file.PersistenceManager MBean</i>	155

<i>org.jboss.mq.pm.jdbc.PersistanceManager MBean</i>	155
<i>org.jboss.mq.il.oil.OILServerILService MBean</i>	156
<i>org.jboss.mq.il.oil.UILServerILService MBean</i>	156
<i>org.jboss.mq.il.jvm.JVMServerILService MBean</i>	157
<i>org.jboss.mq.il.rmi.RMIServerILService MBean</i>	157
<i>org.jboss.mq.server.TopicManager MBean</i>	158
<i>org.jboss.mq.server.QueueManager MBean</i>	158
<i>org.jboss.jms.jndi.JMSProviderLoader MBean</i>	158
<i>org.jboss.jms.asf.ServerSessionPoolLoader MBean</i>	159
SUMMARY	159
5. JBOSSCMP – THE JBOSS CONTAINER MANAGED PERSISTENCE LAYER	160
CONTAINER MANAGED PERSISTENCE – CMP	160
THE JBOSSCMP ARCHITECTURE	161
<i>A Custom file based persistence manager</i>	165
Using the FileStore.....	173
Limitations of the FileStore	180
JAWS – THE DEFAULT CMP IMPLEMENTATION	181
<i>What is O-R mapping?</i>	181
CUSTOMIZING THE BEHAVIOR OF JAWS.....	182
<i>Specifying the DataSource and type mapping</i>	186
Java to SQL type mapping definitions.....	186
Global options and defaults.....	187
Entity bean to database mapping and usage options.....	187
Customization of entity bean home interface finder methods.....	190
Finder examples.....	191
CONFIGURING JDBC	191
<i>The Default JDBC HypersonicDatabase</i>	196
SUMMARY	197
6. JBOSSTX – THE JBOSS TRANSACTION MANAGER.....	198
TRANSACTION/JTA OVERVIEW	198
<i>Pessimistic and optimistic locking</i>	199
<i>The components of a distributed transaction</i>	200
<i>The two-phase XA protocol</i>	201
<i>Heuristic exceptions</i>	201
<i>Transaction IDs and branches</i>	202
<i>Interposing</i>	202
JBOSS TRANSACTION INTERNALS	203
<i>Adapting a Transaction Manager to JBoss</i>	204
<i>The Default Transaction Manager</i>	205
<i>The Tyrex Transaction Manager</i>	205
<i>UserTransaction Support</i>	206
7. JBOSSCX – THE JBOSS CONNECTOR ARCHITECTURE	207
JCA OVERVIEW.....	207
AN OVERVIEW OF THE JBOSSCX ARCHITECTURE	212
<i>ConnectionFactoryLoader MBean</i>	213
<i>ConnectionFactoryLoader MBean</i>	214
<i>RARDeployer MBean</i>	216
<i>A Sample Skeleton JCA Resource Adaptor</i>	216
SUMMARY	228
8. JBOSSSX – THE JBOSS SECURITY EXTENSION FRAMEWORK	229
J2EE DECLARATIVE SECURITY OVERVIEW	229
<i>Security References</i>	232
<i>Security Identity</i>	233
<i>Security roles</i>	234
<i>EJB method permissions</i>	235
<i>Web content security constraints</i>	238
<i>Enabling Declarative Security in JBoss</i>	239

AN INTRODUCTION TO JAAS.....	240
<i>What is JAAS?</i>	240
The JAAS Core Classes.....	241
Subject and Principal.....	241
Authentication of a Subject.....	242
THE JBOSS SECURITY MODEL.....	245
<i>Enabling Declarative Security in JBoss Revisited.</i>	248
THE JBOSSSX SECURITY EXTENSION ARCHITECTURE.....	254
<i>How the JaasSecurityManager Uses JAAS.</i>	256
<i>The JaasSecurityManagerService MBean.</i>	260
An Extension to JaasSecurityManagerService, the JaasSecurityDomain MBean.....	262
USING AND WRITING JBOSSSX LOGIN MODULES.....	263
<i>org.jboss.security.auth.spi.IdentityLoginModule</i>	263
<i>org.jboss.security.auth.spi.UsersRolesLoginModule</i>	264
<i>org.jboss.security.auth.spi.LdapLoginModule</i>	266
<i>org.jboss.security.auth.spi.DatabaseServerLoginModule</i>	270
<i>org.jboss.security.auth.spi.ProxyLoginModule</i>	272
<i>org.jboss.security.ClientLoginModule</i>	272
<i>Writing Custom Login Modules.</i>	273
Support for the Subject Usage Pattern.....	274
THE SECURE REMOTE PASSWORD (SRP) PROTOCOL.....	279
<i>Inside of the SRP algorithm</i>	283
<i>An SRP example</i>	287
RUNNING JBOSS WITH A JAVA 2 SECURITY MANAGER.....	291
USING SSL WITH JBOSS USING JSSE.....	293
SUMMARY.....	300
9. ADVANCED JBOSS CONFIGURATION USING JBOSS.XML.....	301
THE JBOSS.XML DESCRIPTOR.....	301
<i>The container-name Element</i>	307
<i>The call-logging element</i>	307
<i>The container-invoker and container-invoker-conf elements</i>	307
<i>The container-interceptors element</i>	309
<i>The instance-pool and container-pool-conf elements</i>	310
<i>The instance-cache and container-cache-conf elements</i>	310
<i>The persistence-manager element</i>	312
<i>The transaction-manager element</i>	312
<i>The locking-policy element</i>	313
<i>The commit-option and optiond-refresh-rate element</i>	313
<i>The security-domain, role-mapping-manager and authentication-module elements</i>	314
SUMMARY.....	314
10. INTEGRATING SERVLET CONTAINERS.....	315
THE ABSTRACTWEBCONTAINER CLASS.....	315
<i>The AbstractWebContainer Contract</i>	316
<i>Creating an AbstractWebContainer Subclass</i>	321
Use the Thread Context Class Loader.....	321
Integrate Logging Using log4j.....	321
Delegate web container authentication and authorization to JBossSX.....	322
JBOSS/TOMCAT-4.X BUNDLE NOTES.....	323
<i>The Embedded Tomcat Configuration Elements</i>	324
Server.....	324
Service.....	325
Connector.....	325
The HTTP Connector.....	325
The AJP Connector.....	326
The Warp Connector.....	326
Engine.....	327
Host.....	327
Alias.....	327
DefaultContext.....	328
Manager.....	328

Logger	328
Valve	328
Listener	329
<i>Using SSL with the JBoss/Tomcat bundle</i>	329
<i>Setting up Virtual Hosts with the JBoss/Tomcat-4.x bundle</i>	333
<i>Using Apache with the JBoss/Tomcat-4.x bundle</i>	335
JBoss/TOMCAT-3.2.3 BUNDLE NOTES	337
<i>Using SSL with the JBoss/Tomcat bundle</i>	337
JBoss/JETTY-4.0.0 BUNDLE NOTES	337
SUMMARY	338
11. USING JBOSS	339
BUILDING AND RUNNING ENTERPRISE APPLICATIONS WITH JBOSS	339
<i>The MailAccount, MailHandlerMDB and NewMailService component details</i>	342
<i>Building and assembling the mail forwarding application</i>	347
<i>Testing the mail forwarding application</i>	357
<i>Securing the mail forwarding application</i>	362
Building the secured mail forwarding application	370
Testing the secured mail forwarding application	372
MIGRATING THE JAVA PET STORE 1.1.2 APPLICATION TO JBOSS	374
<i>Patching the JPS distribution</i>	375
Creating the jboss.xml and jboss-web.xml descriptors	377
Configure the Hypersonic database	380
Building and deploying the JPS EAR	385
USING THE JBOSS TEST UNIT TESTSUITE	386
12. APPENDIX A	391
ABOUT THE JBOSS GROUP	391
THE GNU LESSER GENERAL PUBLIC LICENSE (LGPL) AND X LICENSE	391
13. APPENDIX B	406
JBoss DESCRIPTOR SCHEMA REFERENCE	406
<i>The JBoss server jboss.xml descriptor DTD</i>	406
<i>The JBoss server jaws.xml descriptor DTD</i>	421
<i>The JBoss server jboss-web.xml descriptor DTD</i>	424
<i>The JBoss server jboss.jcm1 configuration file DTD</i>	425
<i>The JBoss server jbossmq-state.xml configuration file DTD</i>	427
14. APPENDIX C	429
THE BOOK CD CONTENTS	429
15. APPENDIX D, TOOLS AND EXAMPLES	431
USING ANT	431
USING THE LOG4J FRAMEWORK IN JBOSS	437
<i>The org.apache.log4j.Category class</i>	437
The JBoss org.jboss.log.Logger wrapper	439
<i>The org.apache.log4j.Appender interface</i>	440
<i>The org.apache.log4j.Layout class</i>	440
<i>Configuring log4j using org.apache.log4j.PropertyConfigurator</i>	440
<i>Log4j usage patterns</i>	445
<i>The Log4jService MBean revisited</i>	446
INSTALLING AND USING THE BOOK EXAMPLES	447
<i>Building and Running An Example</i>	448
16. APPENDIX E, CHANGE NOTES	450
2.4.5 CHANGES	450
<i>Changes between JBoss_2_4_5_RC1 and JBoss_2_4_4</i>	450
Rel_2_4_5_1	450
Rel_2_4_5_2	450
Rel_2_4_5_3	451
Rel_2_4_5_4	451

I P R E F A C E

Rel_2_4_5_5.....	451
Rel_2_4_5_6.....	451
Rel_2_4_5_9.....	451
Rel_2_4_5_10.....	451
Rel_2_4_5_11.....	451
REL_2_4_5_12.....	453
Rel_2_4_5_13.....	453
Rel_2_4_5_14.....	453
Rel_2_4_5_15.....	453
Rel_2_4_5_16.....	453
<i>Changes between JBoss_2_4_5_RC2 and JBoss_2_4_5_RC1.....</i>	<i>455</i>
Rel_2_4_5_17.....	455
Rel_2_4_5_18.....	455
Rel_2_4_5_19.....	455
Rel_2_4_5_20.....	455
Rel_2_4_5_21.....	455
<i>Changes between JBoss_2_4_5_RC3 and JBoss_2_4_5_RC2.....</i>	<i>456</i>
Rel_2_4_5_22.....	456
<i>Changes between JBoss_2_4_5 and JBoss_2_4_5_RC3.....</i>	<i>456</i>
Rel_2_4_5_23.....	456
Rel_2_4_5_24.....	456
Rel_2_4_5_25.....	457
17. INDEX.....	458

Table of Listings

<i>Listing 1-1, the JBoss 2.4.x branch build process</i>	21
<i>Listing 1-2, the build of the JBoss/Tomcat distribution</i>	26
<i>Listing 2-1, MLET configuration file syntax</i>	36
<i>Listing 2-2, The setup of the MBeanServer and MLet on JBoss startup</i>	39
<i>Listing 2-3, Using JMX to load the JBoss configured components</i>	39
<i>Listing 2-4, A sample jboss.jcml MBean declaration</i>	40
<i>Listing 2-5, The corresponding MLET tag for Listing 2.4</i>	40
<i>Listing 2-6, The org.jboss.util.Service interface</i>	42
<i>Listing 2-7, JNDIMapMBean interface and implementation based on the service interface method pattern</i>	45
<i>Listing 2-8, JNDIMap MBean interface and implementation based on the ServiceMBean interface and ServiceMBeanSupport class</i>	48
<i>Listing 2-9, A sample jboss.jcml entry for the JNDIMap MBean and a client usage code fragment</i>	50
<i>Listing 2-10, the default jboss.conf bootstrap configuration file from the standard JBoss distribution</i>	51
<i>Listing 2-11, the default jboss.jcml services configuration file from the standard JBoss distribution</i>	54
<i>Listing 2-12, The org.jboss.ejb.plugins.jrmp.interfaces.ContainerRemote interface implemented by the JRMPContainerInvoker class</i>	70
<i>Listing 2-13, The org.jboss.ejb.ContainerFactoryMBean interface</i>	73
<i>Listing 2-14, The most common container-configuration elements in the default distribution standardjboss.xml file</i>	77
<i>Listing 2-15, the org.jboss.ejb.ContainerPlugin interface</i>	80
<i>Listing 2-16, the org.jboss.ejb.Interceptor interface</i>	81
<i>Listing 2-17, the org.jboss.ejb.InstancePool interface</i>	82
<i>Listing 2-18, the org.jboss.ejb.InstanceCache interface</i>	83
<i>Listing 2-19, the org.jboss.ejb.EntityPersistenceManager interface</i>	84
<i>Listing 2-20, the org.jboss.ejb.EntityPersistenceStore interface</i>	87
<i>Listing 2-21, the org.jboss.ejb.StatefulSessionPersistenceManager interface</i>	90
<i>Listing 3-1, sample jndi.properties file</i>	97
<i>Listing 3-2, ENC access sample code</i>	98
<i>Listing 3-3, ejb-jar.xml env-entry fragment</i>	106
<i>Listing 3-4, ENC env-entry access code fragment</i>	107
<i>Listing 3-5, example ejb-jar.xml ejb-ref descriptor fragment</i>	108
<i>Listing 3-6, ENC ejb-ref access code fragment</i>	109
<i>Listing 3-7, example jboss.xml ejb-ref fragment</i>	109
<i>Listing 3-8, example ejb-jar.xml ejb-local-ref descriptor fragment</i>	111
<i>Listing 3-9, ENC ejb-local-ref access code fragment</i>	112
<i>Listing 3-10, web.xml resource-ref descriptor fragment</i>	113
<i>Listing 3-11, ENC resource-ref access sample code fragment</i>	114
<i>Listing 3-12, sample jboss-web.xml resource-ref descriptor fragment</i>	114
<i>Listing 3-13, an example ejb-jar.xml resource-env-ref fragment</i>	115
<i>Listing 3-14, ENC resource-env-ref access code fragment</i>	115
<i>Listing 3-15, sample jboss.xml resource-env-ref descriptor fragment</i>	116
<i>Listing 3-16, ExternalContext MBean configurations</i>	122
<i>Listing 4-1, A simple PTP example that demonstrates the basic steps for a JMS client</i>	134
<i>Listing 4-2, The example2 message driven EJB</i>	138
<i>Listing 4-3, the TextMDB ejb-jar.xml and jboss.xml deployment descriptors</i>	142
<i>Listing 4-4, The example2 JMS client</i>	143
<i>Listing 5-1, the example filesystem based implementation of the EntityPersistenceStore interface</i>	165
<i>Listing 5-2, The FileStore usage example entity bean home, remote interfaces and bean class</i>	173
<i>Listing 5-3, The ejb-jar.xml and jboss.xml descriptors for the FileStore example entity bean jar</i>	176
<i>Listing 5-4, The FileStore testcase client application</i>	177
<i>Listing 5-5, the default standardjaws.xml descriptor with all but the Hypersonic SQL type-mapping elements removed</i>	184
<i>Listing 7-1, the jboss.jcml ConnectionFactoryLoader MBean configuration fragment for the example resource adaptor</i>	218
<i>Listing 7-2, the stateless session bean echo method code which shows the access of the resource adaptor connection factory</i>	220
<i>Listing 8-1, example ejb-jar.xml and web.xml descriptor fragments which illustrate the security-role-ref element usage</i>	233
<i>Listing 8-2, an example ejb-jar.xml descriptor fragment which illustrates the security-identity element usage</i>	234
<i>Listing 8-3, example ejb-jar.xml and web.xml descriptor fragments which illustrate the security-role element usage</i>	235
<i>Listing 8-4, an example ejb-jar.xml descriptor fragment which illustrates the method-permission element usage</i>	236
<i>Listing 8-5, a web.xml descriptor fragment which illustrates the use of the security-constraint and related elements</i>	239
<i>Listing 8-6, an illustration of the steps of the authentication process from the application perspective</i>	243
<i>Listing 8-7, The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint</i>	250
<i>Listing 8-8, the jboss.xml descriptor which configures the EchoSecurityProxy as the custom security proxy for the EchoBean</i>	252
<i>Listing 8-9, the jboss.jcml SRP MBean service configuration entries used with example 2</i>	289

Listing 8-10, the JBoss server auth.conf JAAS login configuration file used with example 2.	290
Listing 8-11, the JBoss client auth.conf JAAS login configuration file used with example 2.	291
Listing 8-12, the modifications to the Win32 run.bat start script to run JBoss with a Java 2 security manager.	292
Listing 8-13, the modifications to the UNIX/Linux run.sh start script to run JBoss with a Java 2 security manager.	292
Listing 8-14, The jboss.xml container configuration to enable SSL with stateless session beans.	297
Listing 9-1, the Chapter 8 jboss.xml container configuration to enable SSL with stateless session beans.	305
Listing 10-1, Key methods of the AbstractWebContainer class.	316
Listing 10-2, a psuedo-code description of authenticating a user via the JBossSX API and the java:comp/env/security JNDI context.	322
Listing 10-3, a psuedo-code description of authorization a user via the JBossSX API and the java:comp/env/security JNDI context.	322
Listing 10-4, the JaasSecurityDomain and EmbeddedCatalinaSX MBean configurations for setting up Tomcat-4.x to use SSL as its primary connector protocol.	329
Listing 10-5, the JaasSecurityDomain and EmbeddedCatalinaSX MBean configurations for setting up Tomcat-4.x to use both non-SSL and SSL enabled HTTP connectors.	330
Listing 10-6, An example virtual host configuration.	334
Listing 10-7, An example jboss-web.xml descriptor for deploying a WAR to the www.starkinternational.com virtual host	334
Listing 10-8, Output from the www.starkinternational.com Host component when the Listing 10-7 WAR is deployed.	335
Listing 10-9, an example EmbeddedCatalinaSX MBean configuration that supports integration with Apache using the Ajpv13 protocol connector.	335
Listing 11-1, the book examples build.xml file build.path and client.path classpaths demonstrating the standard requirements for compiling and running with JBoss.	347
Listing 11-2, the ejb-jar.xml descriptor for the mail forwarding application enterprise beans.	350
Listing 11-3, the jboss.xml descriptor required for specification of the MailHandlerMDB deployment settings.	352
Listing 11-4, the JBossCMP jaws.xml descriptor that provides the SQL statements for the custom MailAccountHome interface finders.	353
Listing 11-5, the web.xml descriptor for the web components of the mail forwarding application.	353
Listing 11-6, the application.xml descriptor for the mail application EAR.	354
Listing 11-7, the mail application Ant build.xml file targets for the creation of the application packages.	355
Listing 11-8, the jboss.jcml MBean configuration entries for the MailQueue destination and custom NewMailService.	356
Listing 11-9, the revised ejb-jar.xml descriptor used by the secured version of the mail forwarding application.	363
Listing 11-10, the revised jboss.xml descriptor used by the secured version of the mail forwarding application.	366
Listing 11-11, the revised web.xml descriptor and new jboss-web.xml descriptor used by the secured version of the mail forwarding application.	366
Listing 11-12, the MailAccountServlet logic for obtaining the application domain identity of the authenticated caller.	368
Listing 11-13, the jps1.1.2/src/components/customer/src/jboss.xml file for the customerEjb.jar. The jboss.xml descriptor is required to map the JDBC javax.jdbc.DataSource references to the deployment environment resource factory location.	377
Listing 11-14, the jps1.1.2/src/components/inventory/src/jboss.xml file for the inventoryEjb.jar. The jboss.xml descriptor is required to map the JDBC javax.jdbc.DataSource reference to the deployment environment resource factory location.	378
Listing 11-15, the jps1.1.2/src/components/mail/src/jboss.xml file for the mailerEjb.jar. The jboss.xml descriptor is required to map the JavaMail javax.mail.Session reference to the deployment environment resource factory location.	378
Listing 11-16, the jps1.1.2/src/components/personalization/src/jboss.xml file for the personalizationEjb.jar. The jboss.xml descriptor is required to map the JDBC javax.jdbc.DataSource reference to the deployment environment resource factory location.	378
Listing 11-17, the jps1.1.2/src/components/shoppingcart/src/jboss.xml file for the shoppingcartEjb.jar. The jboss.xml descriptor is required to map the JDBC javax.jdbc.DataSource reference to the deployment environment resource factory location.	379
Listing 11-18, the jboss.xml file for the jps1.1.2/src/components/signon/src signonEjb.jar. The jboss.xml descriptor is required to map the JDBC javax.jdbc.DataSource reference to the deployment environment resource factory location.	379
Listing 11-19, the jps1.1.2/src/petstore/src/docroot/WEB-INF/jboss-web.xml file for the petstore.war. The jboss-web.xml descriptor is required to map the JDBC javax.jdbc.DataSource reference to the deployment environment resource factory location, as well as setting the locations of the EJB reference homes. Note that the EJB references could have been handled in the web.xml descriptor using ejb-link elements.	379
Listing 11-20, the jboss.jcml configuration fragment for setting up the JPS DataSource factories.	381
Listing 12-1, the GNU lesser general public license text	391
Listing 12-2, the X license text	404
Listing 13-1, the jboss_2_4.dtd file	406
Listing 13-2, the jaws_2_4.dtd file	421
Listing 13-3, the jboss-web.dtd file	424
Listing 13-4, the jboss.jcml file DTD	425
Listing 13-5, the jbossmq-state.xml file DTD	427
Listing 15-1, An Ant build.xml script for testing the Ant installation.	432
Listing 15-2, a summary of the key methods in the log4j Category class.	438
Listing 15-3, The JBoss Logger class summary.	439
Listing 15-4, the standard JBoss log4j.properties configuration file.	441

Table of Figures

Figure 0-1: The JBoss JMX integration bus and the standard JBossXX components.	5
Figure 1-1, the directory structure of the JBoss distribution installation	13
Figure 1-2, a view of the JBoss server default configuration set.	15
Figure 2-1, The JBoss JMX integration bus and the standard JBoss components.	29
Figure 2-2, The Relationship between the components of the JMX architecture	30
Figure 2-3, The DTD structure for the jboss.jcml configuration file	40
Figure 2-4, The JBoss components involved in delivering an EJB method invocation to the EJB container.	69
Figure 2-5, The jboss_2_4 DTD elements related to container configuration.	77
Figure 3-1, The ENC elements in the standard EJB 2.0 ejb-jar.xml deployment descriptor.	100
Figure 3-2, The ENC elements in the standard servlet 2.3 web.xml deployment descriptor.	102
Figure 3-3, The ENC elements in the JBoss 2.4 jboss.xml deployment descriptor.	104
Figure 3-4, The ENC elements in the JBoss 2.4 jboss-web.xml deployment descriptor.	105
Figure 3-5, Key components in the JBossNS architecture.	116
Figure 3-6, The HTTP JMX agent view of the configured JBoss MBeans.	124
Figure 3-7, The HTTP JMX MBean view of the JNDIView MBean.	125
Figure 3-8, The HTTP JMX view of the JNDIView list operation output.	126
Figure 4-1, An overview of the key component interfaces in the javax.jms package.	131
Figure 4-2, The MDB portion of the DTD for the jboss.xml descriptor.	141
Figure 4-3, An overview of the key components in the JBossMQ server components and their relationships.	147
Figure 4-4, An overview of the key components in the optimized invocation layer (OIL) transport implementation.	149
Figure 4-5, The JBoss and JBossMQ ASF components and their relationships.	152
Figure 4-6, The DTD for the StateManager user-password, user-durable-subscription XML file	154
Figure 5-1, The JBoss CMP persistence abstraction layer classes.	161
Figure 5-2, The JAWS 2.4 XML configuration file DTD	182
Figure 7-1, An overview of the interaction between the key participants defined by the JCA specification.	209
Figure 7-2, The JCA 1.0 specification figure 6.0 for the class diagram for the connection management architecture.	211
Figure 7-3, The key JBossCX classes that make up the JBoss server JCA system level contract implementation.	212
Figure 7-4, A class diagram for the example file system resource adaptor.	218
Figure 7-5, A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor during deployment.	222
Figure 7-6, 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.	226
Figure 8-1, A subset of the EJB 2.0 deployment descriptor content model that shows the security related elements.	232
Figure 8-2, A subset of the Servlet 2.2 deployment descriptor content model that shows the security related elements.	232
Figure 8-3, The key security model interfaces and their relationship to the JBoss server EJB container elements.	246
Figure 8-4, The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.	248
Figure 8-5, The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.	250
Figure 8-6, The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.	256
Figure 8-7, An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.	258
Figure 8-8, An LDAP server configuration compatible with the testLdap sample configuration.	269
Figure 8-9, The JBossSX components of the SRP client-server framework.	281
Figure 8-10, The SRP client-server authentication algorithm sequence diagram.	284
Figure 8-11, A sequence diagram illustrating the interaction of the SRPCacheLoginModule with the SRP session cache.	287
Figure 9-1, The jboss.xml descriptor DTD EJB related elements.	303
Figure 9-2, The jboss.xml descriptor DTD EJB container configuration related elements.	305
Figure 10-1, The complete jboss-web.xml descriptor DTD.	316
Figure 10-2, An overview of the Tomcat-4.0.3 configuration DTD supported by the EmbeddedCatalinaServiceSX Config attribute.	324
Figure 10-3, The Internet Explorer 5.5 security alert dialog.	332
Figure 10-4, The Internet Explorer 5.5 SSL certificate details dialog.	333
Figure 11-1, An overview of the email forwarding application architecture.	340
Figure 11-2, The MailAccount CMP entity bean and the AccountInfo bulk accessor representation classes.	342
Figure 11-3, The MailHandlerMDB and associated classes.	344
Figure 11-4, The NewMailService MBean and associated classes.	346
Figure 11-5, The mail forwarding application home page.	359
Figure 11-6, The mail forwarding application account management page filled out with some fictional user information.	360
Figure 11-7, An example text message forwarded from a mail account message to a cell phone.	362
Figure 11-8, The secured mail forwarding application home page and login dialog box that is displayed after clicking on the edit account link.	373

<i>Figure 11-9, The mail forwarding account information page for the mail forwarding service user "duke".</i>	374
<i>Figure 11-10, The 1.1.2 Java Pet Store application distribution directory structure.</i>	376
<i>Figure 11-11, The Java Pet Store Demo Database Populate browser view you will see when the JPS database tables are found to be missing.</i>	382
<i>Figure 11-12, The Java Pet Store Demo Database Populate installation page view.</i>	383
<i>Figure 11-13, The Java Pet Store Demo Database Populate installation page view after successful installation of the JPS tables into the Hypersonic database.</i>	384
<i>Figure 11-14, The JBossTest CVS module directory structure.</i>	387
<i>Figure 11-15, An example JBossTest test suite run report status html view as generated by the test-and-report target.</i>	390
<i>Figure 15-1, The DTD for the configuration documents supported by the log4j version 1.1.3 DOMConfigurator.</i>	445



Preface

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.

Crossing into mass-adoption of our framework by the IT corporate world, one of the first points to be addressed was the need for more thorough documentation. I must credit and thank Michael Stephens, our editor at SAMS, for being one of our biggest supporters and for giving us the opportunity to bring JBoss' documentation up to professional quality. Lead JBoss developer and JBoss Group CTO, Scott Stark, spent a great deal of time and effort in rewriting and improving JBoss' documentation so that we could bring you the most complete JBoss application server module coverage possible in a professional book.

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,

marcf

xxxxxxxxxxxxxxxxxxxxx
 Marc Fleury
 President
 JBoss Group, LLC
 xxxxxxxxxxxxxxxxxxxxx

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 1000 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 50,000 downloads per month, JBoss is the most downloaded J2EE based server in the industry.

Dedication

Scott dedicates this book to his parents who continue to love and support me even though I work at providing free software and have not yet demonstrated that I will one day surpass the empire of Bill Gates.

Acknowledgments

I would like to thank Vladimir Blagojevic for his help with on Chapter 2 as well as proof reading of several chapters. Ole Husgaard, who leads the JBossTX development, also provided a great deal of help with the JBossTX chapter. I would also like to thank the JBoss users who have contributed work to the early JBoss documentation effort. Key people from this group include Andreas Schaefer, Simone Bordet, David Jencks, Kevin Boone, Sebastien Alborini, Vincent Harcq, Aaron Mulder, Tom Coleman, Peter Antman, Tobias Frech, and Vladimir Blagojevic

0. Introduction to JBoss

This book is for the JBoss content developer and administrator. The topics covered are those necessary to install, configure and use version 2.4.x of the JBoss Open Source application server.

About Open Source

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:

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.

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 J2EE

The Java™ 2 Platform, Enterprise Edition (J2EE), defines the standard for developing multitier enterprise applications. J2EE simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically, without complex programming. The J2EE platform encompasses a number of technologies including EJBs, JMS, Servlets, JSP, JNDI, JDBC, JCA, JTA, JAAS and CORBA.

The J2EE APIs

J2EE is a set of standards that, when used together, provide an excellent web Web application development and deployment platform. J2EE includes standards for middleware (EJB and JMS), database connectivity (JDBC), transactions (JTA/JTS), presentation (servlets and Java Server Pages), and directory services (JNDI). A summary of the range of APIs/technologies included in J2EE is given in Table 0-1.

Table 0-1: The J2EE APIs and their descriptions

API Name	Description	(JavaSoft URL)
EJBs	Enterprise JavaBeans	(http://java.sun.com/products/ejb)
CORBA	Common Object Request Broker Architecture	(http://java.sun.com/j2ee/corba)
Servlets	Java Servlets	(http://java.sun.com/products/servlet)
JNDI	Java Naming and Directory Interface	(http://java.sun.com/products/jndi)
JDBC	Database Connectivity	(http://java.sun.com/products/jdbc)
XML	Extensible Markup Language	(http://java.sun.com/xml)
JMS	Java Message Service	(http://java.sun.com/products/jms)
JTA/JTS	Transactions	(http://java.sun.com/j2ee/transactions.html)
Connector	Enterprise Information Systems Connector	(http://java.sun.com/j2ee/connector)

JSP	Java Server Pages	(http://java.sun.com/products/jsp)
-----	-------------------	---

For additional information on the technologies of the J2EE platform, see the urls URLs referenced in Table 0-1, or see the JavaSoft J2EE home page at <http://java.sun.com/j2ee>.

Why Open Source for J2EE?

As a Web operating system, J2EE is infrastructure. As such, we believe it is a natural fit for the collaborative, Open Source mode of development facilitated by the Internet. Our group, composed of volunteers from around the world, chooses to open the server and container development. Our belief is that there should be an Open Source J2EE server environment that makes J2EE available to anyone.

The extreme size and complexity of this sort of operating system is yet another compelling reason for it to exist in Open Source. Even Microsoft has had difficulties stabilizing Windows 2000. We at JBoss believe that Open Source technology is a credible, efficient and cost-effective way to scale the development of these large systems.

Who Uses J2EE Technology

As seen from Table I, the range of middleware functionality covered by the J2EE platform is quite extensive. These technologies can be used as a single comprehensive platform, or individually as an enabling technology, or even as a single component. This leads to the question of how the technologies of the J2EE platform are used, and by who. The following list provides some example of who is make use of J2EE.

- **Independent software vendors** — Two years ago, many independent Software software vendors (ISVs) developing Enterprise applications took the Java route. ISVs develop in-house proprietary infrastructure software because of the lack of a defined, open standard. This development is time-consuming, expensive, and complex. Today, most ISVs outsource that infrastructure development to a J2EE server vendor to be able to focus more on "business logic." Choosing an open source server makes sense from a pricing standpoint because the application price won't reflect the infrastructure cost. It also makes sense from a technological standpoint because you have access to the code, which makes for a tighter integration with applications. According to our statistics, about 20% of people who download JBoss do so with the objective of embedding it in their applications.
- **IT departments and startups** — A recent study showed that Java/J2EE, which claims 60% of IT development, is already the dominant platform for Enterprise Web

Software. Most people use our container as a stand-alone web Web application server. In many instances, we have been chosen over more pricey competitors for both development and production. We sport features, such as hot deploy, and runtime-generated stub, and skeleton objects (distributed invocation enablers), that can't be found in most commercial tools, no matter how much you are willing to pay!

- **ISP/ASP, the next wave of Enterprise Software Hosting** — Most ISP providers already offer Web Hosting for static Web pages. For more "enterprise level hosting," you need a J2EE platform. Going beyond simple logic and cgi-bin, JBoss was designed for Application Service Provider (ASP) settings. You can deploy its applications on a set of hosted machines, and have a Web-based Java Management Extension (JMX) console to manage the remote servers. Our integration with Java Server Page (JSP) engines makes JBoss the candidate of choice for ISP usage. While most J2EE vendors do not focus on this market, JBoss is well suited for it in two ways. First, the code is modular so you can administer various configurations, to fit every client's specific needs. Second, there is no license fee per CPU, so you can grow a J2EE server farm at little cost.
- **Module and third-party developers** — Behind JBoss' Open Source success is a highly modular design, which allows us to scale development and integrate code. From the ground up, JBoss is built around the concept of modules and plug-ins. We use the JMX specification to configure and administer the different plug-ins. We integrate various modules, from Tomcat to CocoBase, to offer a state-of-the-art J2EE container. By integrating in JBoss, developers gain access to the dominant application development market and increase the deployment potential for their technology.

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 Management 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.

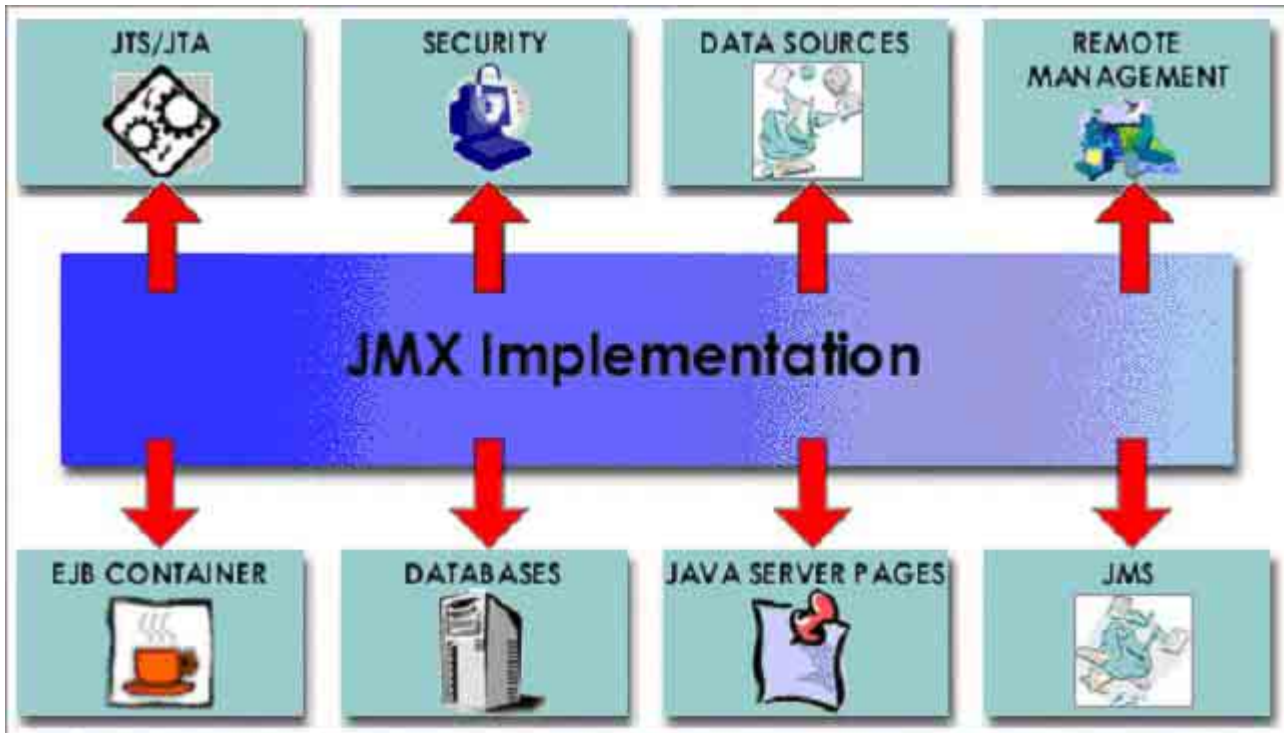


Figure 0-1: The JBoss JMX integration bus and the standard JBossXX components.

While we provide JBoss implementations for many of these services, you are free to include your favorite implementation on top of the JMX enabled base. This allows you to integrate your own transaction or persistence service in JBoss.

JBossServer

In addition to the fact that JBoss is an EJB 1.1-compliant application server, there are some innovative features that make our server a pleasure to use. Specifically, two features make application deployment extremely easy to perform, saving developers much time and effort. The JBoss server takes the grunt work out of EJB application development.

First, there's dynamically, runtime-generated stub and skeleton classes. In many commercial EJB servers, the generation of these classes must be performed in an additional step prior to deployment (such as using an "ebjc" tool). It goes without saying that this extra step requires additional developer overhead, adding significant time to each change-compile-deploy cycle. By generating stub and skeleton classes on the fly, JBoss/Server takes at least several seconds, and perhaps minutes, off of each deployment. As an added benefit, the

method used by JBoss to accomplish this time and effort-saving feature also conserves memory and other server resources because only a single server object supports every deployed Enterprise JavaBeans component.

A second timesaving feature is automatic hot deploy and redeploy. Some of the top commercial EJB servers require you to bounce the server to be able to successfully deploy your application changes. However, JBoss/Server allows you to deploy new applications and redeploy existing applications without stopping and restarting the server. In fact, the feature is as easy as copying your newly built EJB JAR file to the server deployment directory where JBoss/Server picks up the new file, automatically undeploys the old JAR (if any), and deploys the new JAR within seconds. This feature definitely provides the benefit of slicing minutes off of each change-compile-deploy cycle.

JBossMQ

JBossMQ (originally spyderMQ) was released in April 2000 as the first free implementation of the Java Messaging Service (JMS) specification. Based on the 1.0.2 JMS specification, JBossMQ is a clean room, pure java implementation.

It is not uncommon for the Web to fail, for nodes to fail, and for communications in general to fail. Therefore, distributed applications cannot always depend on a synchronous messaging model to reliably deliver notifications. That's why, in addition to synchronous messaging, JMS also provides an asynchronous messaging model that implements the Publish/Subscribe design pattern. A Publish/Subscribe model is critical for successful collaboration between the various participants of a distributed, e-business application. We believe JMS, through our JBossMQ component, plays a central role in the J2EE-based Web operating system provided by the JBoss.

Every aspect of the JMS 1.0.2 spec has been implemented, including

- Both point-to-point and publish-subscribe style messaging
- Durable subscribers
- JMS Application Server Facilities
- The ability to participate in global units of work coordinated by a transaction manager

JBossMQ supports several different kinds of message transport/invoke layers that include

- **RMI** — RMI-based invocation layer.

- **OIL** — Optimized Invocation Layer. This layer uses custom TCP/IP sockets to obtain good network performance and small memory footprint.
- **UIL** — For client applications that cannot accept network connections originating from the server.

JBossTX

JBossTX is a transaction monitor with JTA/JTS support. The Java Transaction Service (JTS) specifies the implementation of a Transaction Manager, which supports the Java Transaction API (JTA) 1.0 specification at the high level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the low level.

JTA allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. JBossTX implements the transaction manager with the Java Transaction Service (JTS), but your code doesn't call the JTS methods directly. Instead, it invokes the JTA methods, which then call the lower level JTS routines.

JBossTX offers the following services:

- Provides applications and application servers the capability to control the scope and duration of a transaction
- Allows multiple application components to perform work that is part of a single, atomic transaction
- Provides the capability to associate global transactions with work performed by transactional resources
- Coordinates the completion of global transactions across multiple resource managers
- Supports transaction synchronization

JBossCMP

During development of JBoss version 1.0 (then known as EJBoss), we put together an object-to-relational (O-R) mapping tool. Enter JAWS, the acronym for Just Another Web Storage, which is an API for mapping Enterprise JavaBeans objects to relational database persistent stores. JAWS has since been renamed JBossCMP and the project has since taken on a life of its own. That's because we are not only maintaining and enhancing the original code base that defined a simple, yet proprietary O-R mapping tool. We are also extending the product to support the popular third-party O-R mapping tools being employed by some JBoss users.

The Minerva JDBC connection pooling module has been added to the codebase. This module complements JBossCMP by adding a pluggable connection pooling mechanism.

O-R mapping technology grew out of the differences between how object-oriented languages represent objects in memory and how relational databases store data on disk. Objects in the Java language might contain only primitive data types, such as `int`, `double`, and very simple aggregate objects such as `String`, making it easy to express the object's layout on disk. In the case of storing such a simple object in a flat disk file, you would just write each primitive data type variable and each `String` object in their string form sequentially into the flat file. As you can imagine, reading such objects back from disk into a memory-based object would be just as easy. However, what about storing more complex objects such as those that contain other objects that contain yet other objects? And what about storing both simple and complex objects into relational databases?

Of course, the more complex the object that must be stored, the more intelligent the O-R mapping tool must be. An O-R mapping tool must understand how to traverse the complex object's memory graph and figure out how to store it to and read it from the persistent store. To add to the complexity, the graph of a single object might contain multiple objects that each references a single, unique object, and it could also contain objects that recursively reference themselves or the original object. In these cases, the O-R mapping tool would have to avoid persisting the same object multiple times, perhaps even ending up in an endless loop because of the self-referencing composition! On the other hand, all complex Java objects finally boil down to variables of primitive data types and those of class `String`. Therefore, while it can be quite challenging to persist very complex objects, it is not impossible. There is definitely light at the end of the tunnel.

The JBossCMP features include

- CMP 1.1 implementation
- JDBC 1.0, 2.0 compatible
- Table creation at deploy time
- Flexible configurable datatypes
- Differential metadata
- Multiple `DataSources` support
- Full java object support
- Collections supported
- EJB-references supported
- Low admin overhead in automated mode

- Advanced table mapping
- Complex finders support
- Pre-defined mappings to 17 JDBC databases, including Oracle, SQLServer, DB2, Sybase, PointBase, Cloudscape, HypersonicSQL, PostgreSQL, MySQL, and more

JBossSX

JBossSX is a security service integration layer that supports both non-JAAS and JAAS based security implementations.

J2EE provides for a limited role based declarative security model, but does not specify how roles are obtained from the operation environment. This is an implementation detail left to the application server. JBossSX provides the standard J2EE security implementation layer and a great deal more. In our product security, you will find features that you won't find anywhere else—no matter how much you are willing to pay.

The key features of JBossSX are as follows:

- Secure authentication of users via JAAS login modules
- Extensible authentication of users via JAAS login modules
- Support for custom per method authentication of users via integration with the EJB container method interceptor
- Support for JAAS Subject based authorization of users
- Flexible mapping from legacy security systems to JAAS Subject based permissions

JBossCX

JBossCX is the JBoss-specific part of a JCA implementation; it takes care of handling resource adaptor deployment, interfacing to the JBoss transaction and security services, and making services available through JNDI. Essentially, it's plumbing.

The J2EE Connector Architecture (JCA) specifies how J2EE application components can access connection-based resources. The JCA will be a required part of J2EE 1.3.

The key features of JBossCX are as follows:

- Support for resource adapters that support local transactions, XA transactions, and for those that don't support transactions at all.

- Flexible resource principal mapping architecture. Currently the only principal mapping scheme implemented is to a single resource principal per connection factory.
- Support for basic password credentials.
- Resource adapters can be automatically deployed by placing the resource adaptor archive (RAR) into the JBoss deployment directory, just like other J2EE components.

Web Servers

The 2.4.x version of JBoss includes two services for Web containers — a Tomcat 4.0.3 service, and a Jetty 4.x service. Using one of these Web services requires non-trivial modification of the JBoss configuration as well as the servlet engine. Because of this, we provide pre-configured bundles of JBoss/Tomcat and JBoss/Jetty that provide a complete J2EE compatible solution.

What this Book Covers

The primary focus of this book is the presentation of the standard JBoss components introduced above, 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. In addition, the final chapter presents a detailed discussion of building and deploying an enterprise application to help you master the details of packaging and deploying applications with JBoss.

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.

This version of the book covers JBoss-2.4.6. The changes made between version 2.4.6 and 2.4.4 are described in “Appendix E, Change Notes”. Any place that refers to the 2.4.5 version also applies to the 2.4.6 release as the difference between 2.4.5 and 2.4.6 was a single rollback.

1. Installing and Building the JBoss Server

Installing the binary distribution and building a distribution from the CVS source code

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 2.4.x. You will learn how to use the binaries provided on the book CD, and how to obtain updated binaries from the JBoss Web site, install the binary, 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 2.4.6 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"  
  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-b24)  
  
Java HotSpot(TM) Client VM (build 1.3.1-b24, 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-2.4.5 directory. We'll look at the contents of this directory next.

Directory Structure

Installation of the JBoss distribution creates a JBoss-2.4.5 directory that contains server start scripts, jars, configuration files 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.

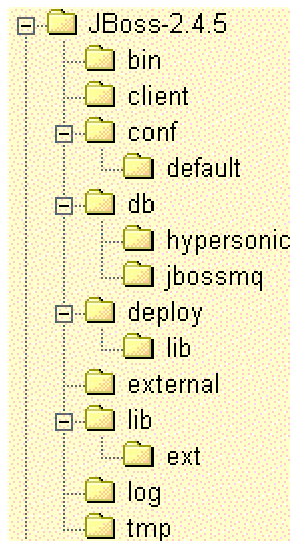


Figure 1-1, the directory structure of the JBoss distribution installation

Throughout the book we will refer to this directory as the JBOSS_DIST directory. The purposes of the various directories are discussed in Table 1-1.

Table 1-1, the JBoss server installation directories and descriptions

Directory	Description
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, 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.
conf	The JBoss configuration set(s) is located here. By default there is only one configuration set—default. Adding more than one configuration set is permitted. If you run JBoss bundled with a Web container (Tomcat or Jetty), a special configuration set is used (catalina or jetty). The key configuration files contained in the default configuration set will be discussed in more detail in the following section,

	“Configuration Files.”
lib	This directory contains jars used by JBoss that must be on the system classpath due to class loading issues. Classes in these jars are either loaded by code that does not use the thread context class loader, or fail to load code/resources using the thread context class loader.
lib/ext	This is the main jar directory. Any jar in this directory is loaded by the main JBoss classloader.
db	This is the directory that contains hypersonic and instantdb databases related files (configuration files, indexing tables, and so on) as well as JBossMQ—JMS provider message queue files.
deploy	This is JBoss's deployment directory. Drop your jars here and they will be deployed automatically.
log	JBoss log files are located in this directory. File logging is turned on by default.
tmp	A working directory used by JBoss during deployment of content found in the deploy directory.

Configuration Files

The `conf` directory contains one or more configuration file sets. The default JBoss configuration file set is located in the `conf/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.

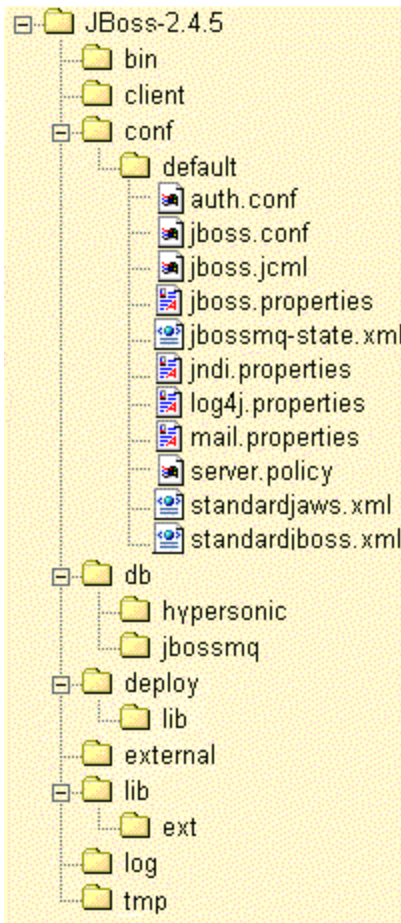


Figure 1-2, a view of the JBoss server default configuration set.

auth.conf

The `auth.conf` file is a JAAS login module configuration file as supported by the default `javax.security.auth.login.Configuration` implementation. It contains sample server side authentication configurations that are applicable when using JAAS based security. See Chapter 8, "JBossSX – The JBoss Security Extension Framework," for additional details on the JBoss security framework.

jboss.conf

Configuration file `jboss.conf` typically contains only those core service MBeans necessary to achieve the initial "bootstrap" of JBoss. These services include the classpath extension inclusion mechanism, logging, configuration service, and service control.

The `jboss.conf` file is loaded by an instance of the `javax.management.loading.MLet` class, and uses standard MLET syntax for JMX MBeans (Chapter 2, "JBoss Server Architecture

Overview", will discuss the details of the syntax). Any standard JMX MBean could be placed in the `jboss.conf` file as long as it does not depend on JBoss service Mbeans, such as naming. MBeans that do depend on JBoss service MBeans need to be configured in the `jboss.jcml` file so that startup dependencies can be controlled.

jboss.jcml

The `jboss.jcml` file lists all JMX MBeans (services) that are going to be included in a running instance of JBoss. Contrary to the JMX MLET file syntax, this file contains well-formed XML.

The need for deviation from MLET syntax is justified because MLET doesn't allow named parameters, but rather only TYPE-VALUE pairs. Having only TYPE-VALUE pairs leads to mismatching MBean parameter problems.

The `jboss.jcml` file is loaded by the `org.jboss.configuration.ConfigurationService` MBean. This service acts much like the standard JMX Mlet class in that it loads and configures MBeans. The dependencies between MBeans are managed by the `org.jboss.util.ServiceControl` MBean. The order of registration determines the order of initialization and startup. The ordering is based on the order in which MBeans are specified in the `jboss.jcml` file. For more details about creating an MBean with dependencies on other MBeans, see Chapter 2.

jboss.properties

`jboss.properties` is a standard Java Properties format file that is loaded into the System properties on startup of the JBoss server. System properties that are not required to be available prior to invocation of the JBoss server main method can be specified here. This file currently contains no properties, and it will be removed in the next release.

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. For additional details on its syntax, see Chapter 4, "JBossMQ - The JBoss Messaging Service".

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.

log4j.properties

The `log4j.properties` file configures the Apache log4j framework category priorities and appenders used by the JBoss server code.

mail.properties

JBoss provides `javax.mail.Session` mail resources for use with the JavaMail APIs. This file specifies mail provider properties, such as where to find SMTP servers, POP servers, and other mail related configuration information.

You are allowed to have multiple mail configurations by using multiple `mail.properties` files. All you have to do is to specify additional `MailService` mbeans with different configuration file attributes in your `jboss.jcml` file. See the `MailService` discussion in Chapter 2, "JBoss Server Architecture Overview" for additional details.

standardjaws.xml

The `standardjaws.xml` represents a default configuration file for JBossCMP engine. It contains the JNDI name of a default `DataSource`, per database JDBC-to-SQL mappings, default CMP entity beans settings, and so on. See Chapter 5, "JBossCMP – The JBoss Container Managed Persistence Layer" for additional details.

standardjboss.xml

The `standardjboss.xml` file provides the default container configurations. Use of this file is an advanced topic covered in Chapter 9, "Advanced JBoss configuration using `jboss.xml`".

Testing the Installation

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-2.4.5/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_CLASSPATH=:run.jar:../lib/crimson.jar
jboss.home = /tmp/JBoss-2.4.5
Using JAAS LoginConfig: file:/tmp/JBoss-2.4.5-beta/conf/default/auth.conf
JBoss release: JBoss-2.4.5 CVSTag=JBoss_2_4_5
JBoss version: 2.4.5.2002-05-10 02:02:17 CDT
Using configuration "default"
```

```
[INFO,root] Started Log4jService, config=file:/tmp/JBoss-
2.4.5/conf/default/log4j.properties
[INFO,Info] Java version: 1.3.1_01,Sun Microsystems Inc.
[INFO,Info] Java VM: Java HotSpot(TM) Server VM 1.3.1_01,Sun Microsystems Inc.
[INFO,Info] System: Linux 2.4.7-10,i386
[INFO,Shutdown] Shutdown hook added
[INFO,ServiceControl] Initializing 46 MBeans. . .
[INFO,ServiceControl] Started 46 services
[INFO,STDERR] JBoss-2.4.5 Started in 0m:5s.640
```

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.

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. You can build the 2.4.5 release of the core server and the various JBossXX modules discussed in this book, as well as the Tomcat service, but you first need to access the SourceForge CVS using the cvs program.

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 38,000 Open Source projects and over 400,000 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:

```
cvsc -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss login
cvsc -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/>.

Understanding the JBoss CVS Modules

There are a large number of JBoss-related CVS modules. Not all modules are relevant to the 2.4.x version of JBoss – some are obsolete, and others are for future versions of JBoss. This book covers those listed in Table 1-2.

Table 1-2, JBoss CVS module names and information

CVS Modulename	Description
----------------	-------------

jboss	The main JBossServer + JBossTX code module
jnp	The JBossNS code module
jbossmq	The JBossMQ code module
jbossctx	The JBossCX code module
jboss pooling	The JBossCX pooling code module
jbossctx	The JBossSX code module
contrib/jetty	The JBoss/Jetty-4.x integration module
contrib/catalina	The JBoss/Tomcat-4.x integration module
jboss-j2ee	The J2EE interface API code
jboss test	The JBoss unit test suite code

The CVS Modulename column gives the *modulename* value to use in the previous CVS syntax example. At one time, each CVS module was a standalone module that could be built independent of any other. This was perceived as useful early on because it allowed developers of the modules to work independently. However, the independence was achieved by copying snapshots of compiled code from the jboss module and any other module that was required for building. These code snapshots tended to get out of date and when it came time to build a complete release, there were integration problems due to inconsistencies between the independent builds. Today all CVS modules depend on the jboss CVS module for the code they need to build. So, for example, to update and build the jbossctx CVS module, one must check out both the jboss and jbossctx CVS modules. Likewise, to completely rebuild a JBoss distribution from all of the CVS module code one must check out all CVS modules and build the dependent modules against the jboss module. This process has been automated using a master Ant build script, and the build process will be in the next section.

Building the JBoss-2.4.5 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 modules, and move into the newly created directory. This directory is referred to as the CVS_WD directory for CVS working directory. The examples in this book will check out code into a /tmp/2.4.5 directory on a Linux system. Next, obtain the build.xml Ant script for the 2.4.5

version of the jboss module. To checkout the correct version of the build.xml script into your directory, perform the `cvs co` command and pass in the version tag of the code you want to access. To access the 2.4.5 version of JBoss, use the tag `JBoss_2_4_5` as shown here:

```
[starksm@banshee starksm]$ cd /tmp/2.4.5/
[starksm@banshee 2.4.5]$ cvs co -r JBoss_2_4_5 jboss/build.xml
U jboss/build.xml
```

The resulting `jboss/build.xml` file is an Ant script that automates the check out of the required CVS modules and the build commands required to build a the complete JBoss distribution. If you want to check out the latest 2.4 branch code to test fixes that will be applied to the next 2.4.x release, you would use a tag name of `Branch_2_4` rather than `JBoss_2_4_5` shown in the previous example.

To execute the build script you must have Ant version 1.4.1 installed. A tutorial on obtaining, installing and using Ant can be found in “Appendix D, Tools and Examples”. To build the JBoss 2.4.5 distribution, copy the `jboss/build.xml` file to your `CVS_WD` and then execute the ant command with no arguments as shown in Listing 1-1. Note that the output of the build process has been truncated in Listing 1-1 to show only the key events, and ‘#n’ bold annotations have been added at the start of the key event lines so they can be referenced for discussion.

Listing 1-1, the JBoss 2.4.x branch build process

```
[starksm@banshee 2.4.5]$ cp jboss/build.xml .
[starksm@banshee 2.4.5]$ ant

Buildfile: build.xml

cvs-co:

#1 do-cvs:
    [echo] Logging in as user anonymous
    [echo] Checking out JBossServer
    [echo] Checking out JBossNS
    [echo] Checking out JBossSX
    [echo] Checking out JBossMQ
    [echo] Checking out JBossCX
    [echo] Checking out JBossPool
    [echo] Checking out JBossJ2EE
    [echo] Checking out Catalina
    [echo] Checking out Jetty
    [echo] Checking out JBossTest
build:

#2 build-jboss:
    [echo] +++ Building JBossServer(module=jboss) for compilation
```

```

compile:
  [mkdir] Created dir: /tmp/2.4.5/jboss/build/classes
  [javac] Compiling 408 source files to /tmp/2.4.5/jboss/build/classes
...
main:
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/bin
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/lib/ext
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/db
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/db/hypersonic
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/deploy
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/deploy/lib
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/log
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/db/jbossmq
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/conf
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/client
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/tmp
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/admin/client/lib
  [mkdir] Created dir: /tmp/2.4.5/jboss/dist/admin/components
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/deploy
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/db/hypersonic
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/db/jbossmq
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/lib
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/log
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/db
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/tmp
  [copy] Copying 15 files to /tmp/2.4.5/jboss/dist/conf
  [copy] Copying 4 files to /tmp/2.4.5/jboss/dist/bin
  [copy] Copying 5 files to /tmp/2.4.5/jboss/dist/lib
  [copy] Copying 25 files to /tmp/2.4.5/jboss/dist/lib/ext
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/bin
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/lib/ext
  [copy] Copying 4 files to /tmp/2.4.5/jboss/dist/client
  [copy] Copying 13 files to /tmp/2.4.5/jboss/dist/client
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/client
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/admin
  [copy] Copying 4 files to /tmp/2.4.5/jboss/dist/admin/client
  [copy] Copying 5 files to /tmp/2.4.5/jboss/dist/admin/client/lib
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/deploy/lib
  [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/deploy/lib

#3 build-jbossj2ee:
  [echo] +++ Building JBoss-J2EE(module=jboss-j2ee)
...
compile:
  [mkdir] Created dir: /tmp/2.4.5/jboss-j2ee/build/classes
  [javac] Compiling 154 source files to /tmp/2.4.5/jboss-j2ee/build/classes
...
src-install:
  [copy] Copying 1 file to /tmp/2.4.5/jboss/src/client
  [copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib

```

```

[copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib

#4 build-jbossns:
[echo] +++ Building JBossNS(module=jnp)

...
compile:
[mkdir] Created dir: /tmp/2.4.5/jnp/build/classes
[javac] Compiling 12 source files to /tmp/2.4.5/jnp/build/classes
...
src-install:
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/client
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib

#5 build-jbosssx:
[echo] +++ Building JBossSX(module=jbosssx)

...
compile:
[javac] Compiling 11 source files to /tmp/2.4.5/jbosssx/build/classes
[javac] Compiling 67 source files to /tmp/2.4.5/jbosssx/build/classes
...
src-install:
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/client
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib

#6 build-jbosscx:
[echo] +++ Building JBossCX(module=jbosscx)

...
compile:
[javac] Compiling 17 source files to /tmp/2.4.5/jbosscx/build/classes
...
src-install:
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib
[echo] +++ Building JBossCX(module=jbosspool)
...
compile:
[mkdir] Created dir: /tmp/2.4.5/jbosspool/build/classes
[javac] Compiling 53 source files to /tmp/2.4.5/jbosspool/build/classes
...
src-install:
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib
[copy] Copying 1 file to /tmp/2.4.5/jboss/src/etc/deploy

#7 build-jbossmq:
[echo] +++ Building JBossMQ(module=jbossmq)

...
compile:
[mkdir] Created dir: /tmp/2.4.5/jbossmq/build/classes
[javac] Compiling 145 source files to /tmp/2.4.5/jbossmq/build/classes

```

```

...
src-install:
    [copy] Copying 1 file to /tmp/2.4.5/jboss/src/client
    [copy] Copying 1 file to /tmp/2.4.5/jboss/src/lib

#8 build-jbossdist:
    [echo] +++ Rebuilding JBossServer(module=jboss)

...
main:
    [copy] Copying 2 files to /tmp/2.4.5/jboss/dist/lib
    [copy] Copying 6 files to /tmp/2.4.5/jboss/dist/lib/ext
    [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/bin
    [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/lib/ext
    [copy] Copying 3 files to /tmp/2.4.5/jboss/dist/client
    [copy] Copying 4 files to /tmp/2.4.5/jboss/dist/client
    [copy] Copying 4 files to /tmp/2.4.5/jboss/dist/admin/client/lib
    [copy] Copying 1 file to /tmp/2.4.5/jboss/dist/deploy/lib

dist:

...

#9 dist-zip:
    [mkdir] Created dir: /tmp/2.4.5/jboss/zip/JBoss-2.4.5.RC3
    [copy] Copying 1139 files to /tmp/2.4.5/jboss/zip/JBoss-2.4.5.RC3
    [copy] Copied 1 empty directory to /tmp/2.4.5/jboss/zip/JBoss-2.4.5.RC3
    [zip] Building zip: /tmp/2.4.5/jboss/JBoss-2.4.5.RC3.zip

#10 tomcat4x-dist:

#11 jetty-dist:

BUILD SUCCESSFUL

Total time: 6 minutes 40 seconds

```

The key events are as follows:

1. The **do-cvs** task performs a check out of all the required JBoss source code from the SourceForge CVS repository using the anonymous login. The echo messages indicate the source code modules that are retrieved.
2. The **build-jboss** task performs a compilation of the JBossServer code and creates an initial distribution directory structure. This build must be performed first because the other JBossXX components require a JBoss distribution for their compilation.

3. The `build-jbossj2ee` task creates the `jboss-j2ee.jar` that contains the standard J2EE interfaces required by classes which use any of the J2EE interfaces. The `src-install` task which follows, installs the jars from the `build-jbossj2ee` task into the `jboss` CVS module source tree. This is the common pattern that all modules dependent on the `jboss` module use. They compile against the core `jboss` module classes and then place their component jars into `jboss` module source tree.
4. The `build-jbossns` task creates the `jnpserver.jar` and `jnp-client.jar` files for the JBossNS component. These jars are placed into the `jboss` module source tree by the `src-install` step.
5. The `build-jbosssx` task creates the `jbosssx.jar`, `jbosssx-client.jar` and `jboss-jaas.jar` files for the JBossSX component. These jars are placed into the `jboss` module source tree by the `src-install` step.
6. The `build-jbosscx` task creates the `jbosscx.jar`, `jbosspool.jar` and `jbosspool-jdbc.rar` files for the JBossCX component. The JBossCX component spans two CVS modules due to the historic fact that two different developers contributed distinct elements. The JBossCX jars are placed into the `jboss` module source tree by the `src-install` step.
7. The `build-jbossmq`
8. The second run of the `build-jbossdist` task copies the updated JBossXX component jars that have been updated in the `jboss` module source tree into the distribution tree.
9. The `dist-zip` task creates a zip archive of the distribution tree that was produced by step 8. This is the archive that is uploaded to SourceForge for distribution.
10. The `tomcat4x-dist` task creates a JBoss/Tomcat-4.x bundle that contains a custom integration service for the Tomcat-4.x series of servlet containers. This task did not produce a bundle in this execution because a `jakarta-tomcat-4.x` binary distribution was not available. The procedure for creating a JBoss/Tomcat-4.x bundle is similar to the JBoss-2.4.5/Tomcat-4.0.3 bundle procedure you will see in the next section.
11. The `jetty-dist` task creates a JBoss/Jetty-4.x bundle that contains a custom integration service for the Jetty-4.x series of servlet containers. This task did not produce a bundle in this execution because the Jetty distribution was not available.

The ultimate goal of this build process is the `jboss/JBoss-2.4.5.zip` archive that represents the complete JBoss 2.4.5 distribution. A distribution that bundled the 4.x series of Tomcat servlet container would have also resulted had you placed the appropriate Tomcat distribution into your `CVS_WD` directory. You will be lead through the extra step required to create the 2.4.5/Tomcat-4.0.3 bundle next.

Building the JBoss-2.4.5/Tomcat-4.0.3 Integrated Bundle Using the CVS Source Code

The JBoss-2.4.5 distribution you built in the previous section does not contain a servlet container. A common requirement for a JBoss installation is to have a servlet container to allow handling of web content like HTML pages, JSP pages and servlets. In this section you will create a JBoss-2.4.5/Tomcat-4.0.3 distribution bundle using the `build.xml` script from the preceding section. The only thing that prevented the build of the JBoss-2.4.5/Tomcat-4.0.3 distribution bundle during the previous build process was the fact that a Tomcat-4.0.3 distribution was not available during the build. To enable the build of this distribution, you need to place the jakarta-Tomcat-4.0.3 binary distribution directory into your `CVS_WD`. The Jakarta-Tomcat-4.0.3 binary can be obtained from the Apache site here <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.3/bin/jakarta-tomcat-4.0.3.zip>. You can then rerun the `build.xml` script to build the JBoss/Tomcat distribution bundle. Listing 1-2 demonstrates the build of the distribution using the `build.xml` Ant script. Only the tail portion of the output that begins with the `tomcat3x-dist` event line is shown.

Listing 1-2, the build of the JBoss/Tomcat distribution

```
[starksm@banshee 2.4.5]$ ant
Buildfile: build.xml
...
tomcat4x-dist:
    [echo] +++ Building JBoss/Catalina bundle(module=contrib/catalina)
Overriding previous definition of reference to catalina.path
...

bundle:
    [mkdir] Created dir: /tmp/2.4.5/contrib/catalina/bundle/JBoss-2.4.5_Tomcat-4.0.3
    [copy] Copying 946 files to /tmp/2.4.5/contrib/catalina/bundle/JBoss-
2.4.5_Tomcat-4.0.3/catalina
    [copy] Copied 5 empty directories to /tmp/2.4.5/contrib/catalina/bundle/JBoss-
2.4.5_Tomcat-4.0.3/catalina
    [copy] Copying 1139 files to /tmp/2.4.5/contrib/catalina/bundle/JBoss-
2.4.5_Tomcat-4.0.3/jboss
    [copy] Copied 1 empty directory to /tmp/2.4.5/contrib/catalina/bundle/JBoss-
2.4.5_Tomcat-4.0.3/jboss
```

```
[copy] Copying 1 file to /tmp/2.4.5/contrib/catalina/bundle/JBoss-2.4.5_Tomcat-4.0.3/jboss/lib/ext
[copy] Copying 14 files to /tmp/2.4.5/contrib/catalina/bundle/JBoss-2.4.5_Tomcat-4.0.3/jboss/conf/catalina
[echo] Adding Catalina MLETs to jboss.conf
[echo] Setting EmbeddedCatalinaSX as WAR Deployer
[echo] Adding EmbeddedCatalinaSX mbean to jboss.jcml

BUILD SUCCESSFUL

Total time: 14 seconds
```

The build output has been truncated to show only the final bundle task which creates the JBoss/Tomcat bundle. This is the bundle distribution directory that contains the JBoss/Tomcat bundle. This directory need simply be zipped up into an archive and it is equivalent to the archive available from the SourceForge download page.

Summary

This chapter covered how to obtain, install, and navigate the JBoss binary distribution. You learned how to obtain the source code for any version of JBoss from the CVS repository at SourceForge. You also learned how to build a distribution from the source using the Ant build.xml script included in the jboss CVS module. With these skills, you can obtain, build, run, and debug any version of JBoss that you use.

Now you are ready to begin learning the details of the various JBoss components highlighted in the introduction. The starting point will be the JBossServer component, which is the heart of the JBoss architecture.

2. JBoss Server Architecture Overview

JBossServer version 2.x architecture sets a new standard for both modular, plug-in design and ease of server and application management.

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 JBossServer version 2.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 in support of 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 the JBoss MBean service notion that adds life cycle operations to the basic JMX management component. You will then be given a through introduction to the EJB container architecture and its extensible plug-in design..

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.

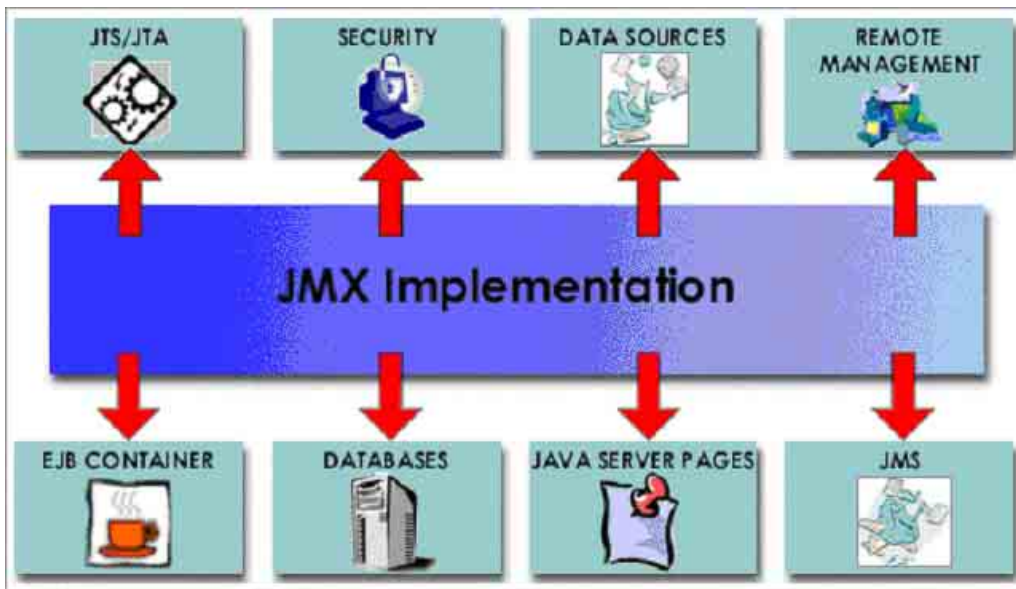


Figure 2-1, The JBoss JMX integration bus and the standard JBoss components.

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 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

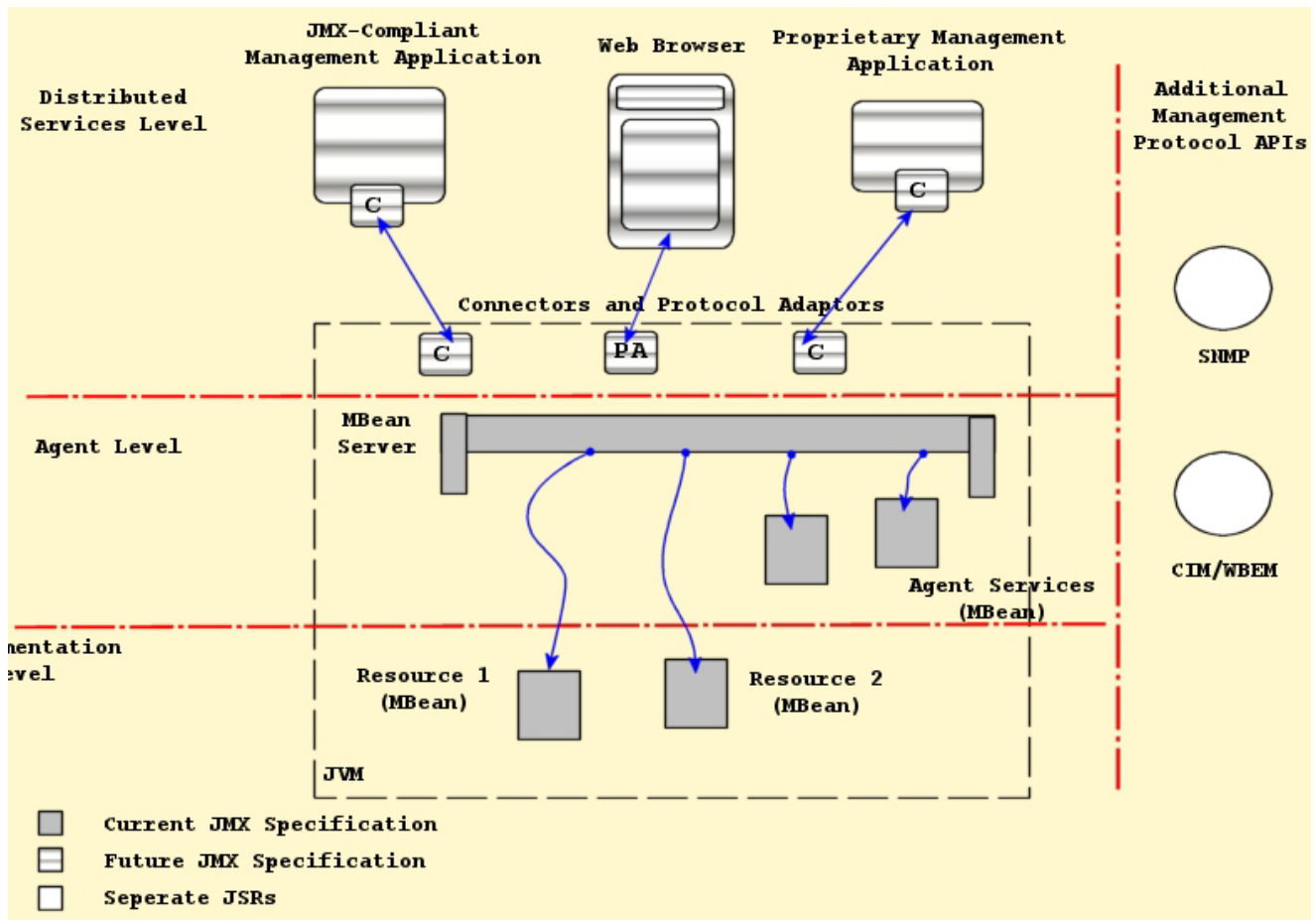


Figure 2-2, The Relationship between the components of the JMX architecture

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" later in this chapter.

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" section later 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.

The Sun reference implementation of the JMX agent is a set of Java classes that provide an implementation of an MBeanServer and all of the supporting agent services, and is available on the JavaSoft Web site at <http://java.sun.com/products/JavaManagement/download.html>.

IBM also provides an implementation of the JMX specification that is available from their alphaWorks Web site at <http://www.alphaworks.ibm.com/tech/TMX4j>.

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 Figure 2-2 by the component boxes with the horizontal lines. 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, writeable, 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.

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 only directly makes use of the dynamic class loading M-Let service.

The dynamic loading M-Let service

This section introduces the capabilities and configuration syntax of the JMX dynamic loading service. The M-Let service provides the capability to retrieve and instantiate MBeans from a remote location specified by a URL.

The M-Let service allows you to instantiate and register in the MBean server one or more MBeans located among a listing of URLs. The MBeans to be loaded are specified in a text based configuration file that uses an XML-like syntax with each MBean specified using an MLET tag. When an M-Let configuration file is loaded, all classes specified in MLET tags are downloaded, and an instance of each MBean specified in the file is created and registered.

The MLet Configuration File

The M-Let service can load a text file known as an MLET configuration file. The M-Let file may contain any number of MLET tags, each for instantiating a different MBean in a JMX agent. The MLET tag has the following syntax shown in Listing 2-1::

Listing 2-1, MLET configuration file syntax

```
<MLET
CODE = class | OBJECT = serfile
ARCHIVE = "archivelist"
[CODEBASE = codebaseURL]
[NAME = MBeanName]
[VERSION = version]
>
[arglist]
</MLET>
```

The attributes of the MLET tag are:

- **CODE = class**
This attribute specifies the fully qualified class name of the MBean to create. The class file of the MBean must be contained in one of the JAR files specified by the ARCHIVE attribute. Either the CODE or the OBJECT attribute must be present.
- **OBJECT = serfile**
This attribute specifies the .ser file that contains a serialized representation of the MBean to create. This file must be contained in one of the JAR files specified by the ARCHIVE attribute. If the JAR file contains a directory hierarchy, this attribute must specify the path of the file within this hierarchy, otherwise a match will not be found.

- **ARCHIVE = archiveList**

This mandatory attribute specifies one or more JAR files containing MBeans or other resources used by the MBean to be obtained. One of the JAR files must contain the file specified by the CODE or OBJECT attribute. If archivelist contains more than one file:

- Each file must be separated from the one that follows it by a comma (,).
- The whole list must be enclosed in double quote marks ("").

All JAR files in the archive list must be stored in the directory specified by the code base URL, or in the same directory as the m-let file that is the default code base when none is given.

- **CODEBASE = codebaseURL**

This optional attribute specifies the code base URL of the MBean to create. It identifies the directory that contains the JAR files specified by the ARCHIVE attribute. This attribute is used when the JAR files are not in the same directory as the m-let configuration file. If this attribute is not specified, the base URL of the m-let file is taken as the code base URL.

- **NAME = MBeanName**

This optional attribute specifies the string format of an object name to be assigned to the MBean instance when the MLet service registers it in the MBean server.

- **VERSION = version**

This optional attribute specifies the version number of the MBean and associated JAR files to be obtained. This version number can be used to specify whether or not the JAR files need to be loaded from the server to update those already loaded by the m-let service. The version must be a series of non-negative decimal integers each separated by a dot (.), for example 3.0, 2.4.4.

- **arglist**

The optional contents of the MLET tag specify a list of one or more arguments to pass to the constructor of the MBean to be instantiated. The MLet service will look for a constructor with a signature that matches the order and types of the arguments specified in the arglist. Instantiating objects with a constructor other than the default constructor is limited to constructor arguments for which there is a string representation. Each item in the arglist corresponds to an argument in the constructor. Use the following syntax to specify the argList:

<ARG TYPE=argumentType VALUE=argumentValue>

where:

- argumentType is the fully qualified class name of the argument (for example java.lang.Integer)
- argumentValue is the string representation of the value of the argument

The classes that make up the m-let service are found in the javax.management.loading package. The MLet class is a standard MBean that implements the MLetMBean interface. The MLet class also extends java.net.URLClassLoader, meaning that MLet is class loader. Its primary operation is:

```
java.util.Set getMBeansFromURL(java.lang.String url)
    throws javax.management.ServiceNotFoundException
```

This loads an MLET configuration file that defines the MBeans to be added to the MBeanServer. The location of the file is specified by the url argument. The MBeans specified in the MLET file will be instantiated and registered by the MBean server. The Set returned contains one entry per MLET tag, and the type of entry specifies either the javax.management.ObjectInstance for the created MBean, or a java.lang.Throwable object for any error that prevented the MBean from being created. The JBoss server utilizes the getMBeansFromURL method during server startup to load the bootstrap JBoss MBean services. The complete JavaDoc for the MLet class API can be found in the reference implementation.

JMX Summary

This section introduced you to the basic concepts behind the JMX architecture. The various levels of the JMX specification and the associated components were presented. The focus of the introduction was on the managed bean components or MBeans, and the MLet MBean and its configuration file.

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 step following the creation of the MBean server is the creation of an MLet instance (javax.management.loading.MLet). These two steps are shown in the following code segment, Listing 2-2.

Listing 2-2, The setup of the MBeanServer and MLet on JBoss startup

```
// Create MBeanServer
MBeanServer server = MBeanServerFactory.createMBeanServer();
// Add configuration directory to MLet classpath
URL confDir = File("../conf/"+confName).toURL();
URL confURL = confDir.toURL();
URL[] urls = {confURL};
// Create MLet
MLet mlet = new MLet(urls);
ObjectName domain = server.getDefaultDomain();
ObjectName mletName = new ObjectName(domain, "service", "MLet");
server.registerMBean(mlet, mletName);
// Set MLet as classloader for this app
Thread.currentThread().setContextClassLoader(mlet);
```

So, the JBoss server's initial state consists of a JMX MBean server and an MLet class loader whose classpath contains only the configuration file set directory that was specified on the command line.

JBoss now loads its configured components by first using the JMX MLet to load a bootstrap MLET configuration file called jboss.conf. This invokes a JBoss specific configuration service MBean that loads MBeans from a custom configuration file called jboss.jcml. Finally, this initializes all MBeans services defined in the jboss.jcml file. This sequence of steps is performed by the code fragment given in Listing 2-3.

Listing 2-3, Using JMX to load the JBoss configured components

```
// Load bootstrap configuration
URL mletConf = mlet.getResource("jboss.conf");
Set beans = (Set) mlet.getMBeansFromURL(mletConf);
// Load JBoss configuration
ObjectName cfgName = new ObjectName(":service=Configuration");
Object[] args = {};
String[] sig = {};
server.invoke(cfgName, "loadConfiguration", args, sig);
// Init and Start MBeans
ObjectName scName = new ObjectName(":service=ServiceControl");
server.invoke(scName, "init", args, sig);
server.invoke(scName, "start", args, sig);
```

There are two reasons for splitting the JBoss components loading into a bootstrap and final step. First, the MLET configuration file syntax is rather cryptic and so editing it is prone to errors. Second, there is no notion of MBean service dependency specification in the MLET configuration file. Dependency means that one MBean service needs other MBean services to function correctly. Because of these issues, a JBoss specific MBean XML configuration file named jboss.jcml was created as well as an

MBean([org.jboss.configuration.ConfigurationService](#)) to read it. The DTD for the [jboss.jcml](#) file is given in Figure 2-3.

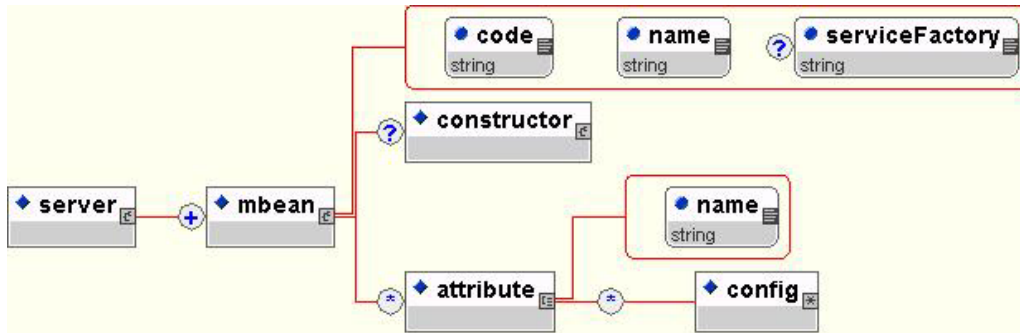


Figure 2-3, The DTD structure for the [jboss.jcml](#) configuration file

The [jboss.jcml](#) file allows for more readable configuration entries since each attribute of an mbean element is named as illustrated by the following [jboss.jcml](#) fragment in Listing 2-4:

Listing 2-4, A sample [jboss.jcml](#) MBean declaration

```
<mbean code="org.jboss.jdbc.HypersonicDatabase"
  name="DefaultDomain:service=Hypersonic">
  <attribute name="Port">1476</attribute>
  <attribute name="Silent">true</attribute>
  <attribute name="Database">default</attribute>
  <attribute name="Trace">false</attribute>
</mbean>
```

The corresponding MLET tag would be as shown in Listing 2-5:

Listing 2-5, The corresponding MLET tag for Listing 2.4

```
<MLET CODE=" org.jboss.jdbc.HypersonicDatabase"
  NAME="DefaultDomain:service=Hypersonic">
<ARG TYPE="java.lang.Integer" VALUE="1476">
<ARG TYPE="java.lang.Boolean" VALUE="true">
<ARG TYPE="java.lang.String" VALUE="default">
<ARG TYPE="java.lang.Boolean" VALUE="false">
</MLET>
```

The [jboss.jcml](#) version is easier to read and maintain because you don't have to know the order of the corresponding MBean constructor arguments. The other feature of the [ConfigurationService](#) MBean is that the order of the mbean element declarations in the [jboss.jcml](#) file defines the MBean dependencies. As each mbean element is read by the [ConfigurationService](#), it is registered with a JBoss life cycle MBean service ([org.jboss.util.ServiceControl](#)). The order in which MBeans are registered with the [ServiceControl](#) MBean defines the order in which the services are initialized and started.

This is also the reverse of the order in which services are stopped on shutdown of the JBoss server. Thus, if you need add a new MBean declaration to the `jboss.jcml` file, and the MBean needs the JNDI naming service MBean, you would add the MBean declaration anywhere after the naming service MBean declaration.

The JBoss server starts out as nothing more than a container for the JMX MBean server, and then loads its personality based on the `jboss.conf` and `jboss.jcml` MBean configuration files 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.

Writing 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 their MLET tag to the `jboss.conf` file, or equivalently adding an mbean tag anywhere to the `jboss.jcml` 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 initialize, start, stop, and destroy itself. The management of the MBean service life cycle is the responsibility of two JBoss MBean services, ConfigurationService and ServiceControl.

The ConfigurationService MBean

JBoss manages the configuration 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.configuration.ConfigurationService` class. The ConfigurationService MBean is loaded when JBoss starts up by the JMX MLet and loads the `jboss.conf` file. This is because the ConfigurationService MBean is declared in the `jboss.conf` file that ships with any JBoss distribution. The ConfigurationService MBean is a bootstrap service that has no dependencies on other JBoss provided MBeans. After the `jboss.conf` file is loaded to create the bootstrap MBeans, the `jboss.jcml` configuration is loaded by invoking `loadConfiguration` method on the ConfigurationService MBean. The `loadConfiguration` method performs the following steps:

1. Load the `jboss.jcml` file as a resource using the current `Thread` context `ClassLoader`.
2. Parse the `jboss.jcml` file and instantiate all MBeans that the file contains
3. Apply the attribute settings from the `jboss.jcml` file to each MBean
4. Register each MBean with the JBoss `ServiceControl` MBean so that the `ServiceControl` MBean will manage MBean service's life cycle. If the MBean does not implement the `org.jboss.util.Service` interface, the MBean is wrapped in a proxy implementation of the `Service` interface that delegates any of the methods of the `Service` interface to the matching MBean methods.

The Service life-cycle interface

The JMX specification does not define any type of life cycle or dependency management aspect for MBeans. The JBoss `ConfigurationService` and `ServiceControl` MBeans do introduce 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 2-6 shows the `org.jboss.util.Service` interface:

Listing 2-6, The `org.jboss.util.Service` interface

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

The `ServiceControl` MBean invokes the methods of the `Service` interface at the appropriate times of the service life-cycle. We'll discuss the methods in more detail in the `ServiceControl` 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 life-cycle notion. When this standard is finalized JBoss will likely move to a JSR 77 based service life-cycle implementation.

The ServiceControl MBean

JBoss manages dependencies between MBeans via the `org.jboss.util.ServiceControl` custom MBean. The `ServiceControl` MBean is another bootstrap MBean loaded by the JMX `MLet` from the `jboss.conf` file on startup of the JBoss server. The `ServiceControl` MBean is a simple MBean that contains an ordered collection of Service interface implementations. After the `ConfigurationService` loads an MBean from the `jboss.jcml` configuration file, it populates the MBean with the MBean's configured attributes, and then the MBean is registered with the `ServiceControl` MBean by invoking the `register` method on `ServiceControl` with the MBean or its Service proxy as the argument. The order in which services are registered with the `ServiceControl` MBean defines the dependency ordering between the services. That is to say, the order in which services are registered with the `ServiceControl` MBean defines the order in which the services are initialized and started.

The `ServiceControl` MBean has four key methods: `init`, `start`, `stop` and `destroy`. These methods correspond to the `Service` interface life cycle methods.

The init method

The JBoss server main entry point calls the `ServiceControl` `init` method after the `jboss.jcml` configuration has been loaded. At the point the `ServiceControl` `init` method is called, the `ConfigurationService` MBean has registered all MBeans defined in the `jboss.jcml` file with the `ServiceControl` MBean. The `init` method makes a copy of current list of `Service` instances and then proceeds to invoke the `init` method on each instance.

The order of initialization is the order of registration, which is the same as the ordering of mbean element entries in the `jboss.jcml` file. When a service's `init` method is called, all services that were registered ahead of it have also had their `init` method invoked. This gives an MBean an opportunity to check that required MBeans or resources exist. The service typically 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 `init` in favor of just the `start` method because that is the first point at which the service can be fully functional.

The start method

The JBoss server main entry point calls the `ServiceControl` `start` method after the `init` method has returned. The `start` method makes a copy of current list of `Service` instances and then proceeds to invoke the `start` method on each instance. When a service's `start` method is called, all services that were registered ahead of it 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 initialized and started if possible.

The stop method

The stop method is invoked by the JBoss server shutdown process, which is managed by the org.jboss.util.Shutdown MBean. The stop method makes a copy of the current list of service instances and then invokes the stop method on each service in reverse order from that of the init and start methods. Thus, services that were last to start are the first to be stopped.

The destroy method

The destroy method is invoked by the JBoss server shutdown process after the stop method. The destroy method makes a copy of the current list of service instances and then invokes the destroy method on each service in reverse order from that of the init and start methods. 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.

Writing JBoss MBean services

Writing a custom MBean service that integrates into the JBoss server requires the use of the org.jboss.util.Service interface pattern if the custom service is dependent on other JBoss services. When a custom MBean depends on other MBean services you cannot perform any JBoss service dependent initialization in any of the javax.management.MBeanRegistration interface methods. Instead, you must do this in the Service interface init and/or start methods. You can do this in 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.util.Service interface.
- Have your MBean interface extend the org.jboss.util.ServiceMBean interface. This is a subinterface of org.jboss.util.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 mind dependencies on JBoss classes, the simplest approach is to have your MBean interface extend from org.jboss.util.ServiceMBean and your MBean implementation class extend from the abstract org.jboss.util.ServiceMBeanSupport class. This class implements the org.jboss.util.ServiceMBean interface except for the String getName() method. ServiceMBeanSupport provides implementations of the init, start, stop, and destroy methods that integrate logging and JBoss service state management. Each method

delegates any subclass specific work to initService, startService, stopService, and destroyService methods respectively. When subclassing ServiceMBeanSupport, you would override one or more of the initService, startService, stopService, and destroyService methods in addition to getName as required

A simple 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 2-7. 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 2-7, 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.io.InputStream;
import java.util.HashMap;
import javax.naming.CompositeName;
import javax.naming.Context;
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 static Context createContext(Context rootCtx, Name name)
        throws NamingException
    {
        Context subctx = rootCtx;
        for(int n = 0; n < name.size(); n++)
        {
            String atom = name.get(n);
            try
            {
                Object obj = subctx.lookup(atom);
                subctx = (Context) obj;
            }
            catch(NamingException e)

```

```

        {
            // No binding exists, create a subcontext
            subctx = subctx.createSubcontext(atom);
        }
    }

    return subctx;
}

private void rebind() throws NamingException
{
    InitialContext rootCtx = new InitialContext();
    // Get the parent context into which we are to bind
    Name fullName = rootCtx.getNameParser("").parse(jndiName);
    System.out.println("fullName="+fullName);
    Name parentName = fullName;
    if( fullName.size() > 1 )
        parentName = fullName.getPrefix(fullName.size()-1);
    else
        parentName = new CompositeName();
    Context parentCtx = createContext(rootCtx, parentName);
    Name atomName = fullName.getSuffix(fullName.size()-1);
    String atom = atomName.get(0);
    NonSerializableFactory.rebind(parentCtx, atom, contextMap);
}

private void unbind(String jndiName)
{
    try
    {
        Context rootCtx = (Context) 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 2.8. 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.util.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 2-7 and Listing 2-8 are highlighted in bold in Listing 2-8.

Listing 2-8, 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.util.ServiceMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
}

package org.jboss.chap2.ex2;

// The JNDIMap MBean implementation
import java.io.InputStream;
import java.util.HashMap;
import javax.naming.CompositeName;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap extends org.jboss.util.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 String getName()
{
    return "JNDIMap(" + jndiName + ")";
}

public void startService() throws Exception
{
    rebind();
}

public void stopService()
{
    unbind(jndiName);
}

private static Context createContext(Context rootCtx, Name name)
    throws NamingException
{
    Context subctx = rootCtx;
    for(int n = 0; n < name.size(); n++)
    {
        String atom = name.get(n);
        try
        {
            {
                Object obj = subctx.lookup(atom);
                subctx = (Context) obj;
            }
            catch(NamingException e)
            {
                // No binding exists, create a subcontext
                subctx = subctx.createSubcontext(atom);
            }
        }
    }

    return subctx;
}

private void rebind() throws NamingException
{
    InitialContext rootCtx = new InitialContext();
    // Get the parent context into which we are to bind
    Name fullName = rootCtx.getNameParser("").parse(jndiName);
    log.debug("fullName="+fullName);
    Name parentName = fullName;
    if( fullName.size() > 1 )
        parentName = fullName.getPrefix(fullName.size()-1);
    else
        parentName = new CompositeName();
    Context parentCtx = createContext(rootCtx, parentName);
    Name atomName = fullName.getSuffix(fullName.size()-1);
    String atom = atomName.get(0);
    NonSerializableFactory.rebind(parentCtx, atom, contextMap);
}

private void unbind(String jndiName)
{

```

```

    try
    {
        Context rootCtx = (Context) new InitialContext();
        rootCtx.unbind(jndiName);
        NonSerializableFactory.unbind(jndiName);
    }
    catch(NamingException e)
    {
        log.error("Failed to unbind map", e);
    }
}
}

```

A sample `jboss.jcml` mbean entry for either implementation of the JNDIMap MBean is given in Listing 2.9 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 2-9, A sample `jboss.jcml` entry for the JNDIMap MBean and a client usage code fragment.

```

<!-- The jboss.jcml file -->
<server>
...
  <!-- The JNDI Naming service -->
  <mbean code="org.jboss.naming.NamingService"
    name="DefaultDomain:service=Naming">
    <attribute name="Port">1099</attribute>
  </mbean>

  <!-- Add the JNDIMap entry after the NamingService since the
    NamingService must be running in order for the JNDIMap
    bean to start.
  -->
  <mbean code="JNDIMap"
    name="DefaultDomain:service=JNDIMap,jndiName=inmemory/maps/MapTest">
    <attribute name="JndiName">inmemory/maps/MapTest</attribute>
  </mbean>
...
</server>

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

```

The Core JBoss MBeans

This section on JBoss and JMX concludes with an overview of the JBoss bootstrap MBeans defined in the `jboss.conf` file. The JBoss service MBeans defined in the `jboss.jcml` file as shipped in the standard JBoss 2.4.5 distribution will also be examined..

The bootstrap MBeans, jboss.conf

Listing 2-10 shows the default jboss.conf configuration file shipped with the standard JBoss server distribution. The listing shows only those entries that are not commented out.

Listing 2-10, the default jboss.conf bootstrap configuration file from the standard JBoss distribution

```
<!-- JBoss JMX Boot-strap Configuration -->

<!-- The log4j based logging service based on the conf
log4j.properties file -->
<MLET CODE = "org.jboss.logging.Log4jService"
  ARCHIVE="jboss.jar,log4j.jar" CODEBASE="../../lib/ext/">
</MLET>

<!-- The log dir needs to be in the classpath to allow
location of log.properties -->
<MLET CODE="org.jboss.util.ClassPathExtension" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
  <ARG TYPE="java.lang.String" VALUE="../../log/">
</MLET>
<!-- Place the lib/ext directory in the classpath -->
<MLET CODE="org.jboss.util.ClassPathExtension" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
  <ARG TYPE="java.lang.String" VALUE="."/>
</MLET>

<MLET CODE = "org.jboss.util.Info" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
</MLET>

<MLET CODE="org.jboss.util.ClassPathExtension" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
  <ARG TYPE="java.lang.String" VALUE="../../tmp/">
</MLET>

<MLET CODE="org.jboss.util.ClassPathExtension" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
  <ARG TYPE="java.lang.String" VALUE="../../db/">
</MLET>

<MLET CODE = "org.jboss.configuration.ConfigurationService"
  ARCHIVE="jboss.jar,..xml.jar" CODEBASE="../../lib/ext/">
</MLET>

<MLET CODE = "org.jboss.util.Shutdown" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
</MLET>

<MLET CODE = "org.jboss.util.ServiceControl" ARCHIVE="jboss.jar"
  CODEBASE="../../lib/ext/">
</MLET>
```

Notice that all MLET tags use the same CODEBASE attribute of “[../../lib/ext](#)”. The CODEBASE is relative to the configuration file set directory. The reason for this is that the URL passed to the MLet constructed by the JBoss startup code corresponds to the configuration file set directory, thus the configuration directory is in the MLet classpath and can be used as the starting point for relative paths.

Recall from Figure 1-1, the directory structure of the JBoss distribution installation, the installation directory structure diagram, that a configuration file set would be a directory like JBOSS_DIST/conf/default, and therefore, the CODEBASE refers to the JBOSS_DIST/lib/ext directory. This is the directory that contains all of the server jars that did not need to be on the system classpath. Similarly, all MLET tags include the jboss.jar in their ARCHIVE attribute. This is because all of the JBoss server MBean services are located in the jboss.jar archive.

Warning: Although the MLET tags in the jboss.conf file look like XML elements, they are not. Because of this, you cannot comment out entries by surrounding them in the XML begin and end comment elements `<!-- -->`. Rather, a commented out element has this pseudo XML comment form:

```
<!-- A commented MLET

-- MLET CODE = "..." ARCHIVE="..." CODEBASE="...">

-- ARG TYPE="..." VALUE="...">

-- /MLET>

-->
```

Each line within the XML comment elements must begin with “--”(double dashes)..

org.jboss.logging.Log4jService

The Log4jService MBean configures the Apache log4j system. JBoss uses the log4j framework as its internal logging API. The Log4jService can be used without any arguments passed to its constructor, in which case the service looks for a log4j.properties file as a resource using the Thread context ClassLoader. A second constructor allows for a single String parameter that specifies the path to the log4j configuration file. The file may be either a Java Properties file or a XML file ending with an .xml suffix. The Log4jService will choose either the org.apache.log4j.xml.DOMConfigurator or the org.apache.log4j.PropertyConfigurator parser depending on whether the configuration file is an XML document. The ARCHIVE attribute of the MLET tag specifies the log4j.jar archive in addition to the common jboss.jar. This is necessary because the Log4jService has explicit

references to log4j classes and so the log4j.jar must be included in the list of jars the MLet uses to completely load the service.

org.jboss.util.ClassPathExtension

The ClassPathExtension MBean provides the capability to augment classpath of the MLet that was constructed by the JBoss server startup code. The ClassPathExtension service takes a single argument that specifies the URL that should be added to the MLet classpath. If the URL is a file URL or a file path that ends in a '/' character and resolves to a directory, all jars found in the directory are added to the MLet classpath. If the file path is a relative path, it is resolved with respect to JBOSS_DIST/lib/ext directory.

Note that three different directories are added to the MLet classpath using relative paths. These resolve to the following directories:

- “./” resolves to JBOSS_DIST/lib/ext
- “../tmp” resolves to JBOSS_DIST/tmp
- “../db” resolves to JBOSS_DIST/db

The JBOSS_DIST/lib/ext directory reference places all of the jars in the directory into the MLet classpath. This directory contains the JBoss server jars and third party support jars. You will typically also place your common jars into lib/ext as well to make them available to MBeans and J2EE components.

The JBOSS_DIST/tmp and JBOSS_DIST/db directories are added to the MLet classpath to allow services that need to know the tmp and db directory locations to search for the directories as classpath resources.

org.jboss.util.Info

The Info MBean is a simple service that displays key information from the `java.lang.System` properties, such as VM vendor, VM version, OS name, and OS version as INFO priority log4j messages. It also dumps out the entire system Properties map as DEBUG priority log4j messages. Note that because this MBean used log4j messages, it technically depends on the Log4jService. Although JMX has not documented a dependency mechanism, MBeans are created in the order declared in the `jboss.conf` file for the Sun and IBM JMX implementations.

org.jboss.configuration.ConfigurationService

The ConfigurationService MBean is the JBoss MBean service loader that we have talked about in some detail. Typically, no constructor arguments are required to configure this

service. In particular note that the location of the `jboss.jcml` file is not specified. This file is located by the service as a resource on the classpath.

`org.jboss.util.Shutdown`

The Shutdown service allows one to shutdown the JBoss server in one of two ways. The first is through a shutdown operation that may be invoked directly or through the JMX server. The second is by virtue of the fact that the Shutdown service hooks into the `java.lang.Runtime ShutdownHook` so that you may Ctrl-C or send the equivalent OS signal to cause the server to shutdown cleanly during a `System.exit()`. The integration with the ShutdownHook mechanism is a JDK 1.3+ specific feature.

`org.jboss.util.ServiceControl`

The ServiceControl MBean is the service life cycle manager that was discussed earlier in this chapter in conjunction with the ConfigurationService. This service has no configurable attributes.

The standard MBean services, `jboss.jcml`

Listing 2-11 shows the default `jboss.jcml` configuration file shipped with the standard JBoss server distribution. The listing shows only those entries that are most commonly used.

Listing 2-11, the default `jboss.jcml` services configuration file from the standard JBoss distribution.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is where you can add and configure your MBeans
ATTENTION: The order of the listing here is the same order as
the MBeans are loaded. Therefore if a MBean depends on another
MBean to be loaded and started it has to be listed after all
the MBeans it depends on.
-->

<server>
  <!-- ===== -->
  <!-- Classloading -->
  <!-- ===== -->
  <mbean code="org.jboss.web.WebService" name="DefaultDomain:service=Webserver">
    <attribute name="Port">8083</attribute>
  </mbean>

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

```

<!-- ===== -->
<!-- Transactions -->
<!-- ===== -->
<mbean code="org.jboss.tm.TransactionManagerService"
name="DefaultDomain:service=TransactionManager">
    <attribute name="TransactionTimeout">300</attribute>

    <!-- Use this attribute if you need to use a specific Xid
        implementation
    <attribute name="XidClassName">oracle.jdbc.xa.OracleXid</attribute>
    -->
</mbean>

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

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

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

<!-- ===== -->
<!-- JDBC -->
<!-- ===== -->

<mbean code="org.jboss.jdbc.JdbcProvider" name="DefaultDomain:service=JdbcProvider">
    <attribute name="Drivers">org.hsql.jdbcDriver</attribute>
</mbean>

<mbean code="org.jboss.jdbc.HypersonicDatabase" name="DefaultDomain:service=Hypersonic">
    <attribute name="Port">1476</attribute>
    <attribute name="Silent">true</attribute>
    <attribute name="Database">default</attribute>
    <attribute name="Trace">false</attribute>
</mbean>

<mbean code="org.jboss.jdbc.XADataSourceLoader"
name="DefaultDomain:service=XADataSource,name=DefaultDS">
    <attribute name="PoolName">DefaultDS</attribute>
    <attribute
name="DataSourceClass">org.jboss.pool.jdbc.xa.wrapper.XADataSourceImpl</attribute>
    <attribute name="Properties"></attribute>
    <attribute name="URL">jdbc:HypersonicSQL:hsql://localhost:1476</attribute>
    <attribute name="GCMinIdleTime">1200000</attribute>
    <attribute name="JDBCUser">sa</attribute>

```

```

    <attribute name="MaxSize">10</attribute>
    <attribute name="Password" />
    <attribute name="GCEnabled">false</attribute>
    <attribute name="InvalidateOnError">false</attribute>
    <attribute name="TimestampUsed">false</attribute>
    <attribute name="Blocking">true</attribute>
    <attribute name="GCInterval">120000</attribute>
    <attribute name="IdleTimeout">1800000</attribute>
    <attribute name="IdleTimeoutEnabled">false</attribute>
    <attribute name="LoggingEnabled">false</attribute>
    <attribute name="MaxIdleTimeoutPercent">1.0</attribute>
    <attribute name="MinSize">0</attribute>
</mbean>

<!-- ===== -->
<!-- J2EE deployment -->
<!-- ===== -->

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

<!-- Uncomment to add embedded tomcat service
<mbean code="org.jboss.tomcat.EmbeddedTomcatServiceSX"
name="DefaultDomain:service=EmbeddedTomcat" />
-->

<!-- ===== -->
<!-- JBossMQ -->
<!-- ===== -->
<mbean code="org.jboss.mq.server.JBossMQService" name="JBossMQ:service=Server"/>

<!-- The StateManager is used to keep JMS perisitent state data. -->
<!-- For example: what durable subscriptions are active. -->
<mbean code="org.jboss.mq.server.StateManager" name="JBossMQ:service=StateManager">
    <attribute name="StateFile">jbossmq-state.xml</attribute>
</mbean>

<!-- The PersistenceManager is used to store messages to disk. -->
<mbean code="org.jboss.mq.pm.rollinglogged.PersistenceManager"
name="JBossMQ:service=PersistenceManager">
    <attribute name="DataDirectory">../../db/jbossmq</attribute>
</mbean>

<!-- InvocationLayers are the different transport methods that can be used to access the
server -->
<mbean code="org.jboss.mq.il.jvm.JVMServerILService"
name="JBossMQ:service=InvocationLayer,type=JVM">
    <attribute name="ConnectionFactoryJNDIRef">java:/ConnectionFactory</attribute>
    <attribute name="XAConnectionFactoryJNDIRef">java:/XAConnectionFactory</attribute>

```

```

</mbean>

<mbean code="org.jboss.mq.il.rmi.RMIServerILService"
name="JBossMQ:service=InvocationLayer,type=RMI">
  <attribute name="ConnectionFactoryJNDIRef">RMIServerConnectionFactory</attribute>
  <attribute name="XAConnectionFactoryJNDIRef">RMIXAConnectionFactory</attribute>
</mbean>

<mbean code="org.jboss.mq.il.oil.OILServerILService"
name="JBossMQ:service=InvocationLayer,type=OIL">
  <attribute name="ConnectionFactoryJNDIRef">ConnectionFactory</attribute>
  <attribute name="XAConnectionFactoryJNDIRef">XAConnectionFactory</attribute>
</mbean>

<mbean code="org.jboss.mq.il.uil.UILServerILService"
name="JBossMQ:service=InvocationLayer,type=UIL">
  <attribute name="ConnectionFactoryJNDIRef">UILConnectionFactory</attribute>
  <attribute name="XAConnectionFactoryJNDIRef">UILXAConnectionFactory</attribute>
</mbean>

<!-- The following three line create 3 topics named: testTopic, example, and bob -->
<mbean code="org.jboss.mq.server.TopicManager"
name="JBossMQ:service=Topic,name=testTopic"/>
<mbean code="org.jboss.mq.server.TopicManager"
name="JBossMQ:service=Topic,name=example"/>
<mbean code="org.jboss.mq.server.TopicManager" name="JBossMQ:service=Topic,name=bob"/>

<!-- The following 9 line create 9 topics named: testQueue, controlQueue, A, B, -->
<!-- C, D, E, F, and ex -->
<mbean code="org.jboss.mq.server.QueueManager"
name="JBossMQ:service=Queue,name=testQueue"/>
<mbean code="org.jboss.mq.server.QueueManager"
name="JBossMQ:service=Queue,name=controlQueue"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=A"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=B"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=C"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=D"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=E"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=F"/>
<mbean code="org.jboss.mq.server.QueueManager" name="JBossMQ:service=Queue,name=ex"/>

<!-- Used for backwards compatability with JBossMQ versions before 1.0.0 -->
<mbean code="org.jboss.naming.NamingAlias"
name="DefaultDomain:service=NamingAlias,fromName=QueueConnectionFactory">
  <attribute name="ToName">ConnectionFactory</attribute>
  <attribute name="FromName">QueueConnectionFactory</attribute>
</mbean>
<mbean code="org.jboss.naming.NamingAlias"
name="DefaultDomain:service=NamingAlias,fromName=TopicConnectionFactory">
  <attribute name="ToName">ConnectionFactory</attribute>
  <attribute name="FromName">TopicConnectionFactory</attribute>
</mbean>

<!-- For Message Driven Beans -->

```

```

<mbean code="org.jboss.jms.jndi.JMSProviderLoader"
name=":service=JMSProviderLoader,name=JBossMQProvider">
  <attribute name="ProviderName">DefaultJMSProvider</attribute>
  <attribute name="ProviderAdapterClass">org.jboss.jms.jndi.JBossMQProvider</attribute>
  <attribute name="QueueFactoryRef">java:/XAConnectionFactory</attribute>
  <attribute name="TopicFactoryRef">java:/XAConnectionFactory</attribute>
</mbean>
<mbean code="org.jboss.jms.asf.ServerSessionPoolLoader"
name=":service=ServerSessionPoolMBean,name=StdJMSPool">
  <attribute name="PoolName">StdJMSPool</attribute>
  <attribute
name="PoolFactoryClass">org.jboss.jms.asf.StdServerSessionPoolFactory</attribute>
</mbean>

<!-- Make sure you change EmbeddedTomcat to Jetty if you are using Jetty -->
<mbean code="org.jboss.deployment.J2eeDeployer" name="J2EE:service=J2eeDeployer">
  <attribute name="DeployerName">Default</attribute>
  <attribute name="JarDeployerName">:service=ContainerFactory</attribute>
  <attribute name="WarDeployerName">:service=EmbeddedTomcat</attribute>
</mbean>

<!-- ===== -->
<!-- JBossCX setup, for J2EE connector architecture support -->
<!-- ===== -->

<mbean code="org.jboss.resource.RARDeployer" name="JCA:service=RARDeployer">
</mbean>

<!-- Minerva no transaction connection manager factory.

      Use this for resource adapters that don't support
      transactions. -->
<mbean code="org.jboss.resource.ConnectionManagerFactoryLoader"
      name="JCA:service=ConnectionManagerFactoryLoader,name=MinervaNoTransCMFactory">
  <attribute name="FactoryName">MinervaNoTransCMFactory</attribute>
  <attribute name="FactoryClass">
    org.jboss.pool.connector.jboss.MinervaNoTransCMFactory
  </attribute>
  <attribute name="Properties"></attribute>
</mbean>

<!-- Minerva local transaction connection manager factory.
      Use this for resource adapters that support "local"
      transactions. -->
<mbean code="org.jboss.resource.ConnectionManagerFactoryLoader"
name="JCA:service=ConnectionManagerFactoryLoader,name=MinervaSharedLocalCMFactory">
  <attribute name="FactoryName">MinervaSharedLocalCMFactory</attribute>
  <attribute name="FactoryClass">
    org.jboss.pool.connector.jboss.MinervaSharedLocalCMFactory
  </attribute>
  <attribute name="Properties"></attribute>
</mbean>

```

```

<!-- Minerva XA transaction connection manager factory

    Use this for resource adapters that support "xa"
    transactions. -->
<mbean code="org.jboss.resource.ConnectionManagerFactoryLoader"
    name="JCA:service=ConnectionManagerFactoryLoader,name=MinervaXACMFactory">
    <attribute name="FactoryName">MinervaXACMFactory</attribute>
    <attribute name="FactoryClass">
        org.jboss.pool.connector.jboss.MinervaXACMFactory
    </attribute>
    <attribute name="Properties"></attribute>
</mbean>

<!-- Connection factory for the Minerva JDBC resource adaptor. This
    points at the same database as DefaultDS. -->
<mbean code="org.jboss.resource.ConnectionFactoryLoader"
    name="JCA:service=ConnectionFactoryLoader,name=MinervaDS">
    <attribute name="FactoryName">MinervaDS</attribute>
    <attribute name="RARDeployerName">JCA:service=RARDeployer</attribute>
    <attribute name="ResourceAdapterName">
        Minerva JDBC LocalTransaction ResourceAdapter
    </attribute>
    <attribute name="Properties">
        ConnectionURL=jdbc:HypersonicSQL:hsqldb://localhost:1476
    </attribute>

    <attribute name="ConnectionManagerFactoryName">
        MinervaSharedLocalCMFactory
    </attribute>
<!-- See the documentation for the specific connection manager
    implementation you are using for the properties you can set -->
<attribute name="ConnectionManagerProperties">
    # Pool type - uncomment to force, otherwise it is the default
    #PoolConfiguration=per-factory

    # Connection pooling properties - see
    # org.jboss.pool.PoolParameters
    MinSize=0
    MaxSize=10
    Blocking=true
    GCEnabled=false
    IdleTimeoutEnabled=false
    InvalidateOnError=false
    TrackLastUsed=false
    GCIntervalMillis=120000
    GCMinIdleMillis=1200000
    IdleTimeoutMillis=1800000
    MaxIdleTimeoutPercent=1.0
</attribute>

<!-- Principal mapping configuration -->
<attribute name="PrincipalMappingClass">
    org.jboss.resource.security.ManyToOnePrincipalMapping

```

```

    </attribute>
    <attribute name="PrincipalMappingProperties">
        userName=sa
        password=
    </attribute>
</mbean>

<!-- JMS XA Resource adaptor, use this to get transacted JMS in beans -->
<mbean code="org.jboss.resource.ConnectionFactoryLoader"
    name="JCA:service=ConnectionFactoryLoader,name=JmsXA">
    <attribute name="FactoryName">JmsXA</attribute>
    <attribute name="RARDeployerName">JCA:service=RARDeployer</attribute>
    <attribute name="ResourceAdapterName">JMS Adapter</attribute>
    <attribute name="ConnectionFactoryName">MinervaXACMFactory</attribute>
    <!-- See the documentation for the specific connection manager
        implementation you are using for the properties you can set -->
    <attribute name="ConnectionFactoryProperties">
        # Pool type - uncomment to force, otherwise it is the default
        #PoolConfiguration=per-factory

        # Connection pooling properties - see
        # org.jboss.pool.PoolParameters
        MinSize=0
        MaxSize=10
        Blocking=true
        GCEnabled=false
        IdleTimeoutEnabled=false
        InvalidateOnError=false
        TrackLastUsed=false
        GCIntervalMillis=120000
        GCMinIdleMillis=1200000
        IdleTimeoutMillis=1800000
        MaxIdleTimeoutPercent=1.0
    </attribute>

    <!-- Principal mapping configuration -->
    <attribute
name="PrincipalMappingClass">org.jboss.resource.security.ManyToOnePrincipalMapping</attribu
te>
    <attribute name="PrincipalMappingProperties">
    </attribute>
</mbean>

<!-- ===== -->
<!-- Auto deployment -->
<!-- ===== -->
<mbean code="org.jboss.ejb.AutoDeployer" name="EJB:service=AutoDeployer">
    <attribute name="Deployers">
        J2EE:service=J2eeDeployer;
        JCA:service=RARDeployer
    </attribute>
    <attribute name="URLs">../deploy,../deploy/lib</attribute>
</mbean>

```



```

<!-- ===== -->
<!-- JMX adaptors -->
<!-- ===== -->

<mbean code="org.jboss.jmx.server.JMXAdaptorService" name="Adaptor:name=RMI" />

<mbean code="org.jboss.jmx.server.RMIConnectorService" name="Connector:name=RMI" />

<mbean code="com.sun.jdmk.comm.HtmlAdaptorServer" name="Adaptor:name=html">
  <attribute name="MaxActiveClientCount">10</attribute>
  <attribute name="Parser" />
  <attribute name="Port">8082</attribute>
</mbean>

<!-- ===== -->
<!-- Mail Connection Factory -->
<!-- ===== -->
<mbean code="org.jboss.mail.MailService" name=":service=Mail">
  <attribute name="JNDIName">Mail</attribute>
  <attribute name="ConfigurationFile">mail.properties</attribute>
  <attribute name="User">user_id</attribute>
  <attribute name="Password">password</attribute>
</mbean>

<!-- ===== -->
<!-- Scheduler Service -->
<!-- ===== -->
<!-- Uncomment this to enable Scheduling - ->
<mbean code="org.jboss.util.Scheduler" name=":service=Scheduler">
  <constructor>
    <arg type="java.lang.String" value=":server=Scheduler"/>
    <arg type="java.lang.String" value="org.jboss.util.Scheduler$SchedulableExample"/>
    <arg type="java.lang.String" value="Schedulabe Test,12345"/>
    <arg type="java.lang.String" value="java.lang.String,int"/>
    <arg type="long" value="0"/>
    <arg type="long" value="10000"/>
    <arg type="long" value="-1"/>
  </constructor>
</mbean>
<!-- - -->

<!-- ===== -->
<!-- Add your custom MBeans here -->
<!-- ===== -->

</server>

```

As you can see from the listing, there are a large number of configured MBean services that ship in the standard JBoss distribution. A summary of each the core MBeans and its attributes are presented in the following sections. The remaining MBeans will be discussed in the corresponding JBossXX component chapter.

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.ejb.ContainerFactory

The ContainerFactory MBean is used to deploy EJB applications. It can be given a URL to an EJB-jar or an EJB-JAR XML file, which will be used to instantiate containers and make them available for use by remote clients.

The configurable attributes of the ContainerFactory include the following:

- **VerifyDeployments:** This flag enables the verification of EJB jar components in accordance with the EJB specifications. Because the EJB specification leaves a number of implementation details as patterns that the bean developer must perform, it is easy to construct an incorrect deployment. The **VerifyDeployments** flag tells the ContainerFactory to validate that the beans in the EJB deployment do correctly conform to the expected patterns. The deployment will fail if any bean fails the verification.
- **VerifierVerbose:** This flag turns on additional output from the verification step. Enabling this flag is useful if an EJB deployment fails verification, and you can't tell what the problem is from the non-verbose output.

- **ValidateDTDs:** This flag enables the validation of EJB deployment descriptors against their DTDs. The container factory parses the standard `ejb-jar.xml` and JBoss specific `jboss.xml` deployment descriptors located in the META-INF directory of the EJB jar deployment. When `ValidateDTDs` is true, the XML parser will validate the deployment descriptor document against the document's DTD.
- **MetricsEnabled:** This is an experimental flag that enables rudimentary metrics for EJB invocations.
- **BeanCacheJMSMonitoringEnabled:** This is an experimental flag that enables the viewing of collected metrics as JMS messages.

`org.jboss.deployment.J2eeDeployer`

The `J2eeDeployer` MBean service allows the deployment of single EJB jars, Web application archives (WARs), and enterprise application archives (EARs). For wars to be deployed, there must be a servlet container, such as Tomcat, embedded in JBoss.

The configurable attributes of the `J2eeDeployer` service are as follows:

- **DeployerName:** A name that is appended to the base "`J2eeDeployer`" service name to create a unique name for the JMX service attribute used in the `javax.management.ObjectName`.
- **JarDeployerName:** The JMX `ObjectName` string for the EJB jar deployer service. The default value is `":service=ContainerFactory"`.
- **WarDeployerName:** The JMX `ObjectName` string for the WAR jar deployer service. The default value is `":service= EmbeddedTomcat"`. You would change the default value if using a servlet container other than Tomcat.

`org.jboss.ejb.AutoDeployer`

The `AutoDeployer` MBean service watches one or more URLs for deployable J2EE 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. If a URL represents a file then the file is deployed and watched for subsequent updates or removal. If a URL represents a directory a check is made to see if the directory contains a META-INF/ejb-jar.xml file, which indicates an unpacked EJB jar deployment. If this file exists, the directory is the deployment unit and the META-INF/ejb-jar.xml is watched for changes to indicate an update has

occurred.

If the directory does not contain a META-INF/ejb-jar.xml file, then all files in the directory are checked to see if they represent valid deployable archives depending on their file extension. Those files that are deployable archives are deployed and watched for updates and removal.

The default value for the URLs attribute is “../deploy,../deploy/lib” which means that any EARs, JARs, WARs or RARs dropped into either directly will be automatically deployed and watched for updates. A common convention is to place EARs, JARs, WARs into the deploy directory while RARs are placed into the deploy/lib directory.

- **Deployers:** The JMX ObjectName strings of the archive deployers the AutoDeployer serves. The default is to use the J2EE component archive deployer service “J2EE:service=J2eeDeployer”.
- **Timeout:** The time in milliseconds between runs of the AutoDeployer thread. The default is 3000 (3 seconds).

org.jboss.jmx.server.RMIConnectorService

The RMIConnectorService MBean service binds an implementation of the org.jboss.jmx.interfaces.RMIConnector into JNDI under the name “jmx:<hostname>:rmi:” where <hostname> is the server hostname. The RMIConnector interface is an RMI interface that exposes a number of the javax.management.MBeanServer interface methods to allow remote clients access to the MBeanServer associated with the JBoss JMX bus. There are no configurable attributes for the RMIConnectorService.

com.sun.jdmk.comm.HtmlAdaptorServer

The HtmlAdaptorServer MBean acts as an HTML server that allows an HTML browser to manage all MBeans in the agent. The HTML protocol adaptor provides the following main HTML pages for managing MBeans in an agent:

- **Agent View:** Provides a list of object names of all the MBeans registered in the agent
- **Agent Administration:** Registers and unregisters MBeans in the agent
- **MBean View:** Reads and writes MBean attributes and performs operations on MBeans in the agent

The configurable attributes of the HtmlAdaptorServer service include the following:

- **MaxActiveClientCount:** The maximum number of concurrent requests the server will accept
- **Port:** The HTTP port on which the adaptor listens for clients

org.jboss.mail.MailService

The MailService MBean provides JavaMail support. The MailService binds a javax.mail.Session in JNDI under the java:/ to allow mail to be sent through the Session object.

The configurable attributes of the MailService service include the following:

- **User:** The user id used to connect to a mail server
- **Password:** The password used to connect to a mail server
- **ConfigurationFile:** The file name of the configuration mail file used by JavaMail to send mail. This file normally resides in the configuration directory of JBoss, and contains name-value pairs (such as "mail.transport.protocol = smtp") as specified in the JavaMail API documentation.
- **JNDIName:** The JNDI subcontext name under the java:/ context into which javax.mail.Session objects are bound. The default is "Mail".

org.jboss.util.Scheduler

The Scheduler MBean service defines the manageable interface for a scheduler that allows you to request an org.jboss.util.Schedulable implementation be run periodically by the service.

The configurable attributes of the Scheduler service include the following:

- **SchedulableClass:** The fully qualified class name of the org.jboss.util.Schedulable implementation to use with the Scheduler.
- **SchedulableArguments:** The comma separated list of arguments for the Schedulable class. This list must have as many elements as the Schedulable Argument Type list; otherwise, the start of the Scheduler will fail. Currently only basic data types, string, and classes with a constructor with a string as only argument are supported.
- **SchedulableArgumentTypes:** The comma separated list of argument types for the Schedulable class. This will be used to find the right constructor, and to create the

right instances with which to call the constructor. This list must have as many elements as the `Schedulable Arguments` list; otherwise, the start of the Scheduler will fail. Currently only basic data types, string, and classes with a constructor with a string as only argument are supported.

- **SchedulePeriod:** The time between two scheduled calls (after the initial call) in milliseconds. This value must be greater than 0.
- **InitialRepetitions:** The number of periods for which the instance will be scheduled. Set to -1 for unlimited repetitions

JBoss and JMX Summary

In this section you were shown how JMX is used to bootstrap the JBoss server components. You were also introduced to the JBoss MBean services notion that extends the basic JMX MBean concept with a set of life cycle operations. An example of how to use MBean services to integrate your custom services was also presented. The JBoss MBeans that are part of the standard distribution were introduced. A summary of each MBean was given along with a description of the MBean's configurable attributes.

The EJB container architecture

The JBoss 2.4 EJB container architecture is a third 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.

To understand how the EJB container works, you will focus on the components encountered along the path an EJB method invocation travels. You will be introduced to the architecture from the perspective of understanding how an EJB call is passed to JBoss through the network layer, and then finally dispatched to the EJB container.

EJBObject and EJBHome

As is discussed in many EJB references, an `javax.ejb.EJBObject` is an object that represents a client's view of the Enterprise Java Bean. It is the responsibility of the container provider to generate the `javax.ejb.EJBHome` and `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.

Virtual EJBObject - the big picture

EJBObject is more of an abstract idea than a physical implementation. Clients are given a remote handle to EJBObjects, but how is the EJBObject physically implemented on the server side? The answer is that it is not implemented at all!

In JBoss there is only one physical object that serves all logical EJBObjects referenced by clients. That physical object is the container. For each type of EJB there is one container object that plays the role of EJBObject by wrapping all instances of a particular EJB type.

JBoss' approach is superior to creating an explicit server side EJBObject in many aspects, and simplifies the container architecture significantly. Clients, however, never notice this. They have an object (dynamic proxy) that looks and feels like a real server EJBObject, but this is merely an illusion. Behind the scenes there is only one instance of a container handling all method invocations for a given class of EJB. The result is full EJBObject conformity.

Dynamic proxies

A dynamic proxy is an object that implements a list of interfaces specified at runtime when the proxy object is created. Each proxy instance has an associated invocation handler object, which implements the java.lang.reflect.InvocationHandler interface.

The EJBObject and EJHome object are created as dynamic proxies using the java.lang.reflect.Proxy.newProxyInstance method, whose signature is as follows:

```
public static Object newProxyInstance(java.lang.ClassLoader loader,
                                     java.lang.Class[] interfaces,
                                     java.lang.reflect.InvocationHandler h)
    throws IllegalArgumentException
```

The parameters are as follows:

- **loader** - the class loader in which to define the proxy class. JBoss passes in the ClassLoader of the home or remote interface class depending on whether an EJBHome or EJBObject is being proxied.
- **interfaces** - the list of interfaces for the proxy class to implement. For the EJBHome proxy the interface list includes the home interface of the bean as well as the javax.ejb.Handle. For the EJBObject proxy the interface list include the remote interface of the bean.
- **h** - the invocation handler to dispatch method invocations to. For the EJBHome proxy, the handler is a org.jboss.ejb.plugins.jrmp.interfaces.HomeProxy object. For the EJBObject proxy, one of the EntityProxy, StatefulSessionProxy or

StatelessSessionProxy objects from the org.jboss.ejb.plugins.jrmp.interfaces package is used depending on the type of the EJB.

Because the java.lang.reflect.Proxy class is serializable, it can be sent to the remote client across the network. The EJBHome proxy is bound into JNDI under the name chosen for the EJBHome by the deployment descriptor. When a client looks up the EJBHome from JNDI, they are unserializing the proxy instance. Because the proxy implements the bean's home interface, it can be cast and used as the home interface.

EJB proxy types

Depending on the type of the EJB on the server, there are four proxy classes: EntityProxy, HomeProxy, StatelessSessionProxy and StatefulSessionProxy. Recall that these classes are not the proxy objects that implement bean home and remote interfaces. These are implementations of the java.lang.reflect.InvocationHandler interface. They are referred to as proxies as well because they serve as proxies for the server EJB container that manages the bean instance with which the client interfaces are associated. All four of the proxy class' subclass the org.jboss.ejb.plugins.jrmp.interfaces.GenericProxy class. The GenericProxy class contains an RMI reference to a org.jboss.ejb.plugins.rmp.interfaces.ContainerRemote interface implementation from the JBoss server. The implementer of ContainerRemote in the JBoss server is the JRMPContainerInvoker.

From client to server

When a client invokes a home or remote method interface on the proxy object they obtained from the JBoss server, the proxy forwards the call to the InvocationHandler, which, as you have seen, is an object of type EntityProxy, HomeProxy, StatelessSessionProxy or StatefulSessionProxy. An attempt is first made to handle the method on the client side. This is possible for methods like toString, equals, hashCode, getEJBMetaData, and so on. When the method invocation is for a method that is implemented by the server side EJB, the call is forwarded to the server using the ContainerRemote RMI reference.

Summarize the components involved in delivery of a client request to the server by tracing a remote call of some business method B of an entity bean. Figure 2-4 provides a high level view of the components involved. First, the method call goes to the proxy that implements the entity bean remote interface. It is then dispatched by the proxy to its invocation handler, which in this case is an EntityProxy. The EntityProxy converts the call into a RemoteMethodInvocation object and then places it into a MarshaledObject. Using the RMI ContainerRemote interface stub of the JRMPContainerInvoker, the remote call is sent to the server's JRMPContainerInvoker object where it is unpacked from MarshaledObject, and handed off to the container.

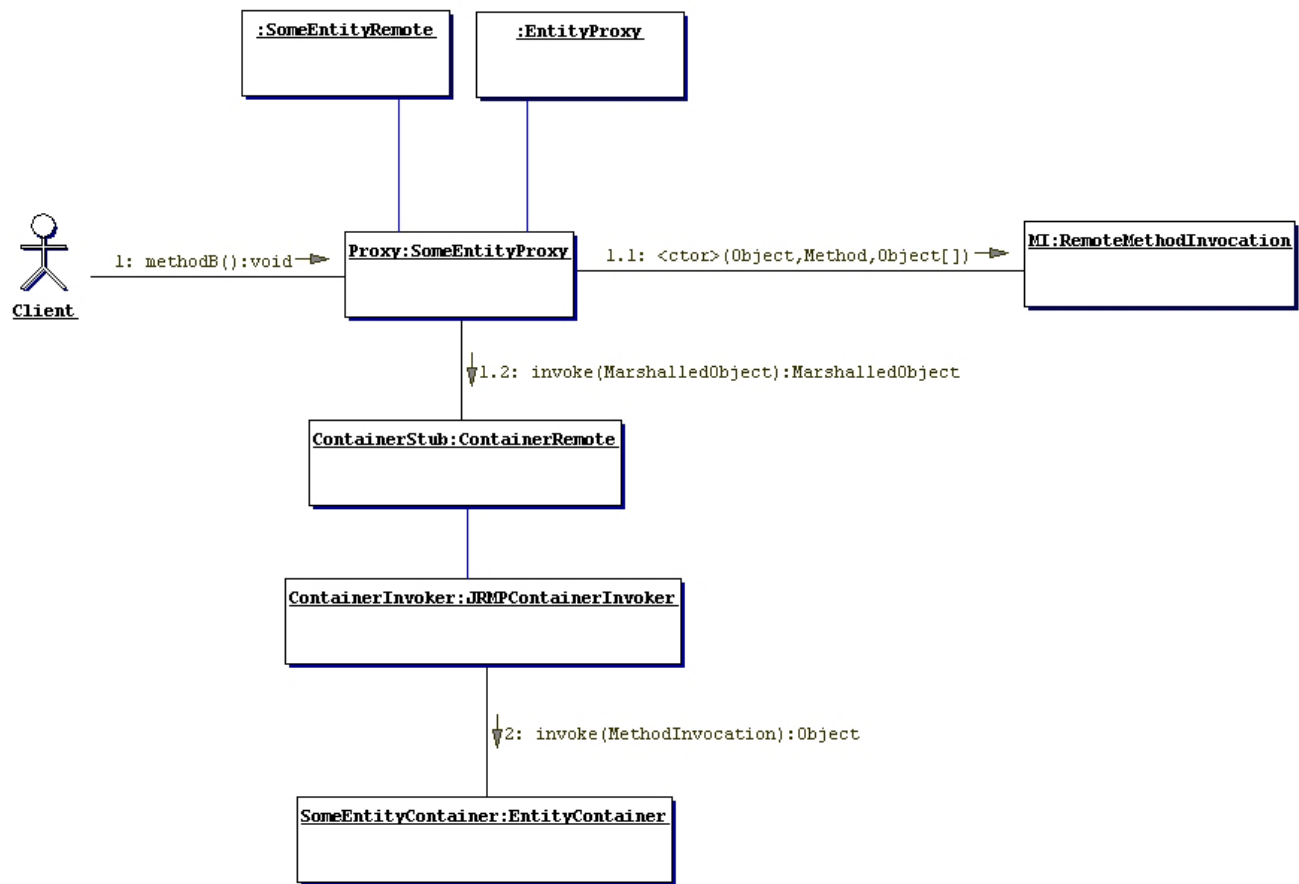


Figure 2-4, The JBoss components involved in delivering an EJB method invocation to the EJB container.

Advantages

This design of client objects gives maximum flexibility in the following sense: All calls that can be handled by the clients are handled locally, preventing the roundtrip across the wire and saving the container from unnecessary calls. Calls coming from clients inside of the server JVM can be optimized by calling the server container invoker directly, and thus avoiding an RMI call and its associated overhead. Finally, only calls that absolutely must leave the local VM are passed across the wire using RMI.

ContainerInvoker – the container transport handler

Certainly one of the most important parts of a distributed system is its remote procedure call (RPC) interface, as well as techniques used in passing that RPC call between different parts of the system. The component that plays the role of the container entry point in JBoss is the **ContainerInvoker** interface. The role of the **ContainerInvoker** is to isolate the transport protocol used to deliver method invocations, and return the reply from the actual dispatch of the method to the EJB implementation. Thus, a given **ContainerInvoker** implementation is

associated with a specific RPC transport. The separation of the client access protocol from the EJB container implementation allows for inclusion of multiple access protocols without changing the EJB container. JBoss can easily add support for RMI/IIOP, SOAP, and any other required transport scheme by simply associating a new ContainerInvoker implementation with a container configuration. Look at the RMI/JRMP version of the ContainerInvoker to get a better understanding of its duties.

The JRMPContainerInvoker

The RMI implementation of the ContainerInvoker provides an RMI version of the org.jboss.ejb.Container invoke and invokeHome methods in the ContainerRemote interface. Listing 2-12 provides the ContainerRemote interface method signatures.

Listing 2-12, The org.jboss.ejb.plugins.jrmp.interfaces.ContainerRemote interface implemented by the JRMPContainerInvoker class.

```
public interface ContainerRemote extends java.rmi.Remote
{
    /**
     * Invoke the remote home instance.
     *
     * @param mi The marshalled object representing the method to
     *          invoke.
     * @return Return value of method invocation.
     *
     * @throws Exception On failure to invoke method.
     */
    MarshalledObject invokeHome(MarshalledObject mi)
        throws Exception;

    /**
     * Invoke a remote object instance.
     *
     * @param mi The marshalled object representing the method to
     *          invoke.
     * @return Return value of method invocation.
     *
     * @throws Exception Failed to invoke method.
     */
    MarshalledObject invoke(MarshalledObject mi)
        throws Exception;

    /**
     * Invoke the local home instance.
     *
     * @param m The method to invoke.
     * @param args The arguments to the method.
     * @param tx The transaction to use for the invocation.
     * @param identity The principal to use for the invocation.
     * @param credential The credentials to use for the invocation.
     */
}
```

```

    * @return          Return value of method invocation.
    *
    * @throws Exception Failed to invoke method.
    */
    Object invokeHome(Method m, Object[] args, Transaction tx,
                      Principal identity, Object credential)
        throws Exception;

    /**
     * Invoke a local object instance.
     *
     * @param id          The identity of the object to invoke.
     * @param m           The method to invoke.
     * @param args        The arguments to the method.
     * @param tx          The transaction to use for the invocation.
     * @param identity    The principal to use for the invocation.
     * @param credential  The credentials to use for the invocation.
     * @return            Return value of method invocation.
     *
     * @throws Exception Failed to invoke method.
     */
    Object invoke(Object id, Method m, Object[] args, Transaction tx,
                  Principal identity, Object credential)
        throws Exception;
}

```

When a JBoss EJB container that is configured to use RMI/JRMP as its transport protocol is initialized, it starts its JRMPContainerInvoker object. This results in the EJB home proxy being bound into JNDI. The home proxy contains a reference to the ContainerRemote interface of the container's JRMPContainerInvoker. The JRMPContainerInvoker also exports itself to the RMI subsystem so that it can begin to accept RMI requests.

ContainerRemote Interface—Two Forms of Invoke Methods

The RMI interface of the JRMPContainerInvoker is the ContainerRemote interface. It has two methods – invoke and invokeHome – each of which has two flavors::

```

public MarshalledObject invoke(MarshalledObject mi) throws Exception;
public MarshalledObject invokeHome(MarshalledObject mi) throws Exception;

```

and

```

public Object invoke(Object id, Method m, Object[] args,
                    Transaction tx, Principal identity, Object credential ) throws Exception;
public Object invokeHome(Method m, Object[] args, Transaction tx,
                        Principal identity, Object credential) throws Exception;

```

The first flavor accepts only one parameter – an RMI MarshalledObject, while the second flavor accepts a method reflection style of parameters. The different signatures exist to allow the JRMPContainerInvoker to accept both remote and local client calls. This allows a caller

in the same VM as the EJB container the choice of a standard call by reference method invocation as an optimization over the RMI call-by-value semantics. Note that because the EJB specification requires that the semantics of an EJB method invocation follow the RMI call-by-value semantics, if an in-VM caller wants to preserve these semantics it would use the form that accepted and returned the MarshaledObject. The optimization choice is a non-EJB 1.1 specification option that offered performance over specification conformance. However, the EJB 2.0 local interfaces notion adds the option of call-by-reference semantics. Thus, a caller using the EJB 2.0 local-interfaces semantics would make use of the optimized method invocation.

Handling the method invocations

Remote calls are delivered to the JRMPContainerInvoker object in the RMI subsystem. The contents of the method invocation are unpacked from MarshaledObject and then passed to the invokeHome or invoke methods. A MarshaledObject contains a byte stream with the serialized representation of an object. As was discussed earlier in this chapter, the serialized object is RemoteMethodInvocation generated by the client side GenericProxy subclass. The RemoteMethodInvocation instance contains all the attributes of the original EJB method invocation, along with any security and transaction contexts. The RemoteMethodInvocation is deserialized from the MarshaledObject, converted to an org.jboss.ejb.MethodInvocation, and handed off to the container.

Local calls coming from clients in the same VM, such as an EJB to EJB method call, are directly handed off to the container unless the container is configured to adhere to RMI call-by-value semantics. This bypasses the network layer, as well as any object marshalling of the call that RMI calls have to go through.

Other ContainerInvoker duties

Before forwarding a call to the container, the ContainerInvoker is responsible for establishing the thread context ClassLoader to that of the container, as well as propagating any transaction and security context information.

Another important role played by the ContainerInvoker is that it provides implementation of the EJBObject and EJBHome parts of the container that are appropriate for the ContainerInvoker protocol transport. As mentioned earlier in this chapter, the JRMPContainerInvoker creates EJBObject and EJBHome in the form of dynamic proxies. The EJBHome is created during the initialization phase of the container. EJBObjects are created as they are needed. For example, an EntityBean finder may result in a set of primary keys whose EJBObjects must be returned to the client. The ContainerInvoker is then responsible for creating EJBObject instances that can be used by the client in accord with the ContainerInvoker RPC protocol.

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 class of 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 a ContainerFactory MBean.

ContainerFactory MBean

The org.jboss.ejb.ContainerFactory MBean is responsible for the creation of EJB containers. Given an EJB-jar that is ready for deployment, the ContainerFactory will create and initialize the necessary EJB containers, one for each type of EJB. The key methods of the org.jboss.ejb.ContainerFactoryMBean interface are given in Listing 2-13.

Listing 2-13, The org.jboss.ejb.ContainerFactoryMBean interface

```
public interface ContainerFactoryMBean
    extends org.jboss.util.ServiceMBean
{
    /** Returns the applications deployed by the container factory
     */
    public java.util.Iterator getDeployedApplications();
    /**
     * Deploy an application
     *
     * @param url URL to the directory with the given EJBs
     *         to be deployed
     * @param appId Id of the application this EJBs belongs
     *         to use for management
     * @exception MalformedURLException
     * @exception DeploymentException
     */
    public void deploy(String url, String appId )
        throws MalformedURLException, DeploymentException;

    /** Deploy an application
     *
     * @param appUrl Url to the application itself
     * @param jarUrls Array of URLs to the JAR files containing
     *         the EJBs
     * @param appId Id of the application this EJBs belongs to
     *         used for management
     * @exception MalformedURLException
     * @exception DeploymentException
     */
    public void deploy( String appUrl, String[] jarUrls, String appId )
        throws MalformedURLException, DeploymentException;

    /** Undeploy an application
     *
     * @param url
     */
}
```

```

    * @exception    MalformedURLException
    * @exception    DeploymentException
    */
    public void undeploy(String url)
        throws MalformedURLException, DeploymentException;

    /** Enable/disable bean verification upon deployment.
     *
     * @param  verify  true to enable the verifier; false to disable
     */
    public void setVerifyDeployments(boolean verify);
    /** Returns the state of the verifier (enabled/disabled)
     *
     * @return  true if verifier is enabled; false otherwise
     */
    public boolean getVerifyDeployments();

    /** Enable/disable bean verifier verbose mode.
     *
     * @param    verbose true to enable verbose mode; false to disable
     */
    public void setVerifierVerbose(boolean verbose);
    /** Returns the state of the verifier (verbose/non-verbose mode).
     *
     * @return  true if the verbose mode is enabled; false otherwise
     */
    public boolean getVerifierVerbose();

    /** Enables/disables the metrics interceptor for containers.
     *
     * @param enable  true to enable; false to disable
     */
    public void setMetricsEnabled(boolean enable);
    /** Checks if this container factory initializes the
     *    metrics interceptor.
     *
     * @return  true if metrics are enabled; false otherwise
     */
    public boolean isMetricsEnabled();

    /** Is the application with this url deployed
     *
     * @param  url
     * @exception    MalformedURLException
     */
    public boolean isDeployed(String url)
        throws MalformedURLException;

    /** Get the flag indicating that ejb-jar.dtd, jboss.dtd &
     *    jboss-web.dtd conforming documents should be validated
     *    against the DTD.
     */
    public boolean getValidateDTDs();
    /** Set the flag indicating that ejb-jar.dtd, jboss.dtd &

```

```
    jboss-web.dtd conforming documents should be validated
    against the DTD.
    */
    public void setValidateDTDs(boolean validate);
}
```

The factory 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.

Container configuration information

JBoss externalizes most if not all of the setup of the EJB containers through an XML file that conforms to the jboss_2_4 DTD. The section of the jboss_2_4 DTD that relates to container configuration information is shown in Figure 2-5.

2 JBOSS SERVER ARCHITECTURE OVERVIEW

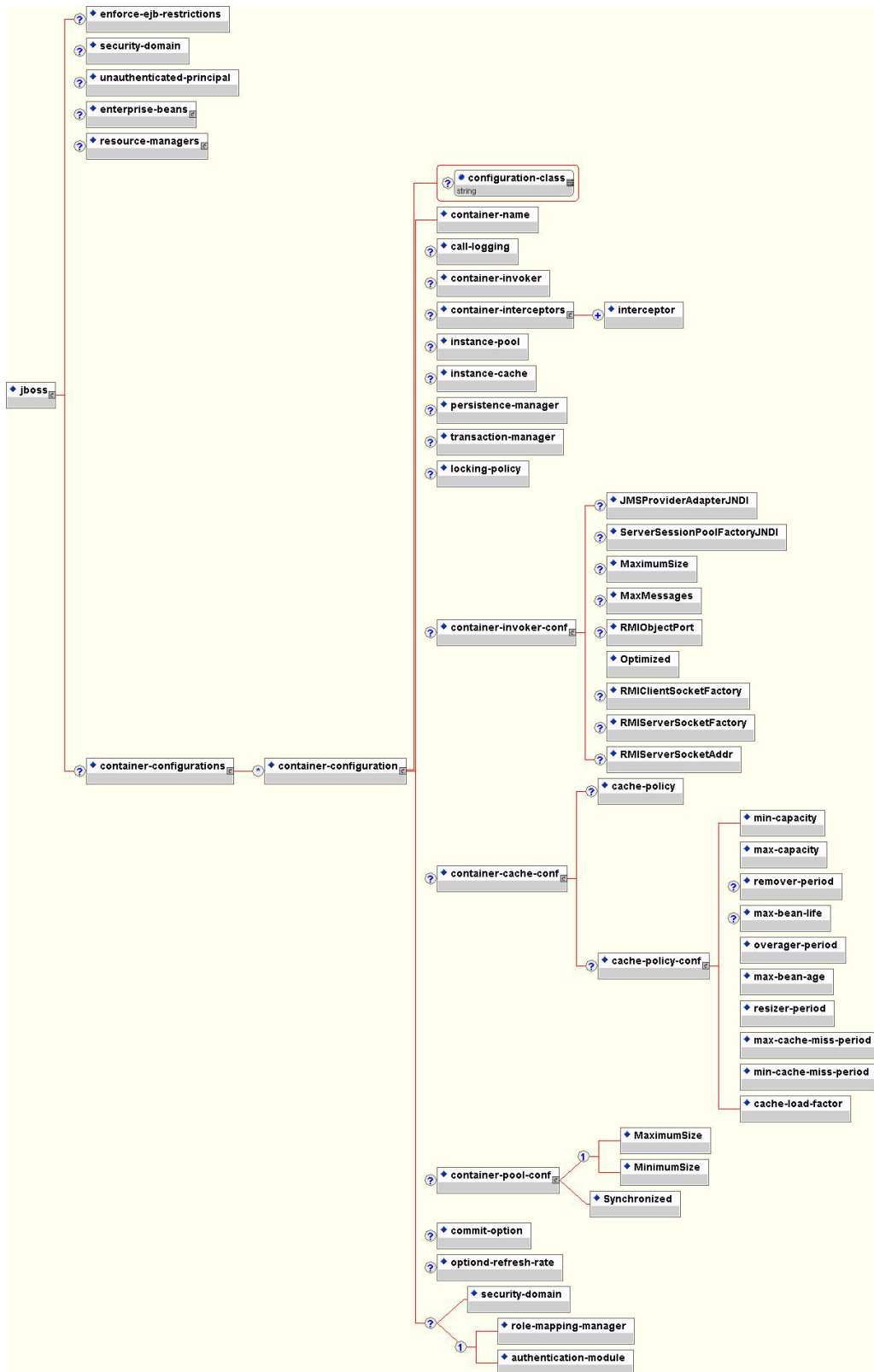


Figure 2-5, The jboss_2_4 DTD elements related to container configuration.

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 configuration file set directories of any JBoss distribution. Listing 2-14 gives a sample of the content of `standardjboss.xml` configuration.

Listing 2-14, The most common container-configuration elements in the default distribution `standardjboss.xml` file.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 2.4//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_2_4.dtd">
<jboss>
  <enforce-ejb-restrictions>false</enforce-ejb-restrictions>
  <container-configurations>
    <container-configuration>
      <container-name>Standard CMP EntityBean</container-name>
      <call-logging>false</call-logging>
      <container-invoker>org.jboss.ejb.plugins.jrmp.server.JRMPContainerInvoker
      </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.EntityLockInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
      </container-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.jaws.JAWSPersistenceManager
      </persistence-manager>
      <transaction-manager>org.jboss.tm.TxManager</transaction-manager>
      <locking-policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</locking-policy>
      <container-invoker-conf>
        <RMIOBJECTPort>4444</RMIOBJECTPort>
        <Optimized>True</Optimized>
      </container-invoker-conf>
      <container-cache-conf>
        <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-policy>
        <cache-policy-conf>
```

```

    <min-capacity>50</min-capacity>
    <max-capacity>1000</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>
  <MinimumSize>10</MinimumSize>
</container-pool-conf>
  <commit-option>A</commit-option>
</container-configuration>
</container-configurations>
</jboss>

```

Figure 2-5 and Listing 2-14 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, which is discussed in Chapter 9, "Advanced JBoss configuration using jboss.xml". 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.

Verifying EJB deployments

Another role that the ContainerFactory performs is the verification of EJB deployments. An option shown in the description of the ContainerFactoryMBean was a flag to validate EJB deployments. When this option is set to true, any bean in an EJB deployment unit is checked for EJB specification compliance. This entails validating that the EJB deployment unit contains the required home and remote interfaces, and that the objects appearing in these interfaces are of the proper types. 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 synch. 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 ContainerFactory 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 classes and 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.

Inside the EJB org.jboss.ejb.Container class

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, and container invokers. 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 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.EntityPersistenceManager
- org.jboss.ejb.EntityPersistenceStore
- 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 2-15.

Listing 2-15, the org.jboss.ejb.ContainerPlugin interface

```
public interface ContainerPlugin extends org.jboss.util.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 2-16.

Listing 2-16, the org.jboss.ejb.Interceptor interface

```
public interface Interceptor extends ContainerPlugin
{
    public void setNext(Interceptor interceptor);
    public Interceptor getNext();
    public Object invokeHome(MethodInvocation mi) throws Exception;
    public Object invoke(MethodInvocation mi) throws Exception;
}
```

All interceptors defined in the container configuration are created and added to the container interceptor chain by the ContainerFactory. The last interceptor is not added by the container factory 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 MethodInvocation object is passed. The first interceptor in the chain is invoked when ContainerInvoker passes a MethodInvocation to the container. 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 2-17 gives the InstancePool interface.

Listing 2-17, 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 2-18 gives the InstanceCache interface.

Listing 2-18, 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 2-19 gives the EntityPersistenceManager interface.

Listing 2-19, the org.jboss.ejb.EntityPersistenceManager interface


```

public interface EntityPersistenceManager extends ContainerPlugin
{
    /** 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. The associated
    ejbCreate and ejbPostCreate methods must be invoked on the
    instance.

    @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
    @exception Exception thrown on any create failure
    */
    public void 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
    @exception RemoteException thrown if some system exception occurs
    @exception FinderException thrown if some heuristic problem occurs
    */
    public 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
    @exception RemoteException thrown if some system exception occurs
    @exception FinderException thrown if some heuristic problem occurs
    */
    public Collection findEntities(Method finderMethod, Object[] args,
        EntityEnterpriseContext instance) throws Exception;

    /** This method is called when an entity shall be activated.
    This is an hook a store can use to establish instance
    contexts (JAWS does for smart updates). The persistence manager
    must invoke ejbActivate on the instance.

    @param instance, the instance to use for the activation
    @exception RemoteException thrown if some system
        exception occurs

```

```

*/
public void activateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/** This method is called whenever an entity shall be loaded
from the underlying storage. The store must populate the instance
fields and the persistence manager must invoke ejbStore.

@param instance, the instance to synchronize
@exception RemoteException thrown if some system
    exception occurs
*/
public void loadEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/** This method is called whenever an entity shall be
stored to the underlying storage. The store must save the
fields of the instance and the persistence manager must
invoke ejbStore on the instance.

@param instance, the instance to synchronize
@exception RemoteException thrown if some system
    exception occurs
*/
public void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/** This method is called when an entity shall be passivated.
The persistence manager must invoke ejbPassivate on the instance.

See the activate discussion for the reason for exposing
EJB callback calls to the store.

@param instance, the instance to passivate
@exception RemoteException thrown if some system
    exception occurs
*/
public void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/** This method is called when an entity shall be removed
from the underlying storage. The store must remove all
state for the instance and the persistence manager invoke
ejbRemove on the instance.
@param instance the instance to remove
@exception RemoteException thrown if some system
    exception occurs
@exception RemoveException thrown if the instance
    could not be removed
*/
public void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}

```

As per the EJB 1.1 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.EntityPersistenceStore](#) interface. By default this is the [org.jboss.ejb.plugins.jaws.JAWSPersistenceManager](#), (JAWS-Just Another Web Store). JAWS performs basic O/R functionality against a JDBC-store. More details about JAWS can be found in the Chapter 5, “JBossCMP – The JBoss Container Managed Persistence Layer.” Listing 2-20 gives the [EntityPersistenceStore](#) interface.

Listing 2-20, the org.jboss.ejb.EntityPersistenceStore interface

```
public interface EntityPersistenceStore extends ContainerPlugin
{
    /** 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.

    The return is the primary key in case of CMP PM
    Null in case of BMP PM (but no store should exist)

    * @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 Object, the primary key computed by CMP PM or null for BMP
    * @exception Exception
    */
    public 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
    * @exception RemoteException thrown if some system exception occurs
    * @exception FinderException thrown if some heuristic problem occurs
    */
    public 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
    * @exception RemoteException thrown if some system exception occurs
    * @exception FinderException thrown if some heuristic problem occurs
    */
    public FinderResults findEntities(Method finderMethod,
        Object[] args, EntityEnterpriseContext instance)
        throws Exception;

    /** This method is called when an entity shall be activated.

    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
    * @exception RemoteException thrown if some system
    exception occurs
    */
    public 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
    * @exception RemoteException thrown if some system
    exception occurs
    */
    public void loadEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /** This method is called whenever a set of entities should
    be preloaded from the underlying storage. The persistence store
    is allowed to make this a null operation

    * @param instances the EntityEnterpriseContexts for the
    entities that must be loaded
    * @param keys a PagableKeyCollection previously returned
    from findEntities.
    */
    public void loadEntities(FinderResults keys)
        throws RemoteException;

    /** 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
    * @exception RemoteException thrown if some system
        exception occurs
    */
    public 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.

    See the activate discussion for the reason for exposing
    EJB callback calls to the store.

    * @param instance the instance to passivate
    * @exception RemoteException thrown if some system
        exception occurs
    */
    public 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
    * @exception RemoteException thrown if some system
        exception occurs
    * @exception RemoveException thrown if the instance
        could not be removed
    */
    public 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 2-21 gives the StatefulSessionPersistenceManager interface.

Listing 2-21, 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.ejb.plugins.StatefulSessionFilePersistenceManager`. As its name implies, `StatefulSessionFilePersistenceManager` utilizes the file system to persist stateful `SessionBeans`. 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.

Tracing the call through container

The preceding sections discussed specific pieces involved in passing an EJB call invocation to the EJB container. Now it is time to put all the pieces together to see how a complete method invocation is passed through the container to the EJB implementation. In particular, we will look at the handling of method calls on a CMP entity bean.

The entry point into the container is when the `ContainerInvoker` calls `invoke` passing in the MethodInvocation object. The container retrieves the first interceptor in its interceptor chain and calls `invoke` on the interceptor, passing the MethodInvocation object. Recall from the `standardjboss.xml` descriptor that the container invokers configured for a CMP entity bean, and the ordering of the interceptors is as follows:

1. `org.jboss.ejb.plugins.LogInterceptor`

2. `org.jboss.ejb.plugins.SecurityInterceptor`
3. `org.jboss.ejb.plugins.TxInterceptorCMT`
4. `org.jboss.ejb.plugins.EntityLockInterceptor`
5. `org.jboss.ejb.plugins.EntityInstanceInterceptor`
6. `org.jboss.ejb.plugins.EntitySynchronizationInterceptor`
7. The `org.jboss.ejb.EntityContainer.ContainerInterceptor` added by the `EntityContainer` itself.

The start of the call is first optionally logged by the `LogInterceptor` depending on the `log4j` logging priority threshold. The information logged includes the method name and parameters.

The `SecurityInterceptor` then checks to see if the bean has been configured with a security domain. If it has, the caller is first authenticated and then authorized against the configured security domain manager and the `ejb-jar.xml` method permissions for the bean. The identity of the caller and the caller's credentials are obtained from the `MethodInvocation` object. If the caller fails the security tests, a `java.security.SecurityException` is thrown and the call is aborted. If there is no security domain specified the call is simply passed to the next interceptor.

The `TxInterceptorCMT` then decides how to manage transactions for this call. The information needed for this decision comes from the method transaction attributes defined in the `ejb-jar.xml` descriptor. The transaction information is associated with the `MethodInvocation` object if appropriate.

This is followed by the `EntityLockInterceptor`, whose role is to schedule the current thread. Entity beans are single-threaded by default, and so only one thread may be executing inside of the bean implementation class at any given moment. The `EntityLockInterceptor` synchronizes all calling threads.

Until this point in the interceptor chain no bean instance has been associated with the `MethodInvocation`. The `EntityInstanceInterceptor` role is to acquire a context representing the target object from the cache. The interceptor calls the `InstanceCache` with the primary key associated with the `MethodInvocation` to do this. Because the cache does not yet have an instance associated with the given primary key, it first gets a free instance from the instance pool, which it associates with the primary key. It then calls the persistence manager that will activate the instance.

After instance acquisition, the EntitySynchronizationInterceptor manages how this instance is synchronized with the database. 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. This is triggered by transaction demarcation in the presence of a transaction. 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 by this interceptor.

Lastly, the container-provided interceptor is invoked. The container always adds itself as the last interceptor at the end of the interceptor chain so that it may delegate business methods to the EJB instance. The instance performs some work, and returns a result. The interceptor chain is now unwound as each interceptor returns from the invoke-operation.

The EntitySynchronizationInterceptor interceptor chooses to store the current state into the database and hence calls storeEntity on the persistence manager, if appropriate depending on the commit and transaction options. The javax.transaction.Synchronization instance created by this interceptor will handle the completion of the transaction.

The EntityInstanceInterceptor then returns the bean instance to the cache.

TxInterceptorCMT interceptor handles the method return according to the transaction settings, possibly committing or rolling back the current transaction.

The SecurityInterceptor never does anything on the call return path, but the LogInterceptor logs the call completion, depending on the log4j logging priority threshold.

Finally, the container invoker returns the result to the client. As you can see, all implementation decisions are performed by various interceptors and plug-ins associated with the container. These decisions are loosely coupled, which allows the deployer of the EJB-application to tweak the behavior of the container to a great degree. This also allows for a number of independent plug-ins to co-exist, each one allowing for slightly, or radically, different behavior.

An example customization would be a persistence manager that used XML files as the backing store instead of a relational database. Another example would security interceptor that used access control lists from a database instead of the ejb-jar.xml descriptor to perform security checks, or multiple security checks could be performed by configuring the container to have multiple security interceptors of different types. All of these options are available by the componentized nature of the container architecture.

Summary

In this chapter you were introduced to JMX and its use as a component bus by the JBoss server. You were also introduced to the JBoss MBean services notion, which is an extension of the basic JMX MBean concept that adds a set of life cycle operations. An example of how to use MBean services to integrate your custom services was presented. Also presented were the JBoss MBeans that are part of the standard distribution. A summary of each MBean was given along with a description of the MBean's configurable attributes.

You were also introduced to the EJB container architecture. The customizable nature of the container architecture due to its use of plug-in interfaces was emphasized. To demonstrate how the various EJB container components interact, you were walked through the container components, as they would be encountered by an EJB method invocation.

You will next cover the naming service implementation used by JBoss, the JBossNS component. This will include the role of naming services in J2EE as well as the JBossNS architecture.

3. JBossNS - The JBoss Naming Service

JBossNS, the JNDI based naming service component

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, JBossNS.

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*. JBossNS is an example JNDI implementation, 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. JBossNS 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
```

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.

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.

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, 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 be 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

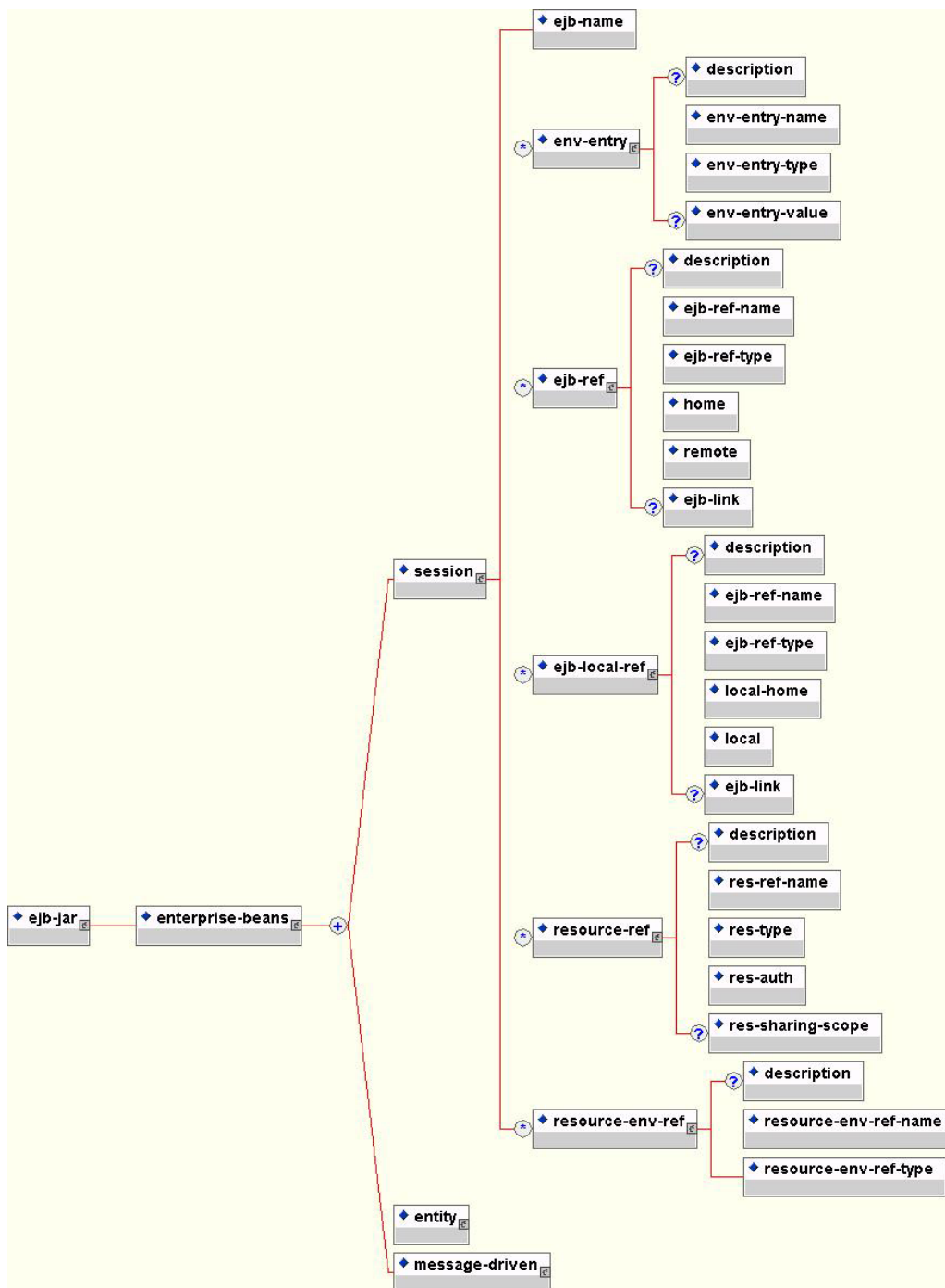
The EJB 2.0 deployment descriptor describes a collection of EJB components and their environment. Each of the three types of EJB components—session, entity, and message-

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 http://java.sun.com/dtd/ejb-jar_2_0.dtd.

Figure 3-1, The ENC elements in the standard EJB 2.0 `ejb-jar.xml` deployment descriptor.



The web.xml ENC Elements

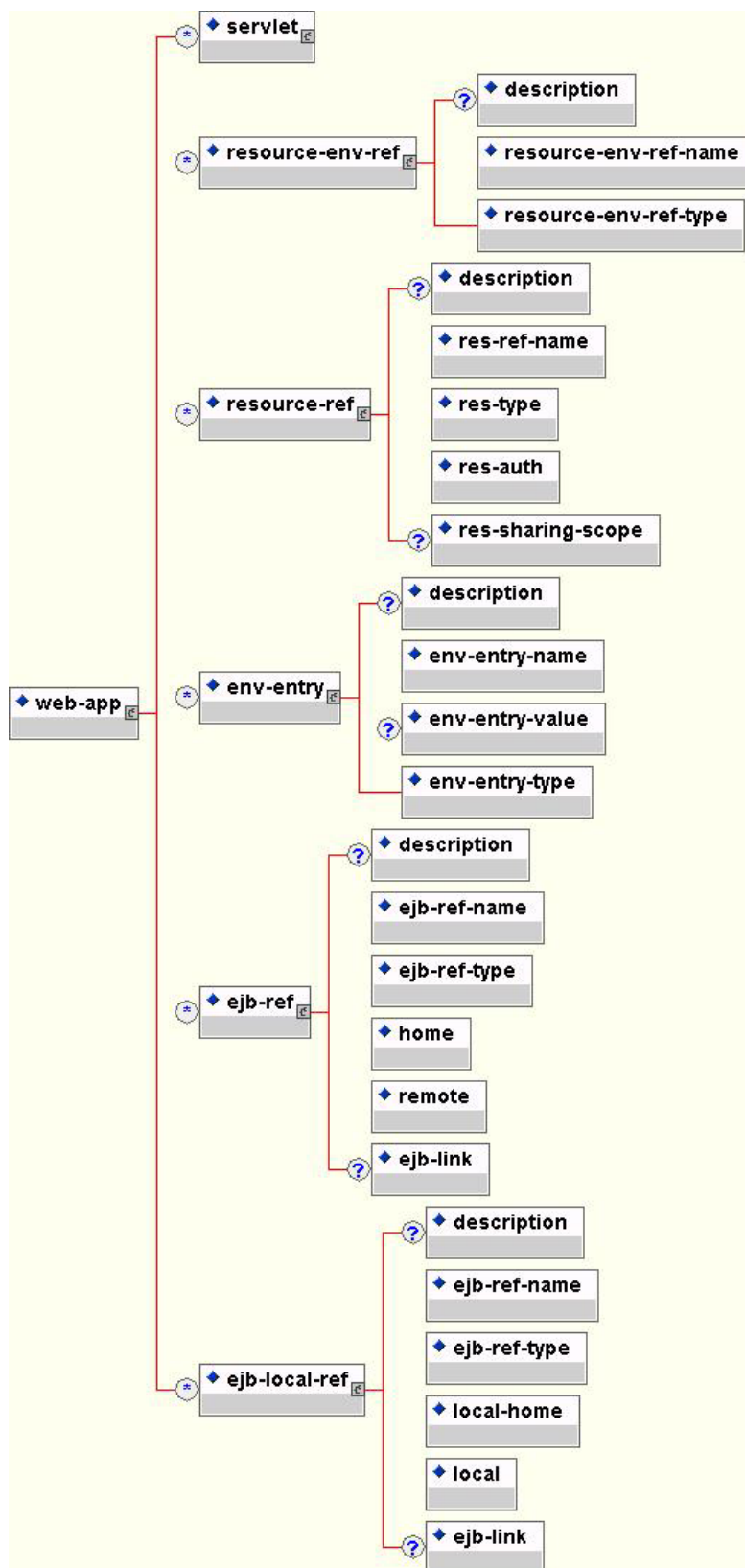
The Servlet 2.3 deployment descriptor describes a collection of Web components and their environment. The ENC for a Web application is declared globally for all servlets and JSP pages in the Web application. Because the Web application developer generally cannot know

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.

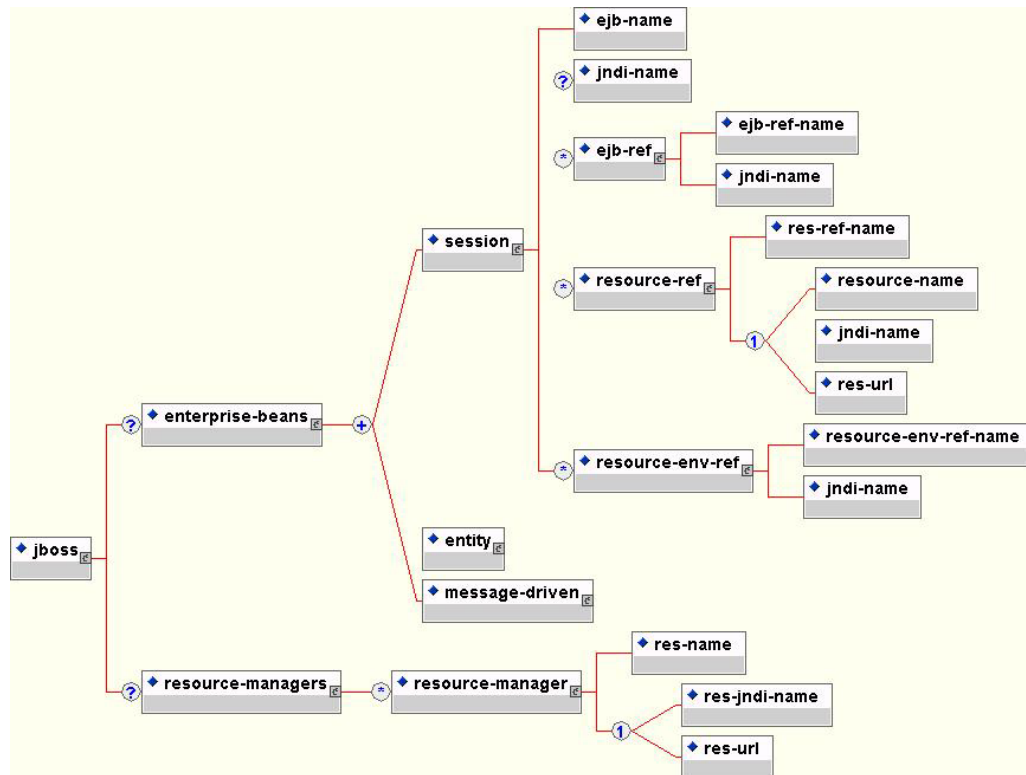
Figure 3-2, The ENC elements in the standard servlet 2.3 web.xml deployment descriptor.



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, The ENC elements in the JBoss 2.4 `jboss.xml` deployment descriptor, 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 2.4 `jboss.xml` deployment descriptor.



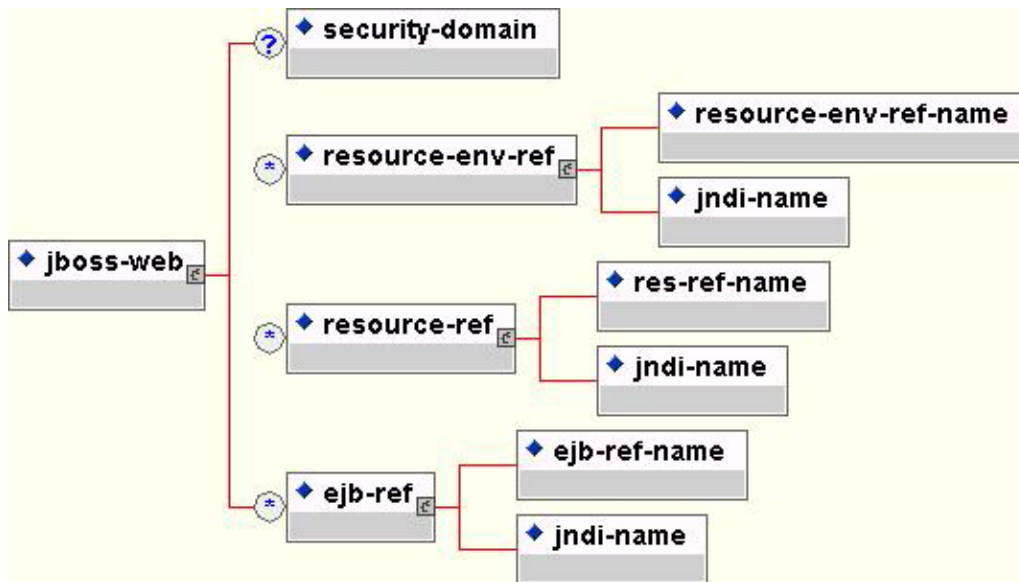
The jboss-web.xml ENC Elements

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 2.4 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, `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, 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>
  <remote> org.jboss.store.ejb.Product</remote>
</ejb-ref>
</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 `jni-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 `jni-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` `jni-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/jni-name` element.

The `jboss-web.xml` descriptor is used only to set the destination to which a Web application ENC `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 `jni-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, 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, 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, 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, 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/QueueFactory</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 `java:comp/env/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.

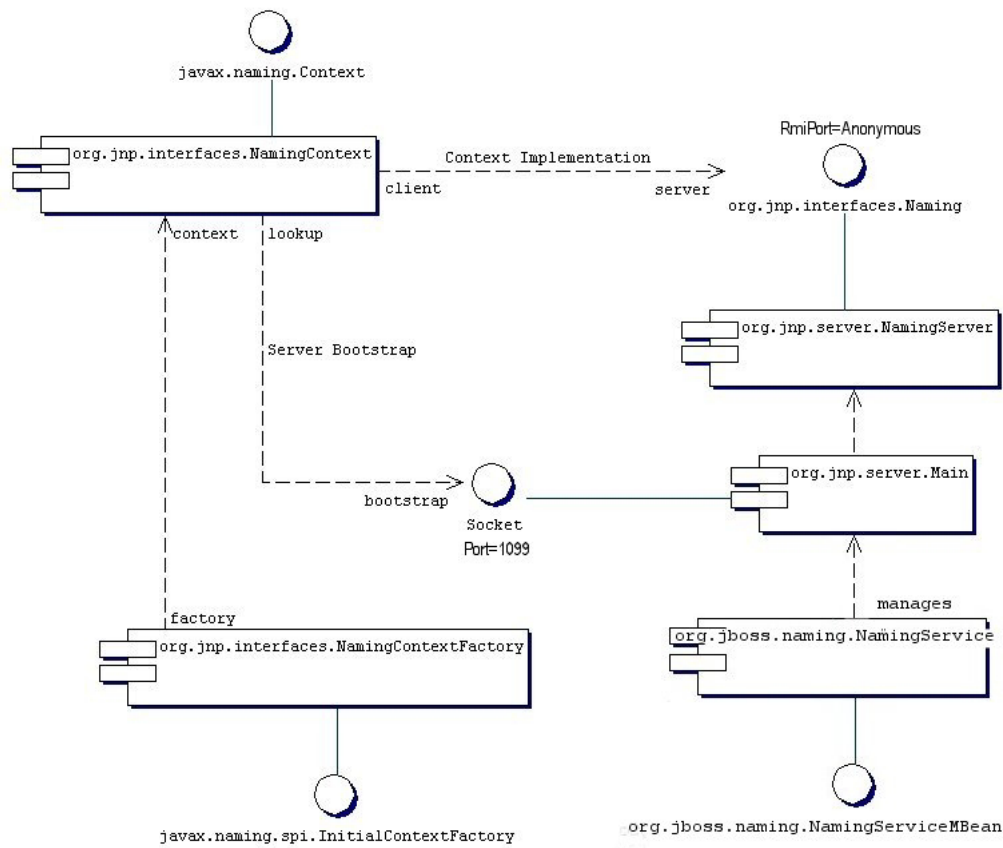
Listing 3-15, sample jboss.xml resource-env-ref descriptor fragment

```
<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.
- **JNPSServerSocketFactory,** 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 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.

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 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 JBossNS InitialContext Factory

The properties required for the InitialContext to work with the JBossNS JNDI provider are as follows:

- **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. This must be the org.jnp.interfaces.NamingContextFactory class for JBossNS.
- **java.naming.provider.url (or Context.PROVIDER_URL)**, The name of the environment property for specifying the location of the JBossNS 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 JBossNS. 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 JBossNS this must be org.jboss.naming:org.jnp.interfaces. This property is essential for locating the jnp: and java: URL context factories bundled with the JBossNS 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 subsequent 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.

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 3-16 shows two configurations—one for an LDAP server, and the other for a local file system directory.

Listing 3-16, ExternalContext MBean configurations

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"
  name=":service=ExternalContext,jndiName=external/ldap/dsape" >
  <attribute name="JndiName">external/ldap/dsape</attribute>
  <attribute name="Properties">dsape.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=":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/dscape. An example dscape.ldap properties file is as follows:

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://ldaphost.displayscape.com:389/o=displayscape.com
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.displayscape.com:389/o=displayscape.com from within the JBoss VM using the following code fragment:

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

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 can be found in the standard jboss.jcml configuration file is as follows:

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

This says that the JNDI name QueueConnectionFactory should be a binding to a LinkRef that points to the binding for the JNDI name ConnectionFactory.

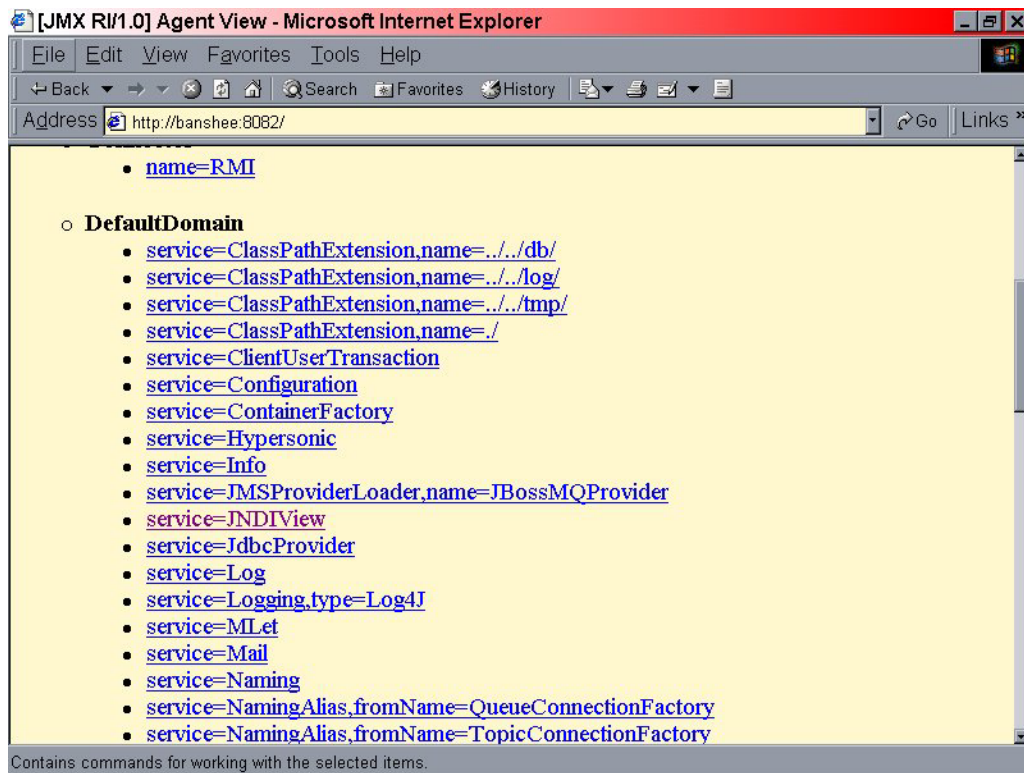
The org.jboss.naming.JNDIView MBean

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="DefaultDomain:service=JNDIView"/>
```

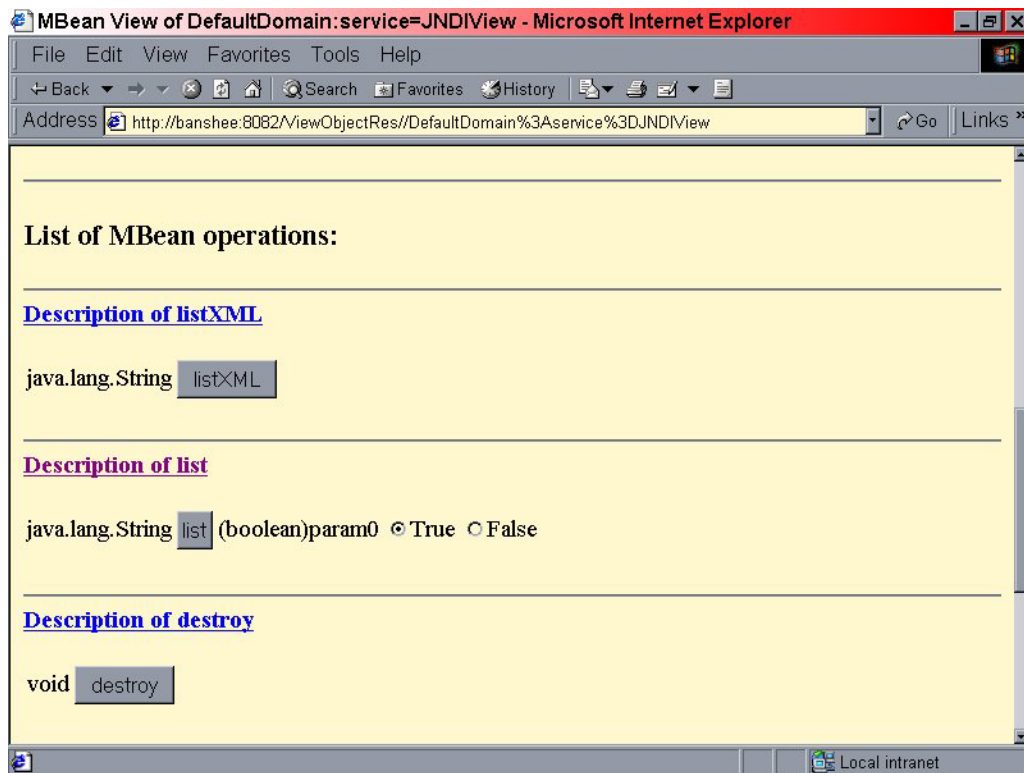
To view the JBoss JNDI namespace using the JNDIView MBean, you connect to the JMX Agent View using the http interface. The default settings put this at <http://localhost:8082/>. On this page you will see a section that lists the registered MBeans by domain. It should look something like that shown in Figure 3-6, The HTTP JMX agent view of the configured JBoss MBeans., where the JNDIView MBean is under the mouse cursor.

Figure 3-6, The HTTP JMX agent view of the configured JBoss MBeans.



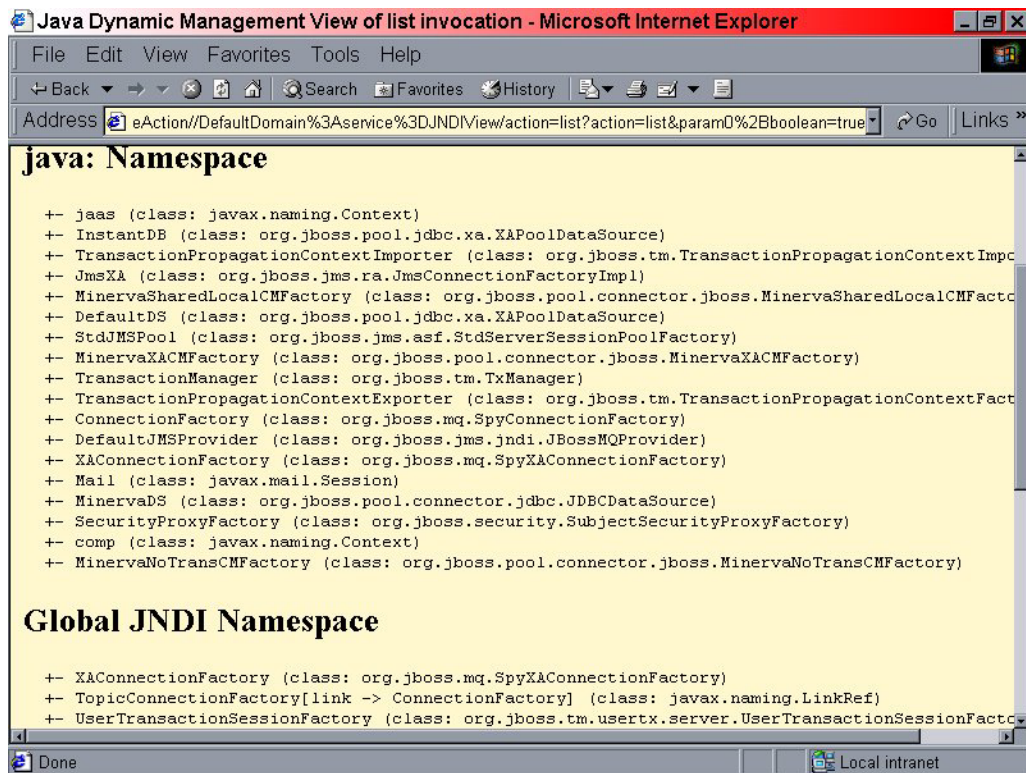
Selecting the JNDIView link takes you to the JNDIView MBean view, which will have a list of the JNDIView MBean operations. This view should look similar to that shown in Figure 3-7, The HTTP JMX MBean view of the JNDIView MBean..

Figure 3-7, 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-2.4.5 distribution server produced the view shown in Figure 3-8, The HTTP JMX view of the JNDIView list operation output..

Figure 3-8, The HTTP JMX view of the JNDIView list operation output.



Summary

This chapter defined the basic JNDI API and common usage conventions as they apply to J2EE. You learned how to configure the J2EE component ENC using the standard and JBoss specific deployment descriptors. You were also introduced to the JBoss JNDI provider service, JBossNS, and saw an overview of its architecture. Lastly, you saw the MBeans of the JBossNS service that controlled its configuration.

The next JBoss component you will cover is the JBossMQ service. JBossMQ is a Java Message Service compatible messaging service implementation.

4. JBossMQ - The JBoss Messaging Service

A JMS 1.0.2 compatible messaging framework

The Java Message Service (JMS) is an asynchronous message delivery abstraction that plays an important role in enterprise integration scenarios since it allows for decoupled communication. This means that a message sender does not have to have a direct connection to the ultimate recipient of the message, as is the case for remote procedure call (RPC) based EJBs. JMS has been around as a specification since August of 1998, but it only recently has been incorporated fully into the J2EE standards as of EJB 2.0 and J2EE 1.3. The JBoss messaging service is an implementation of the JMS 1.0.2 API specification. In this chapter we will provide an overview of JMS and how JMS fits into J2EE, provide some simple examples of JMS and MDB usage, discuss the JBossMQ architecture, and cover the configuration of the JBossMQ service as of the JBoss 2.4.5 release.

An Overview of JMS

JMS is an API that describes an interface to client/server messaging systems. Client/server messaging systems are commonly referred to as message-oriented middleware (MOM) and have been around for many years in the form of proprietary offerings. The purpose of the JMS API is to provide a vendor independent API for using MOM services from Java. To use JMS as the API for a particular MOM provider, the vendor must provide an implementation of the JMS API. JBossMQ is a message-oriented middleware service that integrates into the JBoss server architecture, and provides a JMS compatible implementation.

A message-based architecture is useful when:

- Components do not have tight coupling in the form of references to Java interfaces. Rather than strongly typed interfaces, communication is based on standardized or agreed upon message formats.
- Components are distributed and have unpredictable availability.
- Communication between components can be done in an asynchronous fashion. This means that component 1 can send a message to component 2 and not receive a reply until some future time, if a reply is needed at all.

Such characteristics are common of many enterprise or business-to-business communication patterns.

The JMS Architecture

From a high level view, messaging systems are conceptually simple. Someone sends a message to a destination and someone retrieves the message. The messaging system handles the details of how the destinations are managed and messages sent and received. This simplicity is reflected in the fact that there are just three basic notions that show up in the JMS architecture:

- Administered top-level objects that a JMS client accesses using JNDI. These consist of message destinations and connection factories. A destination is a rendezvous point to which messages are sent and retrieved from. A connection factory is an entry point into the JMS architecture.
- The messages used for communication between destinations. Messages are the communication content and several different types of messages are supported.
- The JMS provider implementation that ties connection factories, destinations and messages together. The majority of the JMS API consists of interfaces for which an implementation must be provided. The JMS provider implementation presents the provider messaging system in terms of the JMS API interface abstractions.

You will be presented each notion in a high-level fashion first to cover the concepts and follow up with the JMS API specifics.

Message Destinations and Connection Factories

To send a message you need a location to address the message to. To receive a message you need a location from which you can pick it up. This rendezvous address is known as a destination in JMS. There are actually two types of destinations in JMS based on two popular messaging models, point-to-point (PTP) and publish/subscribe (pub/sub).

The PTP model is based on the concepts of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from same queue. Queues retain all messages sent to them until the messages are consumed or until the messages expire. For each message sent to a queue, only a single client can receive or consume the message.

The pub/sub model is based on the concepts of message topics, publishers and subscribers. A publisher addresses a message to a specific topic to which one or more subscribers listen. Typically, clients must be listening to the topic during the time the publisher sends a message to the topic in order to receive the message. The JMS architecture allows for a

variation on this theme based on the notion of a durable topic. This is a type of topic that has both queue and traditional topic qualities. Messages sent to a durable topic remain in the topic until subscribers retrieve them, and all subscribers receive the published message.

In order to send a message to a queue, or publish a message to a topic, a client needs to connect to the JMS provider. Likewise, to receive a message from a queue or topic, a client needs to connect to the JMS provider. Connection factories are the mechanism by which clients connect to the JMS provider. Connection factories are the starting point for any JMS client activity.

Queues, topics and connection factories are all referred to as administered objects by the JMS specification. This means that these objects cannot be created by the interfaces available in the JMS API. Rather, they must be administered in a JMS provider specific way using a provider specific mechanism. The only thing that is common across JMS providers is that administered objects are bound into JNDI for subsequent access by JMS clients. Where in the JNDI namespace administered objects are bound is JMS provider specific. The JMS specification fails to define the administration of destinations and connection factories in a JMS provider independent manner to make it easy for a messaging system to support the JMS API. There are typically major differences between messaging system with regard to installation and administration, so the JMS specification simply acknowledges this fact by leaving installation and administration as a JMS provider specific detail.

Messages

Of central importance to JMS is the notion of a message, after all, the purpose of JMS is allow clients to produce and consume messages. JMS messages are composed of three elements, headers, properties and bodies. Headers are really just standardized properties that every message supports. Properties are arbitrary name to value bindings similar to a Java HashMap. Message bodies are the content of the message that cannot be represented by properties. The various types of messages supported by JMS correspond to common types of body content formats that are used in message systems.

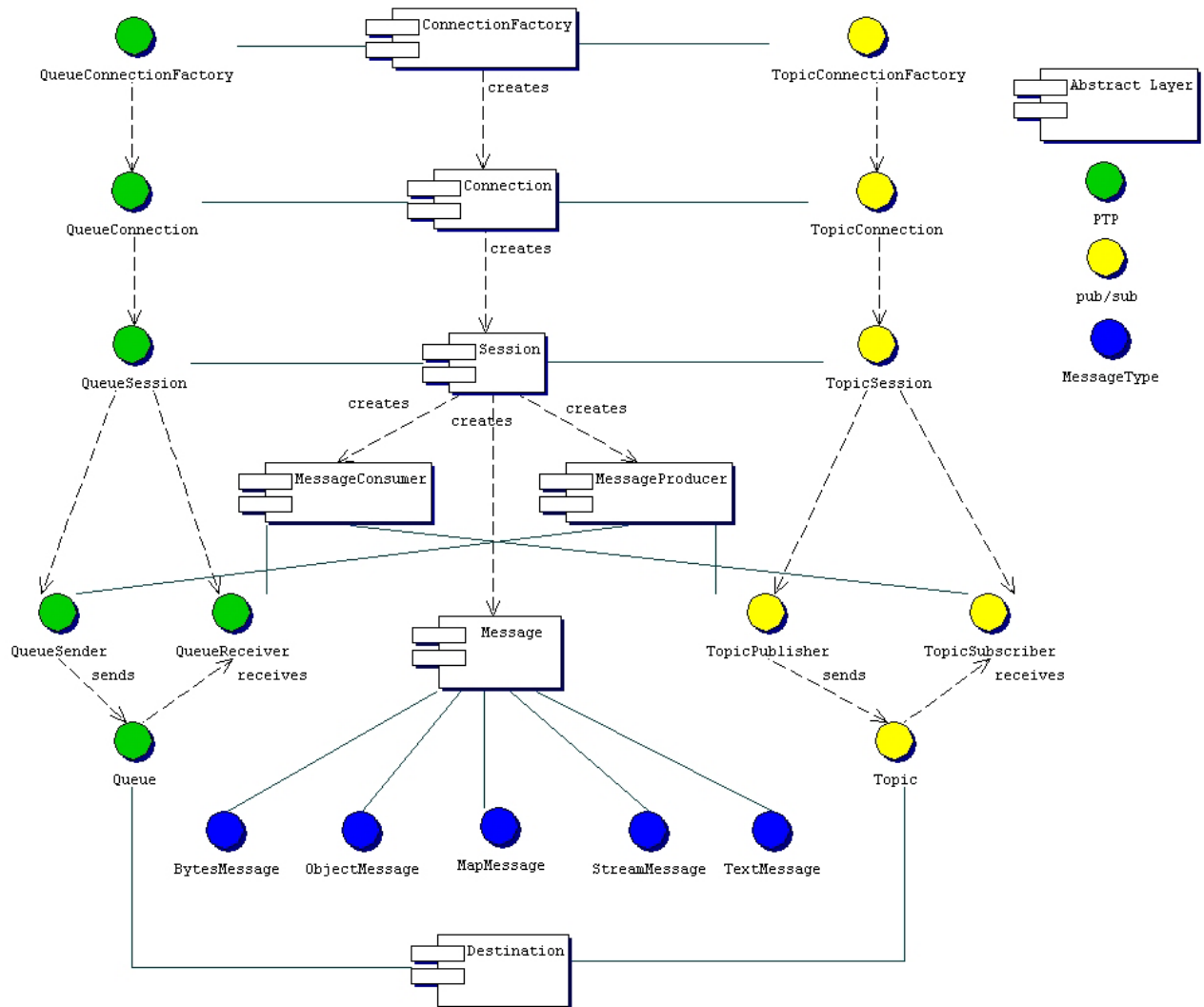
The JMS provider

The responsibility of the JMS provider is to provide an implementation of the JMS API. A typical a JMS provider is an established middleware vendor that provides a JMS implementation for integration with existing middleware services. Alternatively, as is the case with JBossMQ, the JMS provider is a Java messaging system built on lower level Java constructs for the sole purpose of providing a JMS compatible messaging system. Providers will vary in any number of qualitative and quantitative aspects ranging from administration ease of use to performance, scalability and reliability.

The JMS API

In this section we'll cover the interfaces that correspond to the architectural elements we discussed abstractly in the previous section. The JMS API itself is rather small, consisting of a single `javax.jms` package with 44 interfaces, 2 classes and 13 exceptions. Roughly half of the interfaces deal with PTP while the other half deal with pub/sub. Figure 4-1 gives an overview of the interfaces in the `javax.jms` package and their relationships. Each component type has a message model independent interface that is represented by the white rectangular elements. These are divided into PTP specific interfaces represented as green circles and pub/sub specific interfaces represented as yellow circles. There are no PTP or pub/sub specific messages. Rather, there are different types of messages based on the body types that are highlighted as blue circles.

Figure 4-1, An overview of the key component interfaces in the `javax.jms` package.



Connection factories and message destinations interfaces

At the top of Figure 4-1 are the entry point interfaces to JMS, the connection factories. A client initiates a JMS session by obtaining either a [javax.jms.QueueConnectionFactory](#) or a [javax.jms.TopicConnectionFactory](#) depending on whether a PTP to pub/sub model is desired. The sole function of a [QueueConnectionFactory](#) is to create [javax.jms.QueueConnection](#) objects. The sole function of a [TopicConnectionFactory](#) is to create [javax.jms.TopicConnection](#) objects. The creation of a connection can be performed with or without presenting authentication information in the form of a username and password. Whether or not authentication is required is a property of the [ConnectionFactory](#) that was setup by the JMS administrator.

A connection is effectively a handle to a JMS provider that cannot be used to send or receive messages. Its two primary purposes are controlling the flow of messages, which can be started and stopped, and the creation of sessions. A [QueueConnection](#) creates [javax.jms.QueueSessions](#) while a [TopicConnection](#) creates [javax.jms.TopicSessions](#). A session is a lightweight single-threaded context for producing and consuming messages. Sessions are a key object in JMS that provides several functions:

- A Session is a factory for message producers([javax.jms.QueueSender](#) & [javax.jms.TopicPublisher](#)) and message consumers ([javax.jms.QueueReceiver](#) & [javax.jms.TopicSubscriber](#)).
- A Session acts as a factory for the various JMS message objects.
- A Session supports a single series of transactions that combine work spanning its producers and consumers into atomic units.
- A Session defines a serial order for the messages it consumes and the messages it produces.
- A Session retains messages it consumes until they have been acknowledged.
- A Session serializes execution of message listeners registered with its message consumers.

A JMS client uses a [Session](#) to create the JMS messages it sends as well as to create message consumers and producers.

If a client is interested in sending messages it will create a [QueueSender](#) from a [QueueSession](#) for PTP, or a [TopicPublisher](#) from a [TopicSession](#) for pub/sub. Both interfaces are subinterfaces of [javax.jms.MessageProducer](#) that add methods for sending messages.

If a client is interested in receiving messages it will create a QueueReceiver from a QueueSession for PTP, or a TopicSubscriber from a TopicSession for pub/sub. Both interfaces extend the javax.jms.MessageConsumer interface and simply add an accessor for the destination from which they receive messages.

A message destination in JMS is represented in the most abstract form by the javax.jms.Destination interface. This is a tagging interface that contains no methods. A Destination object encapsulates a provider-specific address. There are two subinterfaces of Destination that correspond to the two types of messaging paradigms. The destination interface for the PTP messaging paradigm is javax.jms.Queue. The destination interface for the pub/sub messaging paradigm is javax.jms.Topic. Both Queue and Topic are interfaces that simply provide the ability to get the name of the queue or topic and display the destination as a string. The Queue and Topic interfaces serve as opaque addresses that are passed to other JMS API interface methods.

Messages

Messages are the fundamental objects used to pass information between JMS clients. The javax.jms.Message interface is the root interface of all other messages and defines accessors for all standard message headers and typed property accessors. The property accessors allow you to retrieve a named property as a native Java type, a java.lang.String, or a java.lang.Object. The JMS API defines a limited support for type conversion of property values, and all properties can be retrieved as a java.lang.String.

There are five subinterfaces of Message that correspond to the five different types of bodies or content. The five subtypes are:

javax.jms.BytesMessage: The message body contains a stream of raw bytes. This message type is for literally encoding a body to match an existing message format. In many cases, it is possible to use one of the other body types, which are easier to use. Although the JMS API allows the use of message properties with byte messages, they are typically not used, since the inclusion of properties may affect the format.

javax.jms.MapMessage: The message body contains a set of name-value pairs, where names are String objects, and values are Java primitives. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.

javax.jms.ObjectMessage: The message body contains a java.io.Serializable Java object.

javax.jms.StreamMessage: The message body contains a stream of primitive Java values. It is written and read sequentially using methods similar to the java.io.DataInputStream and java.io.DataOutputStream classes.

java.jms.TextMessage: The message body contains a String object. The inclusion of this message type in the JMS specification is based on the assumption that XML will become a popular mechanism for representing content.

A JMS client creates messages from a javax.jms.Session instance. The Session class is used as the factory for messages in JMS, and it is the only way defined by the JMS specification for obtaining a message.

A PTP Example

Let's bring all of the JMS API interfaces together in the form of a simple PTP example. We will illustrate the typical steps required by a JMS client to send and receive messages in the example. Listing 4-1 provides a complete JMS client example program that sends a TextMessage to a Queue and asynchronously receives the message from the same Queue. The source code for this example can be found in the `src/main/org/jboss/chap5/ex1` directory on the book CD.

Listing 4-1, A simple PTP example that demonstrates the basic steps for a JMS client.

```

1: package org.jboss.chap4.ex1;
3: import javax.jms.JMSException;
4: import javax.jms.Message;
5: import javax.jms.MessageListener;
6: import javax.jms.Queue;
7: import javax.jms.QueueConnection;
8: import javax.jms.QueueConnectionFactory;
9: import javax.jms.QueueReceiver;
10: import javax.jms.QueueSender;
11: import javax.jms.QueueSession;
12: import javax.jms.TextMessage;
13: import javax.naming.InitialContext;
14: import javax.naming.NamingException;
16: import EDU.oswego.cs.dl.util.concurrent.CountDown;
18: /** A complete JMS client example program that sends a
19: TextMessage to a Queue and asynchronously receives the
20: message from the same Queue.
22: @author Scott.Stark@jboss.org
23: @version $Revision:$
24: */
25: public class SendRecvClient
26: {
27:     static CountDown done = new CountDown(1);
28:     QueueConnection conn;
29:     QueueSession session;
30:     Queue que;
32:     public static class ExListener implements MessageListener
33:     {
34:         public void onMessage(Message msg)
35:         {
36:             done.release();

```

```

37:         TextMessage tm = (TextMessage) msg;
38:         try
39:         {
40:             System.out.println("onMessage, recv text="
41:                 + tm.getText());
42:         }
43:         catch(Throwable t)
44:         {
45:             t.printStackTrace();
46:         }
47:     }
48: }
50: public void setupPTP()
51:     throws JMSEException, NamingException
52: {
53:     InitialContext iniCtx = new InitialContext();
54:     Object tmp = iniCtx.lookup("QueueConnectionFactory");
55:     QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
56:     conn = qcf.createQueueConnection();
57:     que = (Queue) iniCtx.lookup("queue/testQueue");
58:     session = conn.createQueueSession(false,
59:         QueueSession.AUTO_ACKNOWLEDGE);
60:     conn.start();
61: }
63: public void sendRecvAsync(String text)
64:     throws JMSEException, NamingException
65: {
66:     System.out.println("Begin sendRecvAsync");
67:     // Setup the PTP connection, session
68:     setupPTP();
69:     // Set the async listener
70:     QueueReceiver recv = session.createReceiver(que);
71:     recv.setMessageListener(new ExListener());
72:     // Send a text msg
73:     QueueSender send = session.createSender(que);
74:     TextMessage tm = session.createTextMessage(text);
75:     send.send(tm);
76:     System.out.println("sendRecvAsync, sent text="
77:         + tm.getText());
78:     send.close();
79:     System.out.println("End sendRecvAsync");
80: }
82: public void stop() throws JMSEException
83: {
84:     conn.stop();
85:     session.close();
86:     conn.close();
87: }
89: public static void main(String args[]) throws Exception
90: {
91:     SendRecvClient client = new SendRecvClient();
92:     client.sendRecvAsync("A text msg");
93:     client.done.acquire();
94:     client.stop();

```

```

95:         System.exit(0);
96:     }
98: }

```

The key steps performed by the client in the order of execution are:

- Lines 91-92, the main entry point creates a SendRecvClient instance and then invokes the sendRecvAsync method passing in the text that should be sent as the body of the TextMessage.
- Line 68, the first step of the sendRecvAsync method is to perform a typical PTP JMS setup by calling the setupPTP method.
- Lines 53-55, of the setupPTP method retrieve the QueueConnectionFactory administered object from JNDI under the name “QueueConnectionFactory”. The location of the QueueConnectionFactory is a configurable parameter as we will see in a later section on the configuration of JBossMQ.
- Line 56, creates a QueueConnection from the QueueConnectionFactory to connect to the JMS provider.
- Line 57, lookup a Queue using the name “queue/testQueue”. This is one of the Queue destinations defined in the default JBossMQ configuration.
- Lines 58-59, create a non-transacted QueueSession that automatically acknowledges messages.
- Line 60, the QueueConnection is started so that message delivery will be enabled.
- Line 70, back in the sendRecvAsync method, creates a QueueReceiver to be used for the asynchronous receipt of messages sent to que.
- Line 71 assigns the MessageListener instance that will be notified of messages sent to que. The JBossMQ provider will invoke the ExListener class’s onMessage method as messages are sent to que.
- Line 73, a QueueSender is created to use for sending messages to que.
- Line 74, a TextMessage is created with the text body passed into the sendRecvAsync method.
- Line 75, shows the send of the TextMessage to que via the QueueSender.

- Line 78, close the QueueSender as we are done with it. It is a good practice to close JMS objects as it allows the JMS provider to reclaim any resources as soon as they are no longer needed.
- Line 93, back in main, wait for the message sent by the main thread to be received by the listener. Since the message delivery is done asynchronously we have to introduce synchronization code to know when the ExListener receives the message sent on line 75.

Ensure that you have a JBoss server running, and run the example by invoking Ant with the chap=4 and ex=1 property values and the output you see should look similar to the following:

```
examples 770>ant -Dchap=4 -Dex=1 run-example
Buildfile: build.xml
run-example:
run-example1:
    [java] Begin sendRecvAsync
    [java] sendRecvAsync, sent text=A text msg
    [java] End sendRecvAsync
    [java] onMessage, recv text=A text msg
```

JMS and J2EE

The addition of the JMS API enhances the J2EE platform by simplifying enterprise development due to the support JMS has for loosely coupled, reliable, asynchronous messaging between J2EE components and external non-J2EE message based systems. JBoss and JBossMQ support the majority of the J2EE 1.3 JMS requirements. This includes:

- The ability for applications, EJB components, and web components to send or synchronously receive JMS messages. Applications can also receive JMS messages asynchronously.
- Support for EJB 2.0 message-driven beans (MDB) that allow a business component to asynchronously consume JMS messages and concurrent processing of messages by MDBs.
- Message sends and receives can participate in distributed transactions.
- Support for accessing JMS administered objects through the application component naming context or ENC.

Message driven beans

A new type message driven enterprise Java bean (MDB) was added in EJB 2.0. It is a variant of the stateless session bean that can be used to process JMS messages

asynchronously. Unlike other EJB types, an MDB does not have a remote or home interface, as clients do not directly invoke it. Rather, an MDB implements the `javax.jms.MessageListener` interface and processes JMS messages delivered to its `onMessage` method. JBoss supports MDBs by using the JBossMQ implementation of the optional JMS application server facilities (ASF) described in section 8 of the JMS 1.0.2 specification.

A JMS provider ASF implementation of `javax.jms.Connection` objects supports the creation of `javax.jms.ConnectionConsumer` instances. The `ConnectionConsumer` interface allows for a pool of `javax.jms.MessageListener` instances that can concurrently consume JMS message. We'll discuss the JBossMQ ASF implementation in more detail in the JBossMQ architecture overview.

An MDB example

We'll extend our use of JMS with JBoss by presenting an MDB example. The example is a modification of that presented in Listing 1, which sends multiple text messages to a MDB and receives modified text messages asynchronously from the MDB. First we will show the MDB and its deployment descriptors, and then move on to the JMS client.

The first step is the coding of the MDB. Listing 4-2 provides the MDB implementation for `example2`.

Listing 4-2, The example2 message driven EJB.

```
package org.jboss.chap4.ex2;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.EJBException;
import javax.jms.JMSException;
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;

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()
    throws EJBException
{
    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 JMSEException, 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 JMSEException
{
    System.out.println("TextMDB.sendReply, this="+hashCode()
        +", dest="+dest);
    QueueSender sender = session.createSender(dest);
    TextMessage tm = session.createTextMessage(text);
    sender.send(tm);
    sender.close();
}
}

```

Items of note in the TextMDB implementation include:

- The TextMDB implements both the javax.ejb.MessageDrivenBean interface as well as the javax.jms.MessageListener interface. The MessageDrivenBean methods correspond to the MDB life-cycle callback methods invoked by the MDB container, while the MessageListener.onMessage method is the JMS message delivery callback method seen in all JMS message consumers.
- In the ejbCreate method, the TextMDB invokes the setupPTP to perform the same steps to connect to the JMS provider that we saw the example 1 client perform. The ejbCreate method is called when an MDB is added to the pool of MessageListeners available to handle messages.
- In the ejbRemove method, the TextMDB closes the JMS resources that were allocated in ejbCreate. The ejbRemove method is called when an MDB is no longer needed in the MessageListener pool.
- In the onMessage method, the TextMDB retrieves the text of the TextMessage it receives and annotates the text with its hashCode to indicate which MDB received the message. It then retrieves the Queue to which replies should be sent using the

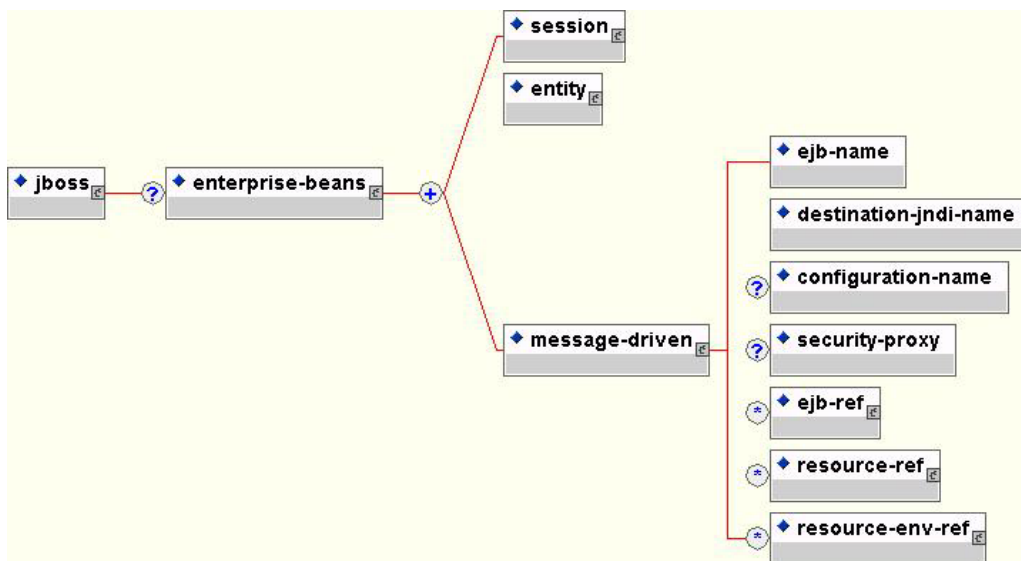
TextMessage.getJMSReplyTo method. The sendReply method is invoked to return the annotated text to the indicated Queue.

- In the sendReply method, a QueueSender is created from the TextMDB QueueSession object with the reply to dest as the target Queue, a new TextMessage is created from the annotated text value, the message is sent, and finally the sender is closed.
- All key methods of the TextMDB include a System.out.println statement indicating the method and TextMDB instance hashCode. This will allow us to see how the MDB container manages MDB instances in response to JMS message arrivals.

Next, we need to create the standard `ejb-jar.xml` and `jboss.xml` xml deployment descriptors for the MDB. Often for simple EJBs one does not need a `jboss.xml` deployment descriptor. There are two reasons why a `jboss.xml` descriptor is needed for this MDB. The first reason is that the `ejb-jar.xml` descriptor does not specify what queue the `TextMDB` listens to. The queue must be specified by the bean deployer, and this done using the `jboss.xml` descriptor. Because of this, MDBs must always include a `jboss.xml` descriptor for the specification of the destination name they are to receive messages from.

Figure 4.2 shows the MDB portion of the DTD for the `jboss.xml` descriptor. Under the message-driven element in Figure 4.2, the only element unique to MDBs is the `destination-jndi-name` element. The value of this element is the deployment environment JNDI name of the Queue or Topic the MDB is to listen to for messages.

Figure 4-2, The MDB portion of the DTD for the `jboss.xml` descriptor.



The second reason the `jboss.xml` descriptor is necessary in this example is for the specification of the mapping between the JMS QueueConnectionFactory reference made in the `ejb-jar.xml` descriptor to the deployed JNDI name of the JBossMQ QueueConnectionFactory. This is done using the `resource-ref` element as we have seen previously in the JBossNS chapter.

Ok, so let's look at the MDB deployment descriptors. Listing 4-3 provides both `ejb-jar.xml` and `jboss.xml` deployment descriptors.

Listing 4-3, the TextMDB ejb-jar.xml and jboss.xml deployment 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.chap4.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>
      <resource-ref>
        <res-ref-name>jms/QCF</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
<?xml version="1.0"?>

<!--The jboss.xml descriptor -->
<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>QueueConnectionFactory</jndi-name>
      </resource-ref>
    </message-driven>
  </enterprise-beans>
</jboss>
```

The `ejb-jar.xml` descriptor declares that the `org.jboss.chap5.example2.TextMDB` will receive messages from a `java.jms.Queue` with container-managed transactions and message auto acknowledgement. It also declares that the `java:comp/env/jms/QCF` ENC binding should reference a `javax.jms.QueueConnectionFactory`.

The `jboss.xml` descriptor fills in the missing pieces. The `destination-jndi-name` element defines that the `Queue` from which the `TextMDB` will receive messages is located under the JNDI name “`queue/B`”. The `resource-ref` element links the `TextMDB` `java:comp/env/jms/QCF` ENC entry to the location of the JBossMQ `QueueConnectionFactory`.

That completes the `example2` MDB. The `example2` JMS client is a slight variation of the `example1` client. Listing 4-4 gives the `example2` client code.

Listing 4-4, The example2 JMS client.

```
package org.jboss.chap4.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 TestMDB from QueueB.

    @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 JMSEException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("QueueConnectionFactory");
    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 JMSEException, 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 JMSEException
{
    conn.stop();
}

```

```

    }

    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 two differences between the example1 client and this example2 client are the use of two Queues and the sending and receipt of multiple messages. In the setupPTP method, two Queues are obtained from JNDI, “queue/A” and “queue/B”. The “queue/A” destination will be used to receive messages and the “queue/B” destination will be where the client sends messages. The sendRecvAsync method also performs 10 TextMessage sends rather than 1 as performed in example1. This means that the main method must wait for 10 messages to be received rather than just 1. This is achieved by initializing the EDU.oswego.cs.dl.util.concurrent.CountDown instance with a value of 10 rather than 1.

To test the example2 MDB, ensure that you have a JBoss server running, and invoke Ant with the chap=4 and ex=2 property values. The output you see in the client console should look similar to the following:

```

examples 635>ant -Dchap=4 -Dex=2 run-example
Buildfile: build.xml

run-example:

prepare:

example2-jar:

run-example2:
...
    [copy] Copying 1 file to G:\JBoss-2.4.5_Tomcat-4.0.3\jboss\deploy
    [echo] Waiting for deploy...
    [java] Begin sendRecvAsync
    [java] onMessage, rcv text=A text msg#0processed by: 4395216
    [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] onMessage, rcv text=A text msg#2processed by: 4395216
    [java] onMessage, rcv text=A text msg#3processed by: 494668
    [java] onMessage, rcv text=A text msg#4processed by: 494668
    [java] onMessage, rcv text=A text msg#5processed by: 4395216

```

```

[java] onMessage, recv text=A text msg#6processed by: 4395216
[java] onMessage, recv text=A text msg#7processed by: 494668
[java] onMessage, recv text=A text msg#8processed by: 494668
[java] onMessage, recv text=A text msg#9processed by: 494668
[java] onMessage, recv text=A text msg#1processed by: 494668
[java] sendRecvAsync, sent text=A text msg#6
[java] onMessage, recv text=A text msg#0processed by: 494668
[java] onMessage, recv text=A text msg#2processed by: 494668
[java] onMessage, recv text=A text msg#3processed by: 494668
[java] onMessage, recv text=A text msg#4processed by: 494668
[java] onMessage, recv text=A text msg#5processed by: 4395216
[java] sendRecvAsync, sent text=A text msg#7
[java] onMessage, recv text=A text msg#6processed by: 4395216
[java] onMessage, recv text=A text msg#7processed by: 4395216
[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#8processed by: 4395216
[java] onMessage, recv text=A text msg#9processed by: 494668

```

BUILD SUCCESSFUL

Total time: 3 seconds

There should also be output on the JBoss server console similar to the following:

```

[Container factory] Deploying:file:/.../deploy/example2.jar/
[Verifier] Verifying file:/.../Default/example2.jar/ejb1002.jar
[Container factory] Deploying TextMDB
[ContainerManagement] Initializing
[ContainerManagement] Initialized
[ContainerManagement] Starting
[ContainerManagement] Started
[Container factory] Deployed application: file:/.../example2.jar/
[J2EE Deployer Default] J2EE application: example2.jar is deployed.
[Default] TextMDB.ctor, this=494668
[Default] TextMDB.setMessageDrivenContext, this=494668
[Default] TextMDB.ejbCreate, this=494668
[Default] TextMDB.ctor, this=4395216
[Default] TextMDB.onMessage, this=494668
[Default] TextMDB.setMessageDrivenContext, this=4395216
[Default] TextMDB.sendReply, this=494668, dest=QUEUE.A
[Default] TextMDB.ejbCreate, this=4395216
[OILClientIL] ConnectionReceiverOILClient is connecting to:
172.17.66.54:1791
[Default] TextMDB.onMessage, this=4395216
[Default] TextMDB.sendReply, this=4395216, dest=QUEUE.A
[Default] TextMDB.onMessage, this=4395216
[Default] TextMDB.sendReply, this=4395216, dest=QUEUE.A
[Default] TextMDB.onMessage, this=494668
[Default] TextMDB.sendReply, this=494668, dest=QUEUE.A
[Default] TextMDB.onMessage, this=494668
[Default] TextMDB.sendReply, this=494668, dest=QUEUE.A

```

```

[Default] TextMDB.onMessage, this=4395216
[Default] TextMDB.sendReply, this=4395216, dest=QUEUE.A
[Default] TextMDB.onMessage, this=4395216
[Default] TextMDB.sendReply, this=4395216, dest=QUEUE.A
[Default] TextMDB.onMessage, this=494668
[Default] TextMDB.sendReply, this=494668, dest=QUEUE.A
[Default] TextMDB.onMessage, this=494668
[Default] TextMDB.sendReply, this=494668, dest=QUEUE.A
[Default] TextMDB.onMessage, this=494668
[Default] TextMDB.sendReply, this=494668, dest=QUEUE.A

```

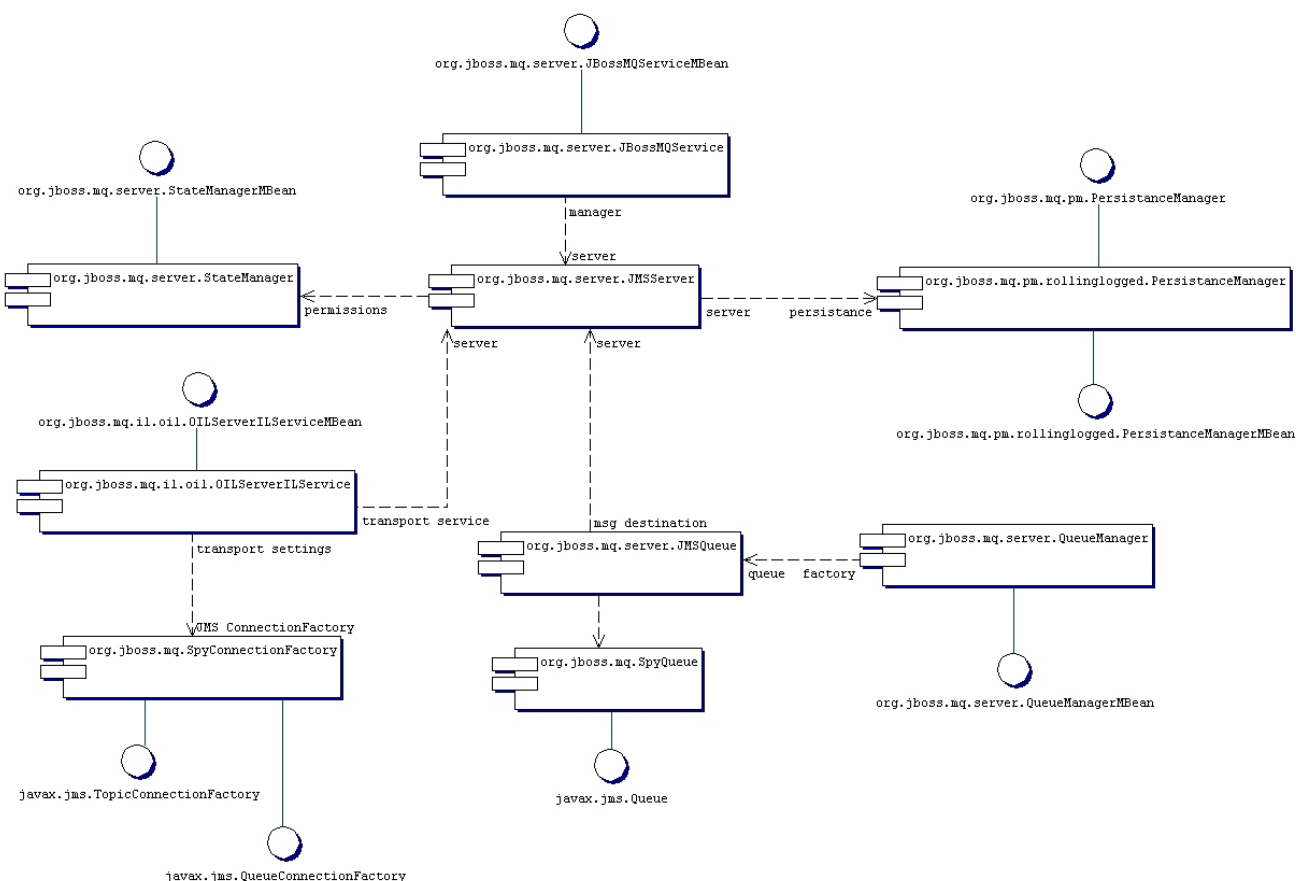
It can be seen from the server console messages that two MDB instances are created to handle the processing of the 10 client messages. This is also evident in the client console message due to the two unique hash code values contained in the “processed by: NNNNN” message annotations.

An Overview of the JBossMQ Architecture

In this section we will introduce some of the key elements of the JBossMQ server architecture. This will be a relatively high-level overview to help understand how JBossMQ operates and integrates with the JBoss server. The overview will focus on the JBossMQ related MBean services that are responsible for the integration of JBossMQ into the JBoss server.

We’ll begin with the core components that make up the JBossMQ server. Figure 4-3, An overview of the key components in the JBossMQ server components and their relationships, provides an overview of the key JBossMQ services and their high-level relationships.

Figure 4-3, An overview of the key components in the JBossMQ server components and their relationships.



In Figure 4-3, An overview of the key components in the JBossMQ server components and their relationships. we that the org.jboss.mq.server.JMSServer is the heart of the server components as it is referenced by nearly every other component. The JMSServer is core of the JMS implementation. The life-cycle of the JMSServer is exposed via the org.jboss.mq.server.JBossMQService MBean. This MBean performs the startup and shutdown of the JMSServer in response to JBoss server life-cycle method invocations. The JBossMQService MBean also exposes operations for creating and destroying Queues and Topics and runtime.

Figure 4-3, An overview of the key components in the JBossMQ server components and their relationships. also shows that the JMSServer delegates user permission checks to the org.jboss.mq.server.StateManager MBean service. This MBean uses an XML file to describe user to password mapping as well as the durable subscriptions.

The handling of persistent messages is seen to be delegated to an implementation of the `org.jboss.mq.pm.PersistanceManager` interface. The choice of which persistence manager the `JMSServer` uses is based on which MBean service is configured. In Figure 4-3, An overview of the key components in the JBossMQ server components and their relationships., the

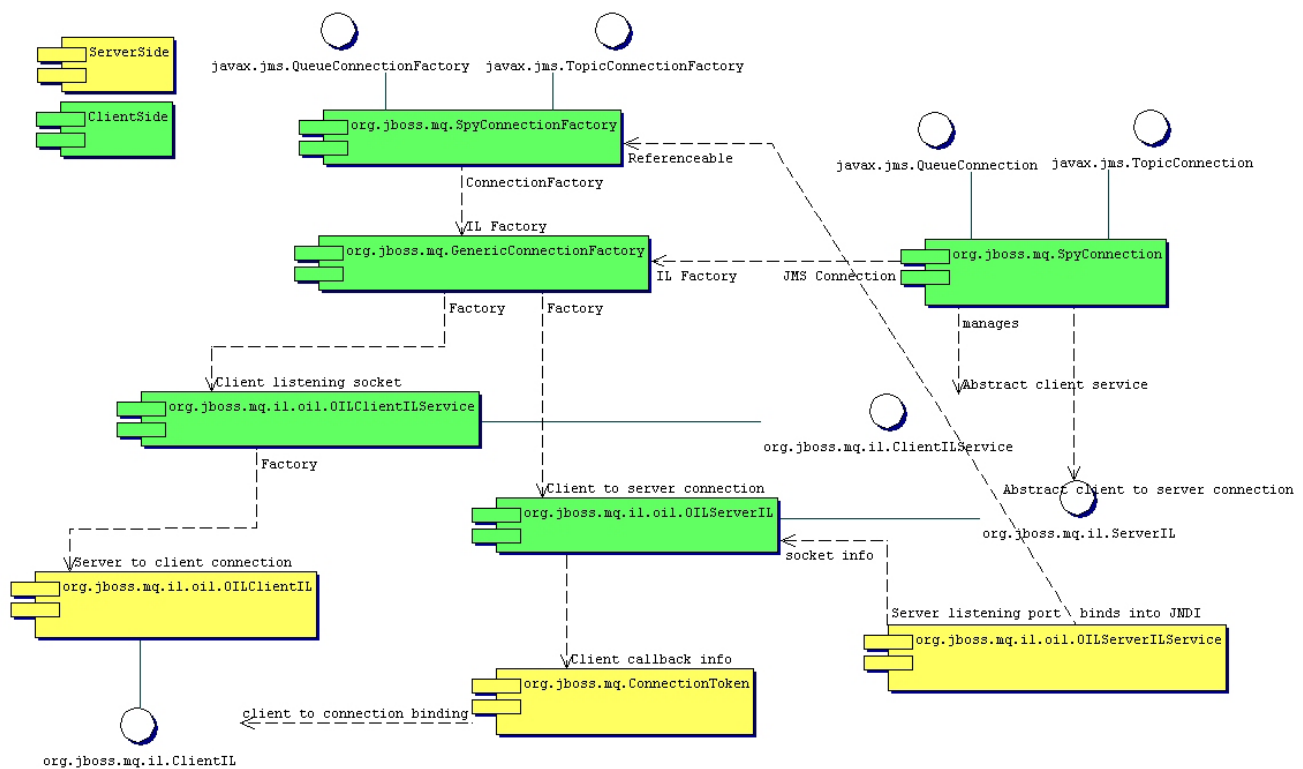
org.jboss.mq.pm.rollinglogged.PersistanceManager is shown. This is a file based persistence manager that is the current default PersistanceManager implementation. We'll look at some of the optional persistence manager MBeans when we discuss the JBossMQ configuration in detail.

Recall that Queues and Topics are administered objects for which there is no management API in the JMS specification. Although the JBossMQService exposes operations for destination administration, it would be nice to be able to specify the Queues and Topics as part of the server configuration. This is the purpose of the org.jboss.mq.server.QueueManager MBean shown in Figure 4-3. Each QueueManager MBean creates and binds an org.jboss.mq.SpyQueue into JNDI as an object under the "queue" context using the name assigned as the QueueName attribute. Topics are managed in a similar way by the org.jboss.mq.server.TopicManager MBean, which is not shown in Figure 4-3.

The last core component is the invocation layer service MBean. In Figure 4-3 the optimized invocation layer (OIL) MBean is shown as the org.jboss.mq.il.oil.OILServerILService component. The invocation layer components are responsible for exposing the JMSServer to clients via some message transport. JBossMQ supports a number of different message transports, each of which is configured by a different MBean service. We will look at all of the available message transport services when we will look at the JBossMQ configuration details in the next section. Figure 4-3 shows the default invocation layer OILServerILService MBean, but none of the related components that deal with message transport.

Let's investigate the invocation layer architecture by drilling down into OILServerILService to see its associated message transport components. This gives us a view as presented in Figure 4-4.

Figure 4-4, An overview of the key components in the optimized invocation layer (OIL) transport implementation.



The OIL message transport protocol is a custom socket based protocol that uses a socket on the JMSServer and another on the JMS client to send messages between the two. The JMSServer side socket allows the client to send messages to the JMSServer while the client side socket allows the server to asynchronously deliver messages to the client.

In Figure 4-4 the yellow shaded components are those that operate within the JMSServer VM while the green shaded components operate in the JMS client VM. The OILServerILService is the MBean that manages the setup of the OIL components. Its responsibilities include starting the JMSServer side listening socket that will accept JMS client connections as well as binding a org.jboss.mq.SpyConnectionFactory into JNDI for access by JMS clients. The SpyConnectionFactory instance that the OILServerILService binds into JNDI is associated with an org.jboss.mq.GenericConnectionFactory that handles the creation of the transport specific versions of the org.jboss.mq.il.ServerIL and org.jboss.mq.il.ClientILService interfaces. The GenericConnectionFactory is setup by the OILServerILService to return OIL specific implementations of the transport independent ServerIL and ClientILService interfaces used by the org.jboss.mq.SpyConnection. This is the mechanism used by every invocation layer MBean service to establish a ConnectionFactory for the server invocation layer protocol. This means that the transport protocol a JMS client uses is determined by which ConnectionFactory the client accesses. If a client wants to use a particular type of transport, the client needs to know what JNDI name the invocation layer MBean service binds its ConnectionFactory under.

Although the SpyConnectionFactory is bound into JNDI inside the JMS Server VM, it is accessed by the client within its VM when the client looks up a QueueConnectionFactory or TopicConnectionFactory in order to create a Connection to the JMS provider. When the client creates a Connection, the creation of the SpyConnection causes the creation of org.jboss.mq.il.oil.OILClientILService and org.jboss.mq.il.oil.OILServerIL instances in the client VM. The OILClientILService creates a server socket that the JMS Server will use to push messages to the client. The information on how to push messages to the client is sent to the server using the org.jboss.mq.ConnectionToken class. This is a serializable pair of a clientId and an org.jboss.mq.il.oil.OILClientIL instance. When the OILClientIL is used in the JMS Server VM, it establishes a connection back to the OILClientILService if none exists using the port and address information set by the OILClientILService.

The OILServerIL instance functions similarly to the OILClientIL, but it was created in the JMS Server and has the port and address of the OILServerILService that created it. When messages are sent to the JMS Server over the SpyConnection, the OILServerIL associated with the connection is used and this will cause a socket connection to be made to the OILServerILService if none currently exists.

This interaction between server and client side components based on factories for interfaces found in the org.jboss.mq.il package is the pattern used by all invocation layer implementations. All that varies are the protocol dependent details of how messages are sent to the JMS Server side component using the ServerIL implementation, and how messages are pushed back to the JMS client using the ClientIL implementation.

JBossMQ Application Server Facilities

Up to this point we have looked at the standard JMS client/server architecture. The JMS specification defines an advanced set of interfaces that allow for concurrent processing of a destination's messages, and collectively this functionality is referred to as application server facilities (ASF). Two of the interfaces that support concurrent message processing, javax.jms.ServerSessionPool and javax.jms.ServerSession, must be provided by the application server in which the processing will occur. Thus, the set of components that make up the JBossMQ ASF involves both JBossMQ components as well as JBoss server components. A diagram of ASF components is presented in Figure 4-5.

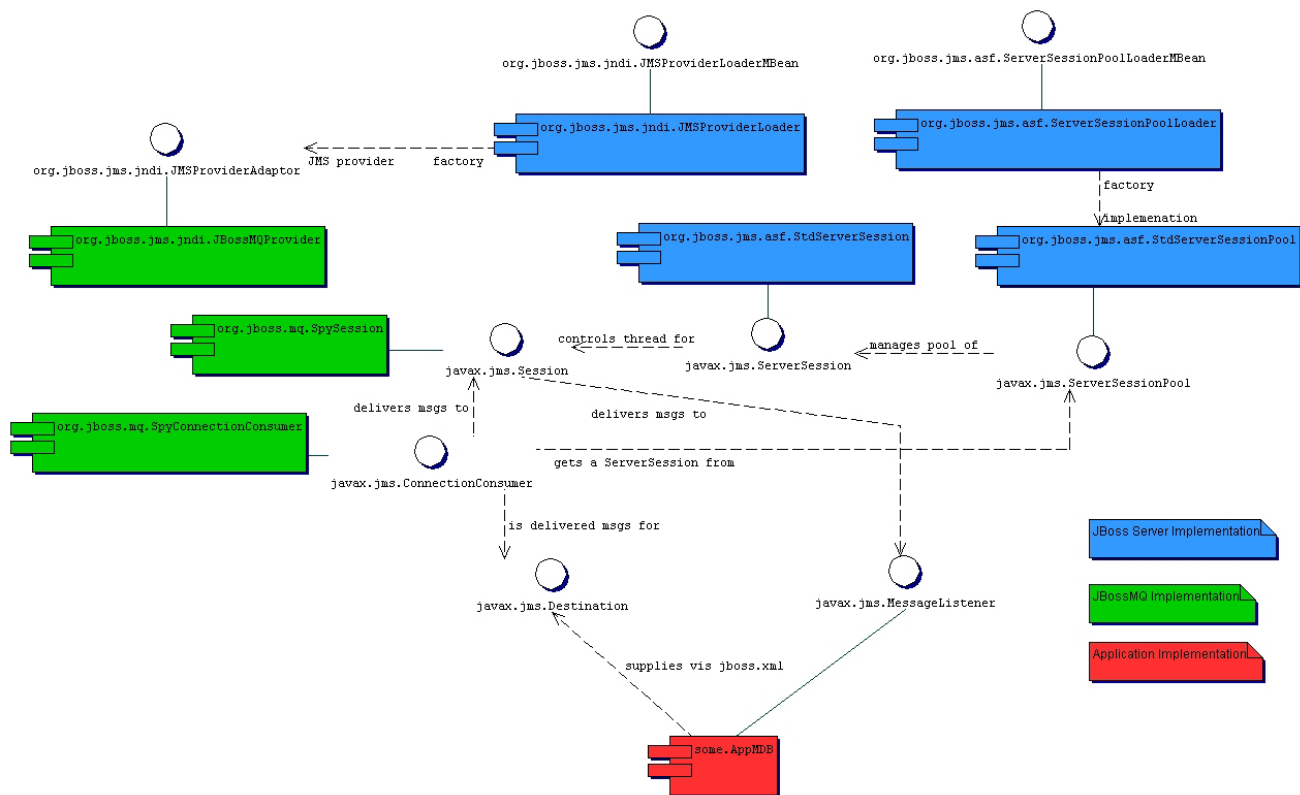


Figure 4-5, The JBoss and JBossMQ ASF components and their relationships.

In Figure 4-5, the white circles represent the JMS interfaces that constitute the ASF. The shaded components represent the implementor of the interfaces and which domain they fall into. The blue components are implemented by the JBoss server, the green components are implemented by the JBossMQ provider, and the red components are implemented by the JMS application that wishes to perform the concurrent message processing. Here the application is illustrated as an MDB because this is the natural use case of the ASF in J2EE. The JBoss server implements MDBs by utilizing the JMS service's ASF to concurrently process messages sent to MDBs.

The responsibilities of the ASF domains are well defined by the JMS specification and so we won't go into a discussion of how the ASF components are implemented. Rather, we want to discuss how ASF components used by the JBoss MDB layer are integrated using MBeans that allow either the application server interfaces, or the JMS provider interfaces to be replaced with alternate implementations.

Let's start with the `org.jboss.jms.jndi.JMSProviderLoader` MBean show in Figure 4-5. This MBean is responsible for loading an instance of the `org.jboss.jms.jndi.JMSProviderAdaptor` interface into the JBoss server and binding it into JNDI. The `JMSProviderAdaptor` interface is an abstraction that defines how to get the root JNDI context for the JMS provider, and an

interface for getting and setting the JNDI names for the Context.PROVIDER_URL for the root InitialContext, and the QueueConnectionFactory and TopicConnectionFactory locations in the root context. This is all that is really necessary to bootstrap use of a JMS provider. By abstracting this information into an interface, alternate JMS ASF provider implementations can be used with the JBoss MDB container. The `org.jboss.jms.jndi.JBossMQProvider` is the default implementation of `JMSProviderAdaptor` interface, and provides the adaptor for the JBossMQ JMS provider. To replace the JBossMQ provider with an alternate JMS ASF implementation, simply create an implementation of the `JMSProviderAdaptor` interface and configure the `JMSProviderLoader` with the class name of the implementation. We'll see an example of this in the configuration section.

In addition to being able to replace the JMS provider used for MDBs, you can also replace the `javax.jms.ServerSessionPool` interface implementation. This is possible by configuring the class name of the `org.jboss.jms.asf.ServerSessionPoolFactory` implementation using the `org.jboss.jms.asf.ServerSessionPoolLoader` MBean `PoolFactoryClass` attribute. The default `ServerSessionPoolFactory` factory implementation is the JBoss `org.jboss.jms.asf.StdServerSessionPoolFactory` class.

Configuring JBossMQ

We'll conclude this chapter by detailing the configurable aspects of JBossMQ. We have already seen a number of the JBossMQ MBeans in chapter 2 when we discussed the standard MBean services found in the default `jboss.jcml` MBean services configuration file. These will be repeated here for completeness in expanded form where appropriate along with the JBossMQ MBeans that were not presented.

`org.jboss.mq.server.JBossMQService` MBean

The `JBossMQService` MBean embeds the JBossMQ JMS message server into the JBoss server. The JBossMQ server is necessary for EJB 2.0 message driven beans as well as any other JMS client usage. The `JBossMQService` has no configurable attributes.

The `JBossMQService` does define operations for the runtime administration of queues and topics. Note that the configuration of queues or topics created via these operations is not saved across restarts of the JBoss server.

- `public void createQueue(String name) throws Exception`
This creates a `javax.jms.Queue` destination.
- `public void destroyQueue(String name) throws Exception`
This destroys an existing `javax.jms.Queue` destination.

- `public void createTopic(String name) throws Exception`
This creates a `javax.jms.Topic` destination.
- `public void createTopic(String name) throws Exception`
This creates a `javax.jms.Topic` destination.

`org.jboss.mq.server.StateManager` MBean

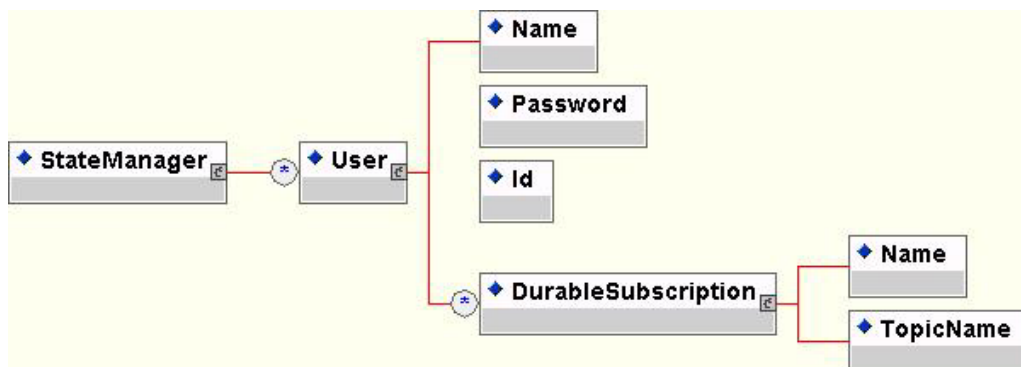
The `StateManager` MBean is a simple user to password and user to durable subscription mapping service. It uses an XML file to obtain this mapping information. In order for a user to create a durable subscription they must have an entry in the `StateManager`.

The configurable attributes of the `StateManager` service are:

- **StateFile:** This is the name of the `StateManager` XML file. The location of the file is resolved relative to the location of the `jboss.jcml` file. If no `StateFile` attribute is specified a default of file named `jbossmq-state.xml` is assumed.

The DTD for the `StateManager` XML file is given graphically in Figure 4-6.

Figure 4-6, The DTD for the `StateManager` user-password, user-durable-subscription XML file



The element meanings are:

- **User/Name:** the username that corresponds to the `Connection.createConnection(username, password)` method.
- **User/Password:** the password that corresponds to the `Connection.createConnection(username, password)` method.
- **User/Id:** the `clientId` that will be associated with the connection for the username.
- **User/DurableSubscription:** a listing of the durable subscriptions associated with the username.

- **User/DurableSubscription/Name:** the name of the durable subscription. This is the value passed in as the name parameter to the `TopicSession.createDurableSubscriber(Topic, name)` method.
- **User/DurableSubscription/TopicName:** the name of the Topic current associated with the durable subscription.

`org.jboss.mq.pm.rollinglogged.PersistenceManager` MBean

The `PersistenceManager` MBean service manages all JBossMQ persistence related services. It is an implementation of the `org.jboss.mq.pm.PersistenceManager` that uses a file store and allows up to 1000 messages to be persisted before it starts discarding the old messages. This is the default persistence manager configured with the JBoss distribution.

The configurable attributes of the service are:

- **DataDirectory:** This is the path to the directory where the service will store its data files.

`org.jboss.mq.pm.file.PersistanceManager` MBean

This is an alternate `PersistenceManager` MBean service implementation that uses a single file per persistent message. This is a very robust implementation, but it is can be quite slow relative to the other implementations.

The configurable attributes of the service are:

- **DataDirectory:** This is the path to the directory where the service will store its data files.

`org.jboss.mq.pm.jdbc.PersistanceManager` MBean

This is an alternate `PersistenceManager` MBean service implementation that stores persistent messages in a JDBC store. It expects a JDBC DataSource that contains two tables that have a schema consistent to the following:

```
CREATE TABLE JMS_MESSAGES
(
  MESSAGEID CHAR(17) NOT NULL,
  DESTINATION VARCHAR(30) NOT NULL,
  MESSAGEBLOB BLOB,
  PRIMARY KEY (MESSAGEID, DESTINATION)
);
CREATE INDEX JMS_MESSAGES_DEST ON JMS_MESSAGES(DESTINATION);
CREATE TABLE JMS_TRANSACTIONS
(
```

```
ID CHAR(17)
)
```

The configurable attributes of the service are:

- **JmsDBPoolName:** The JNDI name of the `javax.sql.DataSource` to use for obtaining the database connections.

org.jboss.mq.il.oil.OILServerILService MBean

The OILServerILService MBean service configures the optimized invocation layer (OIL) transport mechanism, and is the default transport by virtue of the fact that it uses the common “`QueueConnectionFactory`” and “`TopicConnectionFactory`” as the JNDI location of the connection factories that it binds into JNDI.

The configurable attributes of the OILServerILService service are:

- **ConnectionFactoryJNDIRef:** The JNDI name under which the combined `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory` instance will be bound.
- **XAConnectionFactoryJNDIRef:** The JNDI name under which the combined `javax.jms.XAQueueConnectionFactory` and `javax.jms.XATopicConnectionFactory` instance will be bound.
- **ServerBindPort:** the port number the OIL service will listening for client connections on.
- **BindAddress:** interface address the OIL service will bind its listening port on. This is useful for multi-homed hosts.

org.jboss.mq.il.oil.UILServerILService MBean

The UILServerILService MBean service is an invocation layer (OIL) transport mechanism similar to the OIL custom socket transport. The difference is that a single socket connection is used between the client and server and therefore modified transport scheme is used to allow full duplex sending and receiving of messages.

The configurable attributes of the UILServerILService service are:

- **ConnectionFactoryJNDIRef:** The JNDI name under which the combined `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory` instance will be bound.

- **XAConnectionFactoryJNDIRef:** The JNDI name under which the combined javax.jms.XAQueueConnectionFactory and javax.jms.XATopicConnectionFactory instance will be bound.
- **ServerBindPort:** the port number the UIL service will listening for client connections on.
- **BindAddress:** interface address the UIL service will bind its listening port on. This is useful for multi-homed hosts.

org.jboss.mq.il.jvm.JVMServerILService MBean

The JVMServerILService MBean service configures the intra-server VM invocation layer transport mechanism. This is simply an adaptor implementation that incurs only a single method call of overhead in mapping from the ClientIL or ServerIL interfaces onto the JMS server objects. It is a call by reference semantic transport that can only be used by JMS client within the same VM as the JMServer instance.

The configurable attributes of the JVMServerILService service are:

- **ConnectionFactoryJNDIRef:** The JNDI name under which the combined javax.jms.QueueConnectionFactory and javax.jms.TopicConnectionFactory instance will be bound. Note that this should be under the “java:” (e.g., java:/ConnectionFactory) context so that it will not be accessible outside of the server VM since it does not support a remote transport.
- **XAConnectionFactoryJNDIRef:** The JNDI name under which the combined javax.jms.XAQueueConnectionFactory and javax.jms.XATopicConnectionFactory instance will be bound. Note that this should be under the “java:” context so that it will not be accessible outside of the server VM since it does not support a remote transport.

org.jboss.mq.il.rmi.RMIServerILService MBean

The RMIServerILService MBean service configures an RMI based invocation layer transport mechanism. Communication between the client and server use standard Java RMI protocol. This is currently somewhat less efficient than the custom protocol used by the OIL and UIL layers.

The configurable attributes of the RMIServerILService service are:

- **ConnectionFactoryJNDIRef:** The JNDI name under which the combined javax.jms.QueueConnectionFactory and javax.jms.TopicConnectionFactory instance will be bound.
- **XAConnectionFactoryJNDIRef:** The JNDI name under which the combined javax.jms.XAQueueConnectionFactory and javax.jms.XATopicConnectionFactory instance will be bound.

org.jboss.mq.server.TopicManager MBean

The TopicManager MBean service manages a javax.jms.Topic destination. The configurable attributes of the TopicManager service are:

- **TopicName:** The JNDI name under which the javax.jms.Topic will be bound. This is relative to the “topic” context. Thus, a TopicName of myTopic will be located under the JNDI name “topic/myTopic”.

org.jboss.mq.server.QueueManager MBean

The QueueManager MBean service manages a javax.jms.Queue destination. The QueueManager MBean is used to create. The configurable attributes of the QueueManager service are:

- **QueueName:** The JNDI name under which the javax.jms.Queue will be bound. This is relative to the “queue” context. Thus, a QueueName of myQueue will be located under the JNDI name “queue/myQueue”.

org.jboss.jms.jndi.JMSProviderLoader MBean

The JMSProviderLoader MBean service creates a JMS provider instance and binds it into JNDI. A JMS provider adaptor is a class that implements the org.jboss.jms.jndi.JMSProviderAdapter interface. It is used by the message driven bean container to access a JMS service provider in a provider independent manner.

The configurable attributes of the JMSProviderLoader service are:

- **ProviderName:** A unique name for the JMS provider. This will be used to bind the JMSProviderAdapter instance into JNDI under “java:/ProviderName”.
- **ProviderAdapterClass:** The fully qualified class name of the org.jboss.jms.jndi.JMSProviderAdapter interface to create an instance of. To use an alternate JMS provider like Fiorano, one would create an implementation of the JMSProviderAdaptor interface that allows the administration of the InitialContext

provider url, and the locations of the QueueConnectionFactory and TopicConnectionFactory in JNDI.

- **ProviderURL:** The JNDI Context.PROVIDER_URL value to use when creating the JMS provider root InitialContext.
- **QueueFactoryRef:** The JNDI name under which the provider javax.jms.QueueConnectionFactory will be bound.
- **TopicFactoryRef:** The JNDI name under which the javax.jms.TopicConnectionFactory will be bound.

org.jboss.jms.asf.ServerSessionPoolLoader MBean

The ServerSessionPoolLoader MBean service manages a factory for javax.jms.ServerSessionPool objects used by the message driven bean container.

The configurable attributes of the ServerSessionPoolLoader service are:

- **PoolName:** A unique name for the session pool. This will be used to bind the ServerSessionPoolFactory instance into JNDI under “java:/PoolName”.
- **PoolFactoryClass:** The fully qualified class name of the org.jboss.jms.asf.ServerSessionPoolFactory interface to create an instance of.

Summary

In this chapter you were introduced to the J2EE asynchronous messaging API, JMS. You were shown the key JMS interfaces and the two messaging paradigms, PTP and Pub/Sub. Usage of the interfaces was demonstrated through simple examples. The integration of JMS into the J2EE component model via message driven beans was discussed and a simple MDB example was presented.

You were then introduced to the JBossMQ component architecture that provides the JMS implementation. The flexible message transport mechanism of the architecture was highlighted. The final topic covered was the JBossMQ MBeans and their configurations.

The next JBoss component you will be introduced to is the EJB 1.1 container managed persistence engine, JBossCMP.

5. JBossCMP – The JBoss Container Managed Persistence Layer

The JBossCMP layer consists of the components that support the EJB 1.1 container managed persistence (CMP) model

The JBossCMP layer consists of the components that support the EJB 1.1 container managed persistence (CMP) model. In this chapter we will touch on what CMP is all about and how the JBoss EJB container provides extensible support for CMP. We will conclude with an introduction to the default CMP implementation provided with JBoss and how you can customize it for use with your CMP beans.

Container Managed Persistence – CMP

Container managed persistence (CMP) is an entity bean model that moves the task of reading and writing an entity bean's persistent state from the bean developer to the EJB container. An entity bean with container-managed persistence relies on the container to perform data access on behalf of the entity bean. CMP entity beans are simpler to develop than bean managed persistence (BMP) beans because the bean author focuses on the business logic aspect of the entity bean, while the EJB container handles the persistence requirements. The bean provider must define which fields in the entity bean the container persistence engine should manage. That is the extent of the bean provider's role with regard to persistence. Most application servers provide support for allowing the bean developer to customize how the EJB container manages persistent fields. We will discuss the customization support available in the default JBoss CMP engine later in the section on JAWS.

Whether or not CMP is the right model for your entity beans depends primarily on the complexity of any existing relational schema you must use as the source of your entity beans persistent fields. The EJB 1.1 CMP model often cannot be used effectively for highly relational models that spread an object's persistent state over more than one table. In this case the only real option is to use the BMP entity bean model. You can achieve some of the benefits of the simplicity of the CMP model with BMP if you use a sophisticated object-to-relational like CocoBase (<http://www.thoughtinc.com>).

This chapter is not about how to decide between the CMP and BMP models. This chapter is about the JBoss CMP architecture.

The JBossCMP Architecture

In this section will we introduce the JBoss EJB container architecture that supports the CMP model. The JBoss CMP architecture does not depend on a relational JDBC based store. Persistence management is modeled in an abstract and extensible fashion that allows the use of any persistent store. The key classes that make up the persistence abstraction layer for entity beans are presented in Figure 5-1.

Figure 5-1, The JBoss CMP persistence abstraction layer classes.

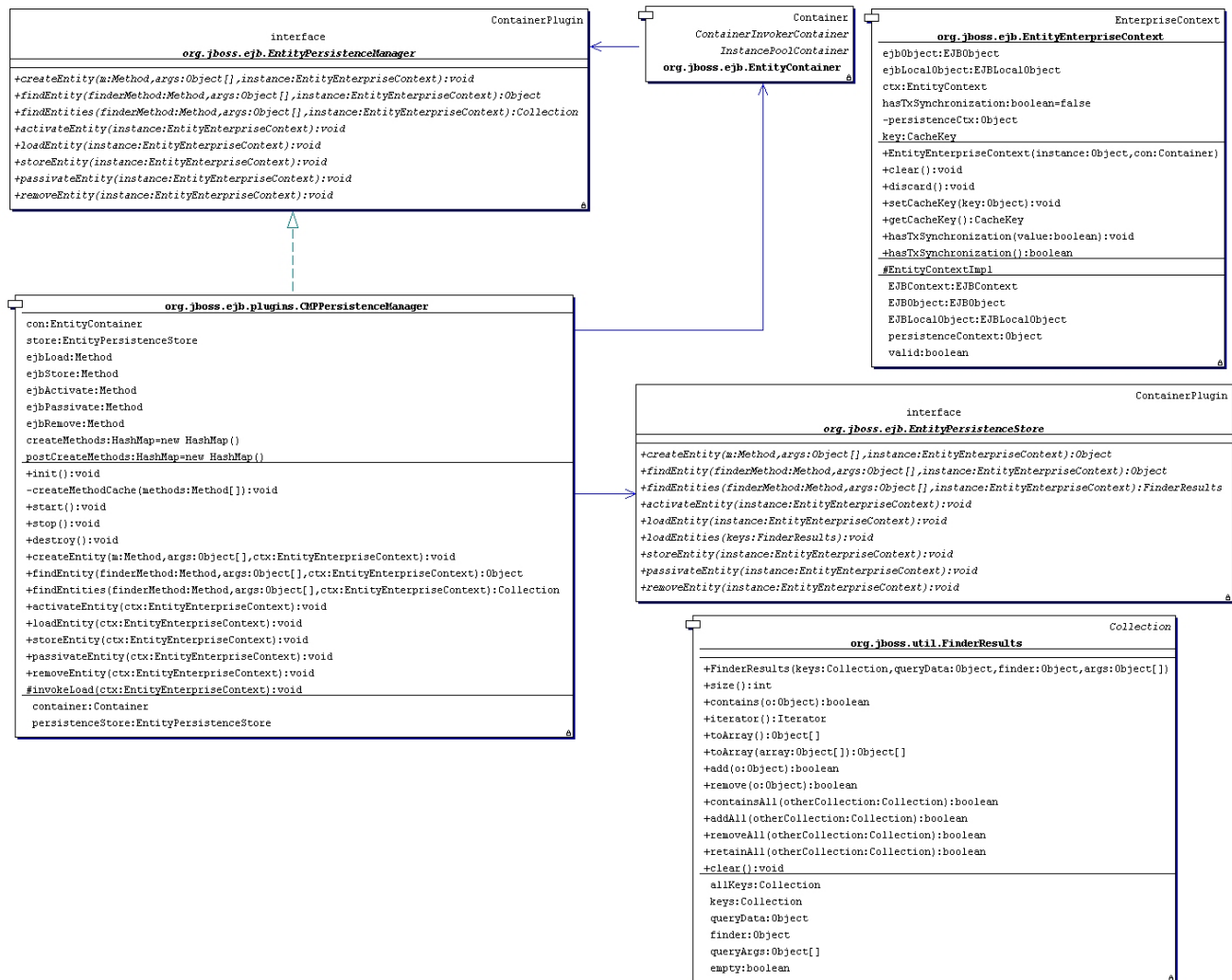


Figure 5-1 shows that the EntityContainer relies on an instance of the org.jboss.ejb.EntityPersistenceManager interface for persistence management. The EntityPersistenceManager abstraction is actually used for both CMP and BMP persistence, but only the CMP implementation is shown in Figure 5-1 as the org.jboss.ejb.plugins.CMPPersistenceManager class. The CMPPersistenceManager implements the semantics of the EJB 1.1 CMP callback model. The various EJB life-cycle callback methods (ejbLoad, ejbStore, ejbActivate, ejbPassivate, ejbRemove) are invoked by the CMPPersistenceManager in response to EntityPersistenceManager method invocations. The actual storage of the entity persistent fields is delegated to an EntityPersistenceStore implementation that takes care of the details of a particular physical store. For the CMP entity container it is the implementation of the EntityPersistenceStore interface that is externalized by the container configuration and may be replaced by an implementation of your choice. The default implementation of the EntityPersistenceStore interface defined in the standardjboss.xml configuration file is the org.jboss.ejb.plugins.jaws.JAWSPersistenceManager. We will talk about the specifics of JAWS in a later section.

Customization of the persistence storage of a CMP entity bean comes down to a providing a custom implementation of the EntityPersistenceStore interface. Let's take a look at the methods of this interface in more detail. The following list describes each method, its arguments, return values and exceptions.

- Object 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 handling the return value and any exception generated by the persistent store. The persistence manager first invokes the entity ejbCreate method, followed by the createEntity method, and then the entity ejbPostCreate method. The value the method must return is the primary key object for the newly created entity bean. The method arguments are:
 - m, the home interface create method that generated this createEntity call.
 - args, the arguments passed to the home interface create method.
 - instance, the context associated with the instance being used for this create call.
- Object findEntity(Method finderMethod, Object[] args, EntityEnterpriseContext instance) throws Exception: This method is called when a single entity is to be found. The persistence store must find out if the desired instance is available in the store, and if it is, return the primary key of the entity. If the entity cannot be found a javax.ejb.ObjectNotFoundException must be

thrown. For any other search related failure a [javax.ejb.FinderException](#) must be thrown. The method arguments are:

- `finderMethod`, the home interface find method that generated this `findEntity` call.
 - `args`, the arguments passed to the home interface find method.
 - `instance`, the context associated with the instance being used for this find call.
- **FinderResults findEntities(Method finderMethod, Object[] args, EntityEnterpriseContext instance) throws Exception**: This method is called when a collection of entities is to be found. The persistence store must find out if the desired instances are available in the store, and if they are, return the matching primary keys as an `org.jboss.util.FinderResults` instance. If the entity cannot be found an empty `FinderResults` should be returned. A null value must not be returned. If a search related error occurs a [javax.ejb.FinderException](#) must be thrown. The method arguments are:
- `finderMethod`, the home interface find method that generated this `findEntity` call.
 - `args`, the arguments passed to the home interface find method.
 - `instance`, the context associated with the instance to use for activation.
- **`void activateEntity(EntityEnterpriseContext instance) throws RemoteException`**: This method is called when an entity should be activated. This event corresponds to moving an entity instance from the pooled to the ready state and does not really have anything to do with the persistent storage of entity. However this call permits one to introduce optimizations in the persistence store. Particularly if the context has a "PersistenceContext" that a store can use (JAWS does for smart updates) and this is as good a callback as any other to set it up. The persistence manager invokes this method after [ejbActivate](#) has been invoked on the entity. On system errors a [RemoteException](#) should be thrown. The method arguments are:
- `instance`, the context associated with the instance being used for this find call.
- **`void loadEntity(EntityEnterpriseContext instance) throws RemoteException`**: This method is called whenever an entity needs to be loaded from the underlying storage. The persistence manager loads the state from the underlying storage by calling the [loadEntity](#) method and then calls [ejbLoad](#) on the supplied instance.

- instance, the context associated with the instance being used for this loadEntity call.
- void loadEntities(FinderResults keys) throws RemoteException: This method is called whenever a set of entities should be preloaded from the underlying storage. The persistence store is allowed to make this a null operation. On system errors a RemoteException should be thrown. The method arguments are:
 - keys, a collection of primary keys previously returned from findEntities call.
- void storeEntity(EntityEnterpriseContext instance) throws RemoteException: This method is called whenever an entity needs to be written to the underlying store. The persistence manager must call ejbStore on the supplied instance and then store the state by invoking this method. On system errors a RemoteException should be thrown. The method arguments are:
 - instance, the context associated with the instance to synchronize with the store.
- void passivateEntity(EntityEnterpriseContext instance) throws RemoteException: This method is called when an entity is to be passivated. The persistence manager must call the ejbPassivate method on the instance prior to calling this method. See the activateEntity method for the reason for exposing EJB pool callback calls to the store. On system errors a RemoteException should be thrown. The method arguments are:
 - instance, the context associated with the instance to passivate.
- void removeEntity(EntityEnterpriseContext instance) throws RemoteException, RemoveException: This method is called when an entity shall be removed from the underlying storage. The persistence manager must call ejbRemove on the instance prior to invoking this method. If the instance cannot be removed from the store a javax.ejb.RemoveException must be thrown. On system errors a RemoteException should be thrown. The method arguments are:
 - instance, the context associated with the instance to remove from the store.

Since EntityPersistenceStore extends the org.jboss.ejb.ContainerPlugin interface, the ContainerPlugin life-cycle methods must also be implemented. These methods are:

- void init(): called by the CMPPersistenceManager as an initialization request notification.

- **void start() :** called by the CMPPersistenceManager as an startup request notification.
- **void stop() :** called by the CMPPersistenceManager as an stop request notification.
- **void destroy() :** called by the CMPPersistenceManager as an final shutdown request notification.
- **void setContainer(Container con) :** called by the CMPPersistenceManager to set the EntityContainer instance that the store is associated with. The EntityContainer can be used to obtain the entity bean metadata and class information.

To help you understand the requirements for implementing a custom store we will create a simple file based implementation of the EntityPersistenceStore interface.

A Custom file based persistence manager

This section will illustrate the steps required to implement a custom store for CMP entity beans. We will create a simple file based implementation of the EntityPersistenceStore interface to demonstrate the basics. Listing 5-1 shows the code for the sample implementation.

Listing 5-1, the example filesystem based implementation of the EntityPersistenceStore interface.

```
package org.jboss.chap5.ex1;

import java.io.File;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.rmi.RemoteException;
import java.rmi.ServerException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import javax.ejb.EJBObject;
import javax.ejb.Handle;
import javax.ejb.EntityBean;
import javax.ejb.CreateException;
```

```

import javax.ejb.DuplicateKeyException;
import javax.ejb.FinderException;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;

import org.apache.log4j.Category;

import org.jboss.ejb.Container;
import org.jboss.ejb.EntityContainer;
import org.jboss.ejb.EntityPersistenceStore;
import org.jboss.ejb.EntityEnterpriseContext;
import org.jboss.metadata.EntityMetaData;
import org.jboss.util.FinderResults;

/** An example EntityPersistenceStore implementation that uses a
    filesystem as the persistent store. An entity bean's cmp fields
    are written/read to/from the filesystem using Java serialization.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class FileStore implements EntityPersistenceStore
{
    Category log;
    EntityContainer entityContainer;
    EntityMetaData metaData;
    File storeRootDir;
    Field pkField;
    Field[] cmpFields;

    /** Called to set the EntityContainer we operate on behalf of.
    */
    public void setContainer(Container c)
    {
        entityContainer = (EntityContainer) c;
        metaData = (EntityMetaData) entityContainer.getBeanMetaData();
        log = Category.getInstance("FileStore#" + metaData.getEjbName());
        log.debug("setContainer, c="+c);
    }

    public void init() throws Exception
    {
        String ejbName = metaData.getEjbName();
        // Locate the jboss.home location or the current dir
        String dbPath = System.getProperty("jboss.home", ".");
        dbPath += "/db";
        log.debug("init, using dbPath="+dbPath);
        storeRootDir = new File(dbPath, ejbName);
        if( storeRootDir.exists() == false )
        {
            boolean created = storeRootDir.mkdirs();
            if( created == false )
                throw new IOException("Failed to create storeRootDir: "
                    +storeRootDir.getAbsolutePath());
        }
    }
}

```

```

    }
    String pkName = metaData.getPrimKeyField();
    Class beanClass = entityContainer.getBeanClass();
    pkField = beanClass.getField(pkName);
    // Get the public fields of the bean class
    ArrayList tmp = new ArrayList();
    Iterator cmpFieldsIter = metaData.getCmpFields();
    while( cmpFieldsIter.hasNext() )
    {
        String name = (String) cmpFieldsIter.next();
        Field f = beanClass.getField(name);
        tmp.add(f);
    }
    cmpFields = new Field[tmp.size()];
    tmp.toArray(cmpFields);
}

public void start()
{
}

public void stop()
{
}

public void destroy()
{
}

public Object createEntity(Method m, Object[] args,
    EntityEnterpriseContext ctx) throws Exception
{
    log.debug("createEntity: m="+m+", args="+args[0]);
    try
    {
        Object bean = ctx.getInstance();
        Object pk = pkField.get(bean);

        // Check exist
        File serFile = getFile(pk);
        if( serFile.exists() == true )
            throw new DuplicateKeyException("Already exists:"+pk);

        // Store to file
        storeEntity(pk, bean);
        return pk;
    }
    catch(Exception e)
    {
        throw new CreateException("Could not create entity:"+e);
    }
}

public Object findEntity(Method m, Object[] args,

```

```

EntityEnterpriseContext ctx)
throws RemoteException, FinderException
{
    log.debug("findEntity: m="+m+", args="+args[0]);
    if( m.getName().equals("findByPrimaryKey") )
    {
        Object pk = args[0];
        File serFile = getFile(pk);
        if( serFile.exists() == false )
            throw new ObjectNotFoundException(pk+" does not exist");

        return pk;
    }
    String msg = "Can't handle finderMethod: "+m;
    throw new FinderException(msg);
}

```

```

public FinderResults findEntities(Method m, Object[] args,
EntityEnterpriseContext ctx)
throws Exception
{
    log.debug("findEntities: m="+m);
    if( m.getName().equals("findAll") )
    {
        String[] files = storeRootDir.list();
        ArrayList result = new ArrayList();
        for(int i = 0; i < files.length; i++)
        {
            if (files[i].endsWith(".ser"))
            {
                String pk = files[i];
                pk = pk.substring(0,pk.length()-4);
                log.debug("found pk="+pk);
                result.add(pk);
            }
        }
        return new FinderResults(result, null, null, null);
    }
    String msg = "Can't handle finderMethod: "+m;
    throw new FinderException(msg);
}

```

```

public void loadEntity(EntityEnterpriseContext ctx)
throws RemoteException
{
    try
    {
        // Read fields from serialized object file
        String pk = ""+ctx.getId();
        log.debug("loadEntity, pk="+pk);
        File serFile = getFile(pk);
        FileInputStream fis = new FileInputStream(serFile);
        ObjectInputStream in = new CMPObjectInputStream(fis);
    }
}

```

```

        Object obj = ctx.getInstance();
        for(int i = 0; i < cmpFields.length; i++)
        {
            Field f = cmpFields[i];
            f.set(obj, in.readObject());
        }
        in.close();
    }
    catch(Exception e)
    {
        throw new ServerException("Load failed", e);
    }
}

public void storeEntity(EntityEnterpriseContext ctx)
    throws RemoteException
{
    storeEntity(ctx.getId(), ctx.getInstance());
}

public void removeEntity(EntityEnterpriseContext ctx)
    throws RemoteException, RemoveException
{
    Object pk = ""+ctx.getId();
    log.debug("removeEntity: pk="+pk);
    File serFile = getFile(pk);
    if( serFile.delete() == false )
        throw new RemoveException("Could not remove file:"+pk);
}

public void loadEntities(FinderResults keys)
{
    // We have no preload phase
}

public void activateEntity(EntityEnterpriseContext ctx)
    throws RemoteException
{
    // We have no passivation logic
}

public void passivateEntity(EntityEnterpriseContext ctx)
    throws RemoteException
{
    // We have no passivation logic
}

protected File getFile(Object id)
{
    String baseName = id.toString();
    return new File(storeRootDir, baseName+".ser");
}

private void storeEntity(Object id, Object obj)
    throws RemoteException
{

```

```

try
{
    // Write fields to serialized object file
    File serFile = getFile(id);
    log.debug("storeEntity, serFile="+serFile);
    FileOutputStream fos = new FileOutputStream(serFile);
    ObjectOutputStream out = new CMPObjectOutputStream(fos);

    for(int i = 0; i < cmpFields.length; i++)
    {
        Field f = cmpFields[i];
        out.writeObject(f.get(obj));
    }
    out.close();
}
catch (Exception e)
{
    throw new ServerException("Store failed", e);
}
}

static class CMPObjectOutputStream
    extends ObjectOutputStream
{
    public CMPObjectOutputStream(OutputStream out)
        throws IOException
    {
        super(out);
        enableReplaceObject(true);
    }

    protected Object replaceObject(Object obj)
        throws IOException
    {
        if (obj instanceof EJBObject)
            return ((EJBObject)obj).getHandle();

        return obj;
    }
}

static class CMPObjectInputStream
    extends ObjectInputStream
{
    public CMPObjectInputStream(InputStream in)
        throws IOException
    {
        super(in);
        enableResolveObject(true);
    }

    protected Object resolveObject(Object obj)
        throws IOException
    {

```

```

        if (obj instanceof Handle)
            return ((Handle)obj).getEJBObject();

        return obj;
    }
}

```

This implementation is a rather simple one designed to help you understand the nature of the EntityPersistenceStore interface requirements. The example has a number of fundamental limitations with respect to use in a production environment that we will highlight after discussing the implementation and demonstrating its use.

The first five methods of Listing 5.1 are from the org.jboss.ejb.ContainerPlugin interface. The ContainerPlugin interface is used to manage the life-cycle of plugins associated with an org.jboss.ejb.Container instance. They allow a plugin to know when it has been associated with a Container, and when the Container is initialized, started, stopped and destroyed. In the FileStore example we only make use of the setContainer and init methods. The setContainer callback informs the FileStore which Container instance it will be used by. The FileStore uses the setContainer callback to save the EntityContainer reference, obtain the entity bean metadata, and initialize a log4j Category instance with a category name of “FileStore#”+ the entity bean name the EntityContainer is managing. We are using the log4j API in this example to allow the debugging output from the FileStore to be annotated with a distinct category to isolate the FileStore statements from the rest of the server component logging.

The init method performs three tasks. The first is the location, and optionally, the creation of the directory into which persistent data for entity beans managed by the EntityContainer will be stored. The directory is the “db/ejbName” subdirectory under the jboss distribution root directory. The second task the init method performs is initialization of a java.lang.reflect.Field instance for the instance variable that serves as the entity bean primary key. The last step is to obtain Field instances for the entity bean instance variables that have been declared as container managed through cmp-field declarations in the ejb-jar.xml descriptor. This information is available from the EntityBeanMetaData associated with the EntityContainer. The Field instances will be used to read/write values from/to the entity bean instances. Note that we could not have performed the last two steps inside of the setContainer method, as the Class object for the entity bean has not been loaded at the time setContainer is called.

The createEntity method is responsible for creating a new persistent record. It is invoked whenever a client calls a create method on the bean’s home interface. The persistent record is created by obtaining the entity bean instance that has been created from the EntityEnterpriseContext using the getInstance method. The primary key for the bean is obtained using reflection via the pkField instance. A File object is created based on the

primary key and a check is made that the file does not currently exist. Each entity bean is stored in a file with a name equal to the bean's primary key as a string + the ".ser" extension. A clash between file names corresponds to duplicate entity bean primary keys. If a file already exists with the calculated name, a DuplicateKeyException is thrown. If no such file exists, the entity's initial state is saved to disk by calling the storeEntity method and the bean's primary key value is returned. Any Exception that occurs during the save is caught and re-thrown as a CreateException.

The findEntity method is invoked whenever a client calls a single-entity finder method on the bean's home interface. The StoreFile implementation of findEntity only handles the required findByPrimaryKey finder, and this is why the finder method name is checked against "findByPrimaryKey". Location of an entity simply consists of constructing a file from the primary key argument and testing for existence of the file. If the file exists the input primary key is returned to indicate an entity was found. If the file does not exist an ObjectNotFoundException is thrown. An attempt to perform any other single-entity finder results in a FinderException being thrown.

The findEntities method is invoked whenever a client calls a multi-entity finder method on the bean's home interface. FileStore implements only the findAll version of the multi-entity finder as this is the easiest version to support. Returning all existing primary keys simply consists of listing all files in the store directory and removing the ".ser" suffix from the file name. An attempt to perform any other multi-entity finder results in a FinderException being thrown.

The loadEntity method is called whenever the container deems it necessary to populate an entity bean instance with the persistent store state. The entity bean's state is loaded by reading its serialized state from its store file using Java serialization and reflection. The entity instance is obtained from the EntityEnterpriseContext using the getInstance method.

The storeEntity method is called whenever the container deems it necessary to save the state of an entity bean instance to the persistent store. This method obtains the entity bean instance and primary key from the EntityEnterpriseContext and then calls the internal storeEntity method to perform the state storage. The internal storeEntity method writes the entity bean's state to its store file using Java serialization and reflection.

The removeEntity method is called whenever a client calls remove on the bean's remote or home interface. Removal of an entity simply entails removal of its persistent store file. A RemoveException is thrown if the file cannot be deleted for any reason.

The remaining EntityPersistenceStore interface methods, loadEntities, activateEntity and passivateEntity are empty methods in the FileStore example, as they serve no purpose in its implementation.

A final item of note in the `FileStore` example is the use of custom `ObjectInputStream` and `ObjectOutputStream` subclasses. Recall from your EJB specification reading that CMP fields are restricted to the following: Java primitive types, `Serializable` types, and references of enterprise beans' remote or home interfaces. The serialized form of remote interfaces does not maintain its association with the EJB instance in the JBoss server. A `javax.ejb.Handle` does. So, the custom `CMPObjectOutputStream` and `CMPObjectInputStream` classes replace `EJBObjects` with `Handles` on output and vice-versa on input.

Using the FileStore

Ok, so we have created a custom persistent store, but how do we make use of it? The answer is that we need to modify the default CMP container configuration to use our `FileStore` class. This requires knowledge about the container configuration section of the `jboss.xml` descriptor, the details of which are described in the advanced JBoss configuration chapter. Instead of taking a detour and talking in detail about the container configuration options available in the `jboss.xml` descriptor, we'll just show you how to use the `FileStore` by example. We will cover the details of the `jboss.xml` elements we introduce in the example when we get to Chapter 9, "Advanced JBoss configuration using `jboss.xml`".

To demonstrate the use of the custom `FileStore` we need a CMP entity bean. Listing 5-2 gives the home, remote interfaces as well as the bean class for the trivial entity bean we will use as our test case. The corresponding `ejb-jar.xml` and `jboss.xml` descriptors for the entity bean jar are given in Listing 5-3.

Listing 5-2, The FileStore usage example entity bean home, remote interfaces and bean class.

```
/** The FileStore example entity remote interface */
package org.jboss.chap5.ex1;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Ex1Entity extends EJBObject
{
    String getState() throws RemoteException;
    public void setIntVar(int i) throws RemoteException;
    public void setFloatVar(float f) throws RemoteException;
    public void setDoubleVar(double d) throws RemoteException;
}

/** The FileStore example entity home interface */
package org.jboss.chap5.example1;

import java.rmi.RemoteException;
import java.util.Collection;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;
```

```

public interface ExlEntityHome extends EJBHome
{
    public ExlEntity create(String key, int i, float f, double d)
        throws RemoteException, CreateException;

    public ExlEntity findByPrimaryKey(String name)
        throws RemoteException, FinderException;
    public Collection findAll()
        throws RemoteException, FinderException;
}

/** The FileStore example entity bean implementation */
package org.jboss.chap5.example1;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

import org.apache.log4j.Category;

public class ExlEntityBean implements EntityBean
{
    private static final Category log =
        Category.getInstance(ExlEntityBean.class);
    private EntityContext entityContext;

    public String key;
    public int intVar;
    public float floatVar;
    public double doubleVar;

    public String ejbCreate(String key, int i, float f, double d)
    {
        this.key = key;
        this.intVar = i;
        this.floatVar = f;
        this.doubleVar = d;
        log.debug("ejbCreate: key="+key+", i="+i+", f="+f+", d="+d);
        return null;
    }

    public void ejbPostCreate(String key, int i, float f, double d)
    {
        log.debug("ejbPostCreate: key="+key);
    }

    public void ejbLoad()
    {
        log.debug("ejbLoad:, key="+key);
    }
}

```

```

public void ejbRemove()
{
    log.debug("ejbRemove:, key="+key);
}

public void ejbStore()
{
    log.debug("ejbStore:, key="+key);
}

public void setEntityContext(EntityContext context)
{
    entityContext = context;
    log.debug("setEntityContext:, key="+key);
}

public void unsetEntityContext()
{
    entityContext = null;
    log.debug("unsetEntityContext:, key="+key);
}

public void ejbActivate()
{
}

public void ejbPassivate()
{
}

public String getState()
{
    log.debug("getState:, key="+key);
    StringBuffer sb = new StringBuffer("ExlEntityBean{");
    sb.append("key=");
    sb.append(key);
    sb.append(";i=");
    sb.append(intVar);
    sb.append(";f=");
    sb.append(floatVar);
    sb.append(";d=");
    sb.append(doubleVar);
    sb.append("}");
    return sb.toString();
}

public void setIntVar(int i)
{
    this.intVar = i;
}

public void setFloatVar(float f)
{
    this.floatVar = f;
}

public void setDoubleVar(double d)
{

```

```

        this.doubleVar = d;
    }
}

```

Listing 5-3, The ejb-jar.xml and jboss.xml descriptors for the FileStore example entity bean jar.

```

<!-- 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>
    <entity>
      <ejb-name>Ex1EntityBean</ejb-name>
      <home>org.jboss.chap5.example1.Ex1EntityHome</home>
      <remote>org.jboss.chap5.example1.Ex1Entity</remote>
      <ejb-class>org.jboss.chap5.example1.Ex1EntityBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field><field-name>key</field-name></cmp-field>
      <cmp-field><field-name>intVar</field-name></cmp-field>
      <cmp-field><field-name>floatVar</field-name></cmp-field>
      <cmp-field><field-name>doubleVar</field-name></cmp-field>
      <primkey-field>key</primkey-field>
    </entity>
  </enterprise-beans>
</ejb-jar>

<!-- The jboss.xml descriptor -->
<?xml version="1.0"?>
<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Standard CMP EntityBean</container-name>
      <persistence-manager>org.jboss.chap5.example1.FileStore</persistence-manager>
    </container-configuration>
  </container-configurations>
</jboss>

```

The ejb-jar.xml descriptor defines four CMP fields: key, intVar, floatVar and doubleVar with key being the primary key field of type java.lang.String. The jboss.xml descriptor overrides the “Standard CMP EntityBean” container-configuration element to use the example org.jboss.chap5.example1.FileStore class as the persistence-manager value. Accept this as magic for now and we will explain all when we get to the advanced JBoss configuration chapter.

The last item needed to complete the FileStore testing is a client application that accesses the CMP entity bean. Its code is presented in Listing 5-4.

Listing 5-4, The FileStore testcase client application.

```
package org.jboss.chap5.ex1;

import java.util.Collection;
import java.util.Iterator;
import javax.ejb.ObjectNotFoundException;
import javax.naming.InitialContext;

public class Ex1Client
{
    static final int N = 4;

    public static void main(String args[]) throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("Ex1EntityBean");
        Ex1EntityHome home = (Ex1EntityHome) ref;
        Ex1Entity bean = home.create("Bean0", 1, 1.0f, 1.0);
        System.out.println("bean = "+bean.getState());
        bean = home.findByPrimaryKey("Bean0");
        System.out.println("findByPrimaryKey(Beano): "+bean);
        bean.remove();

        try
        {
            bean = home.findByPrimaryKey("Bean0");
            String msg = "findByPrimaryKey did NOT fail as expected";
            throw new IllegalStateException(msg);
        }
        catch(ObjectNotFoundException e)
        {
            System.out.println("findByPrimaryKey failed as expected");
        }

        System.out.println("Creating Bean1..Bean"+N);
        for(int i = 1; i <= N; i++)
        {
            bean = home.create("Bean"+i, 1+i, 1.0f+i, 1.0+i);
        }
        System.out.println("Finding all beans");
        Collection allBeans = home.findAll();
        Iterator iter = allBeans.iterator();
        while( iter.hasNext() )
        {
            bean = (Ex1Entity) iter.next();
            System.out.println("    "+bean.getState());
        }

        System.out.println("Removing Bean1..Bean"+N);
    }
}
```

```

        for(int i = 1; i <= N; i ++){
            {
                home.remove("Bean"+i);
            }
            System.out.println("Done");
        }
    }
}

```

Ok, now we are ready to test the FileStore implementation. To do this, make sure you have the JBoss server running and execute the chapter 6, example 1 test code by using Ant as follows from your examples root directory:

```

examples 854>ant -Dchap=5 -Dex=1 run-example
Buildfile: build.xml

...

chap5-ex1-jar:

run-example1:
    [copy] Copying 1 file to G:\JBoss-2.4.3\deploy
    [echo] Waiting for deploy...
    [java] bean = Ex1EntityBean{key=Bean0;i=1;f=1.0;d=1.0}
    [java] findByPrimaryKey(Beano): Ex1EntityBean:Beano
    [java] findByPrimaryKey failed as expected
    [java] Creating Bean1..Bean4
    [java] Finding all beans
    [java]   Ex1EntityBean{key=Bean1;i=2;f=2.0;d=2.0}
    [java]   Ex1EntityBean{key=Bean2;i=3;f=3.0;d=3.0}
    [java]   Ex1EntityBean{key=Bean3;i=4;f=4.0;d=4.0}
    [java]   Ex1EntityBean{key=Bean4;i=5;f=5.0;d=5.0}
    [java] Removing Bean1..Bean4
    [java] Done

BUILD SUCCESSFUL

Total time: 8 seconds

```

For this example you will want to also look at the server.log to see the output generated by the FileStore and the Ex1EntityBean. The tail of the log just after running the client should resemble the following. Note that many of the lines have been truncated to fit the output into the book format so see your server.log file for the full output.

```

[Ex1EntityBean] setEntityContext:, key=null
[Ex1EntityBean] ejbCreate: key=Bean0, i=1, f=1.0, d=1.0
[FileStore#Ex1EntityBean] createEntity: m=Ex1Entity create(...)
[FileStore#Ex1EntityBean] storeEntity, serFile=.../Bean0.ser
[Ex1EntityBean] ejbPostCreate: key=Bean0, i=1, f=1.0, d=1.0
[Ex1EntityBean] ejbStore:, key=Bean0
[FileStore#Ex1EntityBean] storeEntity, serFile=.../Bean0.ser
[Ex1EntityBean] getState:, key=Bean0

```

```

[ExlEntityBean] ejbStore:, key=Bean0
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean0.ser
[ExlEntityBean] setEntityContext:, key=null
[ExlEntityBean] ejbRemove:, key=Bean0
[FileStore#ExlEntityBean] removeEntity: pk=Bean0
[ExlEntityBean] setEntityContext:, key=null
[FileStore#ExlEntityBean] findEntity: m=ExlEntity findByPrimaryKey(...)
[ExlEntityBean] setEntityContext:, key=null
[ExlEntityBean] ejbCreate: key=Bean1, i=2, f=2.0, d=2.0
[FileStore#ExlEntityBean] createEntity: m=ExlEntity create(...)
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean1.ser
[ExlEntityBean] ejbPostCreate: key=Bean1, i=2, f=2.0, d=2.0
[ExlEntityBean] ejbStore:, key=Bean1
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean1.ser
[ExlEntityBean] setEntityContext:, key=null
[ExlEntityBean] ejbCreate: key=Bean2, i=3, f=3.0, d=3.0
[FileStore#ExlEntityBean] createEntity: m=ExlEntity create(...)
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean2.ser
[ExlEntityBean] ejbPostCreate: key=Bean2, i=3, f=3.0, d=3.0
[ExlEntityBean] ejbStore:, key=Bean2
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean2.ser
[ExlEntityBean] setEntityContext:, key=null
[ExlEntityBean] ejbCreate: key=Bean3, i=4, f=4.0, d=4.0
[FileStore#ExlEntityBean] createEntity: m=ExlEntity create(...)
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean3.ser
[ExlEntityBean] ejbPostCreate: key=Bean3, i=4, f=4.0, d=4.0
[ExlEntityBean] ejbStore:, key=Bean3
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean3.ser
[ExlEntityBean] setEntityContext:, key=null
[ExlEntityBean] ejbCreate: key=Bean4, i=5, f=5.0, d=5.0
[FileStore#ExlEntityBean] createEntity: m=ExlEntity create(...)
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean4.ser
[ExlEntityBean] ejbPostCreate: key=Bean4, i=5, f=5.0, d=5.0
[ExlEntityBean] ejbStore:, key=Bean4
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean4.ser
[ExlEntityBean] setEntityContext:, key=null
[FileStore#ExlEntityBean] findEntities: m=Collection findAll()
[FileStore#ExlEntityBean] found pk=Bean1
[FileStore#ExlEntityBean] found pk=Bean2
[FileStore#ExlEntityBean] found pk=Bean3
[FileStore#ExlEntityBean] found pk=Bean4
[ExlEntityBean] getState:, key=Bean1
[ExlEntityBean] ejbStore:, key=Bean1
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean1.ser
[ExlEntityBean] getState:, key=Bean2
[ExlEntityBean] ejbStore:, key=Bean2
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean2.ser
[ExlEntityBean] getState:, key=Bean3
[ExlEntityBean] ejbStore:, key=Bean3
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean3.ser
[ExlEntityBean] getState:, key=Bean4
[ExlEntityBean] ejbStore:, key=Bean4
[FileStore#ExlEntityBean] storeEntity, serFile=.../Bean4.ser
[ExlEntityBean] ejbRemove:, key=Bean1

```

```
[FileStore#Ex1EntityBean] removeEntity: pk=Bean1
[Ex1EntityBean] ejbRemove:, key=Bean2
[FileStore#Ex1EntityBean] removeEntity: pk=Bean2
[Ex1EntityBean] ejbRemove:, key=Bean3
[FileStore#Ex1EntityBean] removeEntity: pk=Bean3
[Ex1EntityBean] ejbRemove:, key=Bean4
[FileStore#Ex1EntityBean] removeEntity: pk=Bean4
```

The Ex1Client exercises bean creation, removal, and finders. The server.log output illustrates the interaction between the EJB container and its calls to the Ex1EntityBean and the FileStore persistence manager. Since the Ex1Client ultimately removes all the entity beans it creates, the only proof that the FileStore was operating will be the existence of a db/Ex1EntityBean directory under your JBoss server distribution root directory. Experiment with modifying the Ex1Client to have it leave the beans it creates, stop and restart the server and then find all existing beans to verify that the FileStore is functioning as a persistent store.

Limitations of the FileStore

The FileStore example is a simple pedagogical example designed to illustrate the basic functionality of an EntityPersistenceStore implementation. The FileStore class has a number of limitations that would need to be addressed for a robust implementation. The most important limitations are with respect to improper transaction semantics. Some of the major limitations of the example include:

- Reads and writes are not atomic. If a write fails midway through the entity beans list of persistent fields, the image on disk is an inconsistent mix of the previous values with partially updated values. The same problem exists during the read of the persistent data into the entity bean. The reads and writes need to be made atomic operations.
- Reads and writes are not rolled back on rollback of the current transaction. If a read or write operation succeeds, but the transaction in which the operation is enlisted fails due to some other transacted resource, the FileStore does not undo the last operation.
- Only simple primary keys are supported. Support for compound primary keys made of more than one CMP field should also be added.
- The findEntities implementation of findAll assumes that the primary key type is java.lang.String. This is because the primary keys it returns are derived from the names of the persistent store files. One way to allow for a primary key of any type would be to introduce a “.key” file that contained the serialized content of the primary key.

- The only supported finder methods are findByPrimaryKey and findAll. Typically a user wants to be able to locate entities using queries other than identity against a primary key.

It is left as an exercise for the reader to address these issues.

JAWS – The default CMP implementation

JAWS (Just Another Web Storage) is the default object-to-relational (O-R) mapping tool provided with JBoss for CMP entity beans. JAWS plugs into the JBoss container as the EntityPersistenceStore implementation in the default CMP container configuration defined in `standarjboss.xml`. JAWS is essentially a much more sophisticated version of the FileStore example we just looked at, which uses a JDBC database rather than a filesystem as its persistent store. The topic of the remainder of this chapter is the customization of JAWS for your particular entity beans and database. Before jumping into that topic however, let's take a brief detour to discuss what O-R mapping is about.

What is O-R mapping?

O-R mapping technology grew out of the differences between how object-oriented languages represent objects in memory and how relational databases store data on disk. Objects in the Java language might contain only primitive data types such as `int`, `double`, and very simple aggregate objects such as `String`, making it very easy to express the object's layout on disk. In the case of storing such a simple object in a file, you could just write each primitive data type variable and each `String` sequentially into the file. Reading such an object back from disk into a memory-based object would be just as easy. However, what about storing more complex objects such as those that contain other objects that contain yet other objects? And what about storing both simple and complex objects into relational databases?

As the complexity of the objects being stored increases the intelligence of the O-R mapping tool must also increase. An O-R mapping tool must understand how to traverse the complex object's memory graph and figure out how to store it to and read it from disk. To add to the complexity, the graph of a single object might contain multiple objects that each references a single, unique object, as well as objects that recursively references itself. In these cases the O-R mapping tool would have to avoid persisting the same object multiple times, perhaps even ending up in an endless loop because of the self-referencing composition! On the other hand, all complex Java objects finally boil down to variables of primitive data types and other simple Objects. Therefore, while it can be quite challenging to persist very complex objects, it is not impossible. JAWS is the free Open Source offering that we include as the default CMP management tool to ensure that JBoss has a high quality O-R framework.

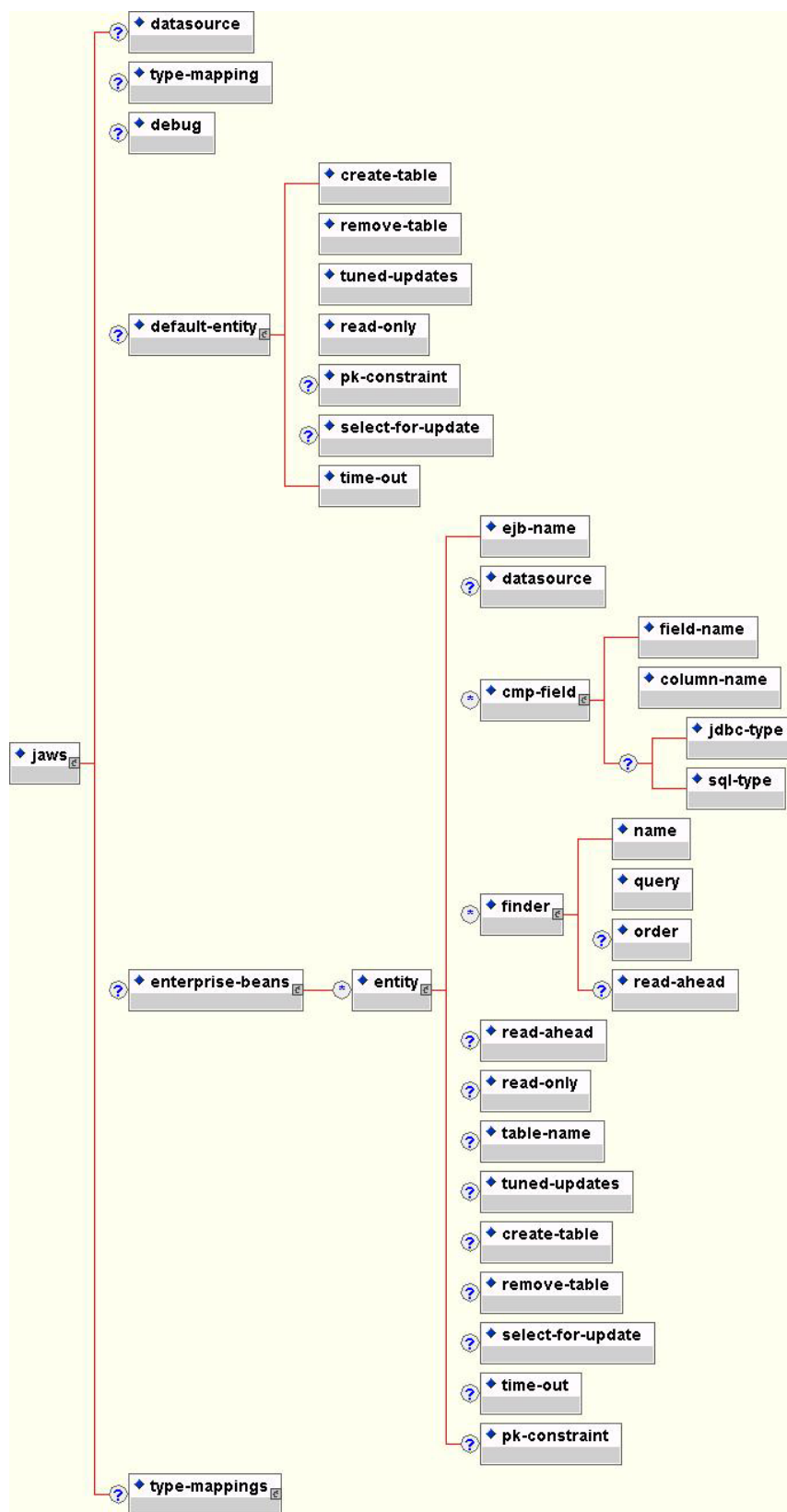
If you still need more capabilities, there are commercial O-R tools that can be used with JBoss. `CocoBase`, a professional high-end tool available from `THOUGHT inc` is one JBoss

Group partner solution. Another lightweight, low-cost solution is the MVCSoft Persistence Manager. See the www.jboss.org partner's section for additional information on these tools.

Customizing the behavior of JAWS

JAWS comes pre-configured to use the bundled HypersonicSQL embedded JDBC database and automatic table creation per CMP entity bean. This allows one to experiment with CMP entity beans with no JBoss specific configuration. However, this is only generally appropriate for relatively simple beans and testing. Fortunately a good deal of JAWS' behavior is customizable using two XML configuration files, `standardjaws.xml` and `jaws.xml`. Both files conform to the DTD defined by the `jaws_2_4.dtd`, which is represented graphically in Figure 5-2. Listing 5-5 shows a portion of the `standardjaws.xml` descriptor to give you a feel for the nature of the elements. Only the Hypersonic SQL type-mapping is shown in Listing 5-5.

Figure 5-2, The JAWS 2.4 XML configuration file DTD



Listing 5-5, the default standardjaws.xml descriptor with all but the Hypersonic SQL type-mapping elements removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaws>
  <datasource>java:/DefaultDS</datasource>
  <type-mapping>Hypersonic SQL</type-mapping>
  <debug>false</debug>

  <default-entity>
    <create-table>true</create-table>
    <remove-table>false</remove-table>
    <tuned-updates>false</tuned-updates>
    <read-only>false</read-only>
    <time-out>300</time-out>
    <select-for-update>false</select-for-update>
  </default-entity>

  <type-mappings>
    <type-mapping>
      <name>Hypersonic SQL</name>
      <mapping>
        <java-type>java.lang.Byte</java-type>
        <jdbc-type>SMALLINT</jdbc-type>
        <sql-type>SMALLINT</sql-type>
      </mapping>
      <mapping>
        <java-type>java.util.Date</java-type>
        <jdbc-type>DATE</jdbc-type>
        <sql-type>DATE</sql-type>
      </mapping>
      <mapping>
        <java-type>java.lang.Boolean</java-type>
        <jdbc-type>BIT</jdbc-type>
        <sql-type>BIT</sql-type>
      </mapping>
      <mapping>
        <java-type>java.lang.Integer</java-type>
        <jdbc-type>INTEGER</jdbc-type>
        <sql-type>INTEGER</sql-type>
      </mapping>
      <mapping>
        <java-type>java.lang.Object</java-type>
        <jdbc-type>JAVA_OBJECT</jdbc-type>
        <sql-type>OBJECT</sql-type>
      </mapping>
      <mapping>
        <java-type>java.lang.Short</java-type>
        <jdbc-type>SMALLINT</jdbc-type>
        <sql-type>SMALLINT</sql-type>
      </mapping>
      <mapping>
        <java-type>java.lang.Character</java-type>
```

```

        <jdbc-type>CHAR</jdbc-type>
        <sql-type>CHAR</sql-type>
    </mapping>
    <mapping>
        <java-type>java.lang.String</java-type>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(256)</sql-type>
    </mapping>
    <mapping>
        <java-type>java.sql.Timestamp</java-type>
        <jdbc-type>TIMESTAMP</jdbc-type>
        <sql-type>TIMESTAMP</sql-type>
    </mapping>
    <mapping>
        <java-type>java.lang.Float</java-type>
        <jdbc-type>REAL</jdbc-type>
        <sql-type>REAL</sql-type>
    </mapping>
    <mapping>
        <java-type>java.lang.Long</java-type>
        <jdbc-type>BIGINT</jdbc-type>
        <sql-type>BIGINT</sql-type>
    </mapping>
    <mapping>
        <java-type>java.lang.Double</java-type>
        <jdbc-type>DOUBLE</jdbc-type>
        <sql-type>DOUBLE</sql-type>
    </mapping>
</type-mapping>
</type-mappings>
</jaws>

```

The `standardjboss.xml` file is located in the JBoss server configuration file set directory. It serves as the default values specification. The `jaws.xml` file is a JBoss server specific file that you include in your EJB jars inside the META-INF directory along with the required `ejb-jar.xml` and optional `jboss.xml` descriptors. It allows customization of the default JAWS behavior in the same manner that the `jboss.xml` descriptor allows for customization of the `standardjboss.xml` definitions.

When you need to change the default behavior of JAWS, where you do it depends on whether you want the change to be seen globally or isolated to a particular EJB jar deployment. Changes that should be seen globally should be made in the `standardjboss.xml` descriptor, while EJB jar specific changes should be made in the `jar META-INF/jaws.xml` descriptor.

The immediate child elements of the `jaws` root element shown in Figure 5-2 define the range of configurable behavior available. The main configuration categories include:

- Specification of the javax.sql.DataSource location and the Java to SQL type mapping to use with the DataSource.
- Global options and defaults.
- Entity bean to database mapping and usage options.
- Customization of entity bean home interface finder methods.
- Java to SQL type mapping definitions.

We'll discuss each configuration category and the associated JAWS DTD elements in the following sections.

Specifying the DataSource and type mapping

The datasource element specifies the JNDI name of the javax.sql.DataSource binding for obtaining connections to your database. We'll talk about how DataSource bindings are configured later on in the configuring JDBC section. The default value of "java:/DefaultDS" does not need to be changed simply because you are not using the bundled HypersonicSQL database. You would change the datasource element value if you have multiple DataSource bindings defined, and the DataSource you wish to use does not correspond to the one you have configured as the "java:/DefaultDS" binding.

An aspect of JAWS that you are likely to want to change is the Java to SQL type mapping. If your not using the embedded HypersonicSQL database as your persistent store, you need to change the type-mapping to one that is compatible with your database. If there is not a predefined type-mappings/type-mapping element in the standardjaws.xml descriptor that matches your database, you can either try one that has equivalent or similar type mappings, or define a new one.

Java to SQL type mapping definitions

The type-mapping element defines which collection of mappings from a Java class type to JDBC and SQL types to use with the datasource. Its value is a reference to a name element in the type-mappings/type-mapping-definition section of the XML file. The mappings for a large number of databases have been defined and are available in the standardjaws.xml descriptor. If an existing type-mappings/type-mapping-definition does not exist for your database, you need to create an appropriate type mapping. A type-mapping-definition consists of the name of the definition and zero or more mapping elements that define the Java class type to JDBC and SQL types tuples. The values of the mapping element child elements are as follows:

- The java-type element specifies the fully qualified name of a Java class. This is the Java type of an entity bean cmp-field.
- The jdbc-type element specifies Java class name to JDBC type mapping. The value of the jdbc-type element is the string name of the java.sql.Types constant to which the Java class should map. This is used to determine what JDBC type to use when encoding an entity bean field value into a JDBC java.sql.PreparedStatement.
- The sql-type element specifies the database SQL declaration for the jdbc-type. This is used when JAWS creates a table for an entity bean.

Global options and defaults

There are a number of options that affect the default behavior of JAWS and are configured via the debug and default-entity elements. The debug element defines whether or not JAWS logs its SQL activity. When debug is true logging is enabled, and when false (the default), no logging of SQL activity is performed. Setting debug to true is useful when you need to see how JAWS is interacting with your database.

The various child elements of the default-entity element establish global defaults for beans with no specific entity element specification as well as missing elements of an entity declaration. We will go over their meanings when we discuss the entity element.

Entity bean to database mapping and usage options

The default behavior for JAWS is to use a database table per CMP entity bean with each CMP field of the entity bean mapped to a column in the table. The name of the table will be equal to the name of the EJB, and the names of the columns will be that of the CMP fields of the EJB. JAWS will even create the table for you if it does not exist.

This behavior may not be what you want due to pre-existing tables with different naming conventions. You can customize how JAWS maps an entity bean onto a table using the enterprise-beans/entity element specification. There are three classes of child elements in the entity element: bean to table mapping information, JAWS behavior options, and home interface finder specification. We'll first discuss the bean to table mapping configuration elements and follow that with the behavior elements and finder specifications.

The child elements of the entity element that allow you to control how an entity bean is mapped onto a JDBC table include the following:

- ejb-name: this element specifies the name of the EJB as declared in the ejb-jar.xml descriptor. It must match the value of an ejb-jar/enterprise-beans/entity/ejb-name

element from the standard ejb-jar.xml descriptor. In the absence of a table-name element, the value of the ejb-name element is used as the database table name.

- table-name: The name to use as the database table name.
- datasource: The JNDI name of the `javax.sql.DataSource` binding for obtaining connections to the database which contains the entity bean table.
- cmp-field: This element or elements allows one to override the database column name used for a CMP field. One can also optionally override the field type to JDBC and SQL type mappings. The child elements are:
 - field-name: The name of the CMP field as defined in the ejb-jar.xml descriptor. This must match an existing cmp-field/field-name in the ejb-jar.xml descriptor.
 - column-name: The name to be used as the database column name for the CMP field.
 - jdbc-type: The optional string name of the `java.sql.Types` constant to which the CMP field class should map.
 - sql-type: The optional sql-type element specifies the database SQL declaration for the jdbc-type. It will only be used if JAWS creates the entity bean table.

We delayed the discussion of the JAWS behavior options from the previous section to here as those elements apply to the default behavioral as well as per entity bean behavioral options. The option related elements and their meaning are:

- read-ahead: This element is a true/false flag indicating whether or not all data for the entities selected should be loaded immediately. Note that JAWS/JBoss cannot guarantee serializable transactions with the read-ahead set to true.
- read-only: This element is a true/false flag which when set to true indicates that changes to the bean's state should not be persisted.
- create-table: This element is a true/false flag which when set to true indicates that on deploy the entity bean table should be created if it does not exist.
- remove-table: This element is a true/false flag which when set to true indicates that on undeploy then entity bean table should be dropped from the database along with all data.

- **tuned-updates**: This element is a true/false flag which when set to true indicates that modifications of the entity bean should generate update SQL statements that only contain the changed CMP fields.
- **select-for-update**: This element is a true/false flag which when set to true indicates on loading an entity bean a 'SELECT ... FOR UPDATE' style of SQL statement should be used. This locks the table row and can be used to synchronize concurrent use as would be the case when multiple JBoss server instances use the database.
- **time-out**: This element specifies the interval in milliseconds between reloads of read-only entity beans. This element does not apply to any entity bean with a read-only setting of false.
- **pk-constraint**: This element is a true/false flag which when set to true, and create-table is true, indicates that the SQL table create statement must include a primary key constraint on the entity bean primary key columns.

As an example of a customized bean to table mapping for an existing table, consider this table creation statement:

```
create table Products (productID varchar(64), majorVersion int,
    minorVersion int, description text);
```

Suppose that we have a CMP entity bean named ProductBean that contains the following fields we wish to be mapped on the this table:

```
String key;
int major;
int minor;
String description;
```

We can specify the mapping from the ProductBean EJB to the existing table using the following jaws.xml descriptor:

```
<jaws>
  <enterprise-beans>
    <entity>
      <ejb-name>ProductBean</ejb-name>
      <table-name>Products</table-name>
      <create-table>false</create-table>
      <cmp-field>
        <field-name>key</field-name>
        <column-name>productID</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>major</field-name>
        <column-name>majorVersion</column-name>
      </cmp-field>
```

```

    <cmp-field>
      <field-name>minor</field-name>
      <column-name>minorVersion</column-name>
    </cmp-field>
    <cmp-field>
      <field-name>description</field-name>
      <column-name>description</column-name>
    </cmp-field>
  </entity>
</enterprise-beans>
</jaws>

```

Customization of entity bean home interface finder methods

JAWS handles the creation of finder methods defined in the entity bean home interface. The types of finder methods that JAWS can handle without any customization include:

- **findByPrimaryKey:** The required single-entity finder method that allows one to locate an entity bean by its primary key.
- **findAll:** A multi-entity finder that returns all existing entity beans. This finder method is generated only if declared in the entity bean home interface.
- **findBy<field>(<field-type> value):** JAWS can generate a multi-entity finder for finders that query one of the entity bean CMP fields. The <field> value must be the name of one of the entity bean CMP fields. The <field-type> value must be the Java type of the CMP field. Such a finder method is generated only if declared in the entity bean home interface.

If you need additional finder methods you must declare a custom finder using the jaws.xml descriptor. To do this you specify a finder child element in an enterprise-beans/entity element. The child elements of the finder element are:

- **name:** the name of the finder method as declared in the entity bean home interface.
- **query:** the SQL statement portion of the WHERE clause. Inside of this statement, arguments that are passed into the finder method can be referenced as {n} where n is the 0 based position of the argument in the finder method argument list.
- **order:** the optional SQL statement portion of the ORDER BY clause.
- **read-ahead:** an optional Boolean flag that indicates if all data for found entities should be loaded immediately.

Finder examples

Given a product entity bean named `ProductBean` with major and minor CMP fields of type int, which correspond to the product version information, suppose you need a finder method to locate all beta products (those with major version ≤ 1). A custom finder called `findBetaVersions` could be specified as follows:

```
<jaws>
  <enterprise-beans>
    <entity>
      <ejb-name>ProductBean</ejb-name>
      <finder>
        <name>findBetaVersions</name>
        <query>major &lt;= 0</query>
        <order>key</order>
      </finder>
    </entity>
  </enterprise-beans>
</jaws>
```

The SQL query that would be generated by JAWS for this find would look like:

```
SELECT ProductBean.key FROM ProductBean where major <= 0 ORDER BY key
```

Note that the \leq operator had to be specified using the `<` XML entity since a raw `'<'` would have been interpreted as the start of a new XML element. Next suppose that you also wanted a `findOlderProducts` finder that selected all products where the major version number was less than the version provided as an argument. The home interface method would look like:

```
Collection findOlderProducts(int version) throws RemoteException
```

and the corresponding custom finder would be coded as:

```
<jaws>
  <enterprise-beans>
    <entity>
      <ejb-name>ProductBean</ejb-name>
      <finder>
        <name>findOlderProducts</name>
        <query>major &lt; {0}</query>
      </finder>
    </entity>
  </enterprise-beans>
</jaws>
```

Configuring JDBC

This is the first chapter that we have really talked about databases, so we'll take this opportunity to go over the details of integrating a database using a JDBC

java.sql.DataSource into the JBoss server. We have already seen the two MBeans that allow one to load a JDBC driver and configure a DataSource when we overviewed the MBeans found in the default `jboss.jcml` file in chapter 2. The MBeans were the org.jboss.jdbc.JdbcProvider and org.jboss.jdbc.XADataSourceLoader. Here will discuss the configuration of the MySQL Open Source database to go through the steps required to setup a DataSource. We're using the MySQL database because it is not configured for use by default in the JBoss distribution. You can obtain the latest version of the MySQL database from its home page: <http://www.mysql.com/>. The JDBC driver for MySQL can be obtained from here: <http://mmmysql.sourceforge.net/>

The first step is to configure JBoss to load your database JDBC driver. In this example setup we will be using version 2.0.6 of the MySQL JDBC driver. The corresponding jar file is named `mm.mysql-2.0.6.jar`. You need to make the JDBC driver classes available to the JBoss server by adding the `mm.mysql-2.0.6.jar` to the server classpath. The easiest way to do this is to simply drop the jar into the JBoss server `lib/ext` directory. All jars located in this directory are added to the classpath used by the server on startup.

Once that is accomplished, we need to add the name of the JDBC java.sql.Driver interface implementation class to the JdbcProvider MBean configuration. The JdbcProvider MBean is a simple service that loads one or more JDBC drivers into the server VM. The JBoss server classpath must contain the appropriate jars for the JDBC drivers that are to be loaded. Typically you place the JDBC driver jar files into the `JBOSS_DIST/lib/ext` to satisfy this requirement. The configurable attributes for the JdbcProvider are:

- **Drivers:** A comma-separated list of the JDBC driver fully qualified class names that are to be loaded when the service starts. Each driver is loaded by calling Class.forName(name), where `name` is the fully qualified class name of the JDBC driver.

As an example, consider the MySQL JDBC driver. Its Driver class implementation is called org.gjt.mm.mysql.Driver. To add the MySQL driver you would include the driver class name in the JdbcProvider configuration by adding the class name to the `Drivers` attribute as shown here:

```
<mbean code="org.jboss.jdbc.JdbcProvider"
  name="DefaultDomain:service=JdbcProvider">
  <attribute name="Drivers">org.hsqldb.jdbcDriver,
    org.gjt.mm.mysql.Driver
  </attribute>
</mbean>
```

The `Drivers` attribute value is a comma-separated list of class names that implement the Driver interface, which should be loaded on startup. Adding the org.gjt.mm.mysql.Driver name to JdbcProvider configuration will cause the MySQL JDBC driver to be loaded into the

VM when the JBoss server starts. The driver will then be accessible to the XADataSourceLoader MBean. The next step is the configuration of the MySQL DataSource.

A javax.sql.DataSource object is a factory for java.sql.Connection objects. You obtain a DataSource instance by accessing its binding using a JNDI lookup. The XADataSourceLoader MBean manages connection pools and creates a DataSource binding for a JDBC driver. Connections obtained from the resulting DataSource are automatically registered with any transaction associated with the current thread and support two-phase commit. You configure the MBean with the properties appropriate for the JDBC driver you want to establish a DataSource for. The configurable attributes of the XADataSourceLoader MBean are given in the following list. The attributes that we will use for the MySQL setup are annotated with an additional paragraph.

- **PoolName:** The name to assign to the pool. This is used to bind the pool into JNDI under “java:/name”.

We will set this value to MySQLDS. This means that the MySQL DataSource will be available from within the server VM under the JNDI name “java:/MySQLDS”.

- **DataSourceClass:** The fully qualified class name of the javax.sql.XADataSource implementation. If your JDBC driver does not include support for XADataSource then use org.jboss.pool.jdbc.xa.wrapper.XADataSourceImpl and this class will proxy XADataSource calls to your driver.

We will use the org.jboss.pool.jdbc.xa.wrapper.XADataSourceImpl as the MySQL driver does not provide XADataSource support.

- **Properties:** Any properties required to connect to the data source. This should be expressed in a String of the form name1=value1;name2=value2;name3=value3...
- **URL:** The JDBC URL used to connect to the data source

We will connect to the test database on a local MySQL server. Therefore, our JDBC URL will be jdbc:mysql://localhost/test

- **JDBCUser:** The user name used to connect to the data source.

The MySQL default install requires no username when connecting from localhost so we leave this empty

- **Password:** The user password used to connect to the data source.

The MySQL default install requires no password when connecting from localhost so we leave this empty

- **MinSize:** The minimum size of the pool. The pool always starts with one instance, but if shrinking is enabled the pool will never fall below this size. It has no effect if shrinking is not enabled. The default is 0.
- **MaxSize:** The maximum size of the pool. Once the pool has grown to hold this number of instances, it will not add any more instances. If one of the pooled instances is available when a request comes in, it will be returned. If none of the pooled instances are available, the pool will either block until an instance is available, or return null (see the Blocking parameter). If you set this to zero, the pool size will be unlimited. The default is 0.

We will use a value of 5 for the example.

- **GCMinIdleTime:** If garbage collection is enabled, the amount of time (in milliseconds) that must pass before a connection in use is garbage collected - forcibly returned to the pool. The default is 1200000 (20 minutes).
- **GCEnabled:** Whether the pool should check for connections that have not been returned to the pool after a long period of time. This would catch things like a client that disconnects suddenly without closing database connections gracefully, or queries that take an unexpectedly long time to run. This is not generally useful in an EJB environment, though it may be for stateful session beans that keep a DB connection as part of their state. This is in contrast to the idle timeout, which closes connection that have been idle in the pool. The default is false.
- **GCInterval:** How often garbage collection and shrinking should run (in milliseconds), if they are enabled. The default is 120000 (2 minutes).
- **InvalidateOnError:** Sets the response for errors. If this flag is set and an error event occurs, the connection is removed from the pool entirely. Otherwise, the object is returned to the pool of available objects. For example, a SQL error may not indicate a bad database connection (flag not set), while a TCP/IP error probably indicates a bad network connection (flag set). The default is false.
- **TimestampUsed:** Sets whether object clients can update the last used time. If so, the last used time will be updated for significant actions (executing a query, navigating on a ResultSet, etc.). If not, the last used time will only be updated when the object is given to a client and returned to the pool. This time is important if shrinking or garbage collection are enabled (particularly the later). The default is false.

- **Blocking:** Controls the behavior of the pool when all the connections are in use. If set to true, then a client that requests a connection will wait until one is available. If set to false, then the pool will return null immediately (and the client may retry). Note: If you set blocking to false, your client must be prepared to handle null results! The default is true.
- **BlockingTimeout:** Sets how long to wait in milliseconds for a free connection when blocking, -1 indicates to wait forever. The default is -1.
- **IdleTimeout:** Set the idle timeout for unused connections. If a connection has been unused in the pool for this amount of time, it will be released the next time garbage collection and shrinking are run (see GCInterval). The default is 30 minutes.
- **IdleTimeoutEnabled:** Whether the pool should close idle connections. This prevents the pool from keeping a large number of connections open indefinitely after a spike in activity. Any connection that has been unused in the pool for longer than this amount of time will be closed. If you do not want the pool to shrink so rapidly, you can set the MaxIdleTimeoutPercent and then some connections will be recreated to replace the closed ones. This is in contrast to garbage collection, which returns connections to the pool that have been checked out of the pool but not returned for a long period of time. The default is false.
- **MaxIdleTimeoutPercent:** Sets the idle timeout percent as a fraction between 0 and 1. If a number of connections are determined to be idle, they will all be closed and removed from the pool. However, if the ratio of objects released to objects in the pool is greater than this fraction, some new objects will be created to replace the closed objects. This prevents the pool size from decreasing too rapidly. Set to 0 to decrease the pool size by a maximum of 1 object per test, or 1 to never replace objects that have exceeded the idle timeout. The pool will always replace enough closed connections to stay at the minimum size. The default is 1.
- **LoggingEnabled:** Whether the pool should record activity to the JBoss log. This includes events like connections being checked out and returned. It is generally only useful for troubleshooting purposes (to find a connection leak, etc.). The default is false.
- **TransactionIsolation:** Sets the Transaction isolation level on the SQL Connection. Valid values are TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE. The choice of the level allows one to choose a trade-off between performance and transaction isolation. The values listed are from poor to full isolation; but from good to

poor performance. Refer to `java.sql.Connection` for more information. The default is to use the JDBC driver default.

Creating a `XADataSourceLoader` MBean configuration with the indicated attribute settings gives us the following configuration to add to the `jboss.jcml` file:

```
<mbean code="org.jboss.jdbc.XADataSourceLoader"
  name="DefaultDomain:service=XADataSource,name=MySQLDS">
  <attribute name="PoolName">MySQLDS</attribute>
  <attribute name="DataSourceClass">
    org.jboss.pool.jdbc.xa.wrapper.XADataSourceImpl
  </attribute>
  <attribute name="URL">jdbc:mysql://localhost/test</attribute>
  <attribute name="JDBCUser"></attribute>
  <attribute name="Password"></attribute>
  <attribute name="MinSize">0</attribute>
  <attribute name="MaxSize">5</attribute>
</mbean>
```

That concludes the steps required to configure a database `DataSource`. The online documentation on the www.jboss.org website lists some configurations for common databases, so if you need a configuration for your database that is the best place to start.

The Default JDBC HypersonicDatabase

JBoss comes bundled with an embeddable Open Source database called Hypersonic SQL (<http://sourceforge.net/projects/hsqldb>). It is configured using the `org.jboss.jdbc.HypersonicDatabase` MBean service, which integrates the Hypersonic database into the JBoss server. On startup the `HypersonicDatabase` MBean initializes an in memory database.

The configurable attributes for the `HypersonicDatabase` service include the following:

- **Database:** The name of database. This translated into a file path under the `JBOSS_DIST/db/hypersonic` JBoss distribution directory. The default value is "default".
- **Port:** The listening port number to use for the Hypersonic database server connection.
- **Silent:** A Boolean flag indicating if the server should not display diagnostic messages. The default value is true indicating that no messages should be displayed.
- **Trace:** A Boolean flag indicating if the server should display JDBC trace messages. The default value is false indicating that no messages should be displayed.

Summary

6. JBossTX – The JBoss Transaction Manager

The JBossTX supports the integration of a Java Transaction API (JTA) transaction manager implementation.

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.

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.

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 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.

Interposing

Distributed transactions can span many nodes, and each of these nodes can have several recoverable resources. This means that during the XA two-phase commit protocol many slow network calls may have to be performed before the transaction is finally completed. To get around this, a technique known as interposing is used.

For example, consider a system consisting of server A and server B. The transaction is started on server A, and the transaction manager on server A is responsible for coordinating the transaction. On server B five recoverable resources are involved in the transaction. Without interposing, the transaction manager on server A would have to do 10 calls to the resources on server B during the XA two-phase commit protocol exchange.

However, if server B also has a local transaction manager running, it could interpose between the recoverable resources on server B and the transaction manager on server A. When the first recoverable resource on server B is accessed, it is not registered with the transaction manager on server A. Instead it is registered with the interposing transaction manager on server B, and the interposing transaction manager on server B registers itself as a resource with the transaction manager on server A. When the other recoverable resources on server B are accessed within the transaction, they are only registered with the interposing transaction manager on server B.

In such a setup, the transaction manager on server A is called the superior transaction coordinator, and the interposing transaction manager on server B is called the subordinate transaction coordinator. When it is time to commit the transaction, the transaction manager on server A only knows of a single resource on server B. During the prepare phase the superior transaction coordinator on server A does a single prepare call to the subordinate transaction on server B. When the subordinate transaction coordinator gets this call, it calls prepare on the five resources that were involved in the transaction, and the subordinate transaction coordinator gives an OK reply to the prepare call only if all five recoverable resources gave OK replies to the prepare call from the subordinate transaction coordinator.

In the second phase, the superior transaction coordinator on server A sends a single commit to the subordinate transaction coordinator on server B, and the subordinate transaction coordinator then calls commit on the five recoverable resources on server B.

A subordinate transaction coordinator can also be superior transaction coordinator to other subordinate transaction coordinators, thus enabling several levels of interposing. It is interesting to note that to a superior transaction coordinator a subordinate transaction coordinator looks just like any other recoverable resource.

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 `TransactionManager` 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 of 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.util.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.
- **XidClassName:** The class to use for 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.

The Tyrex Transaction Manager

Tyrex is a full-blown CORBA-based transaction manager that supports transaction context propagation between different server VMs. Anatoly Akkerman from New York University wrote a JBoss MBean service for this transaction manager that enables its use in JBoss without using the IIOP wire protocol. The Tyrex MBean class is included in the default bundle as org.jboss.tm.plugins.tyrex.TransactionManagerService.

The MBean has only one configurable attribute called `ConfigFileName`, which is the location of the Tyrex domain configuration file as either a URL string or a classpath resource. If no `ConfigFileName` value is specified it defaults to “domain.xml”. The Tyrex MBean is included in the default `jboss.jcml` configuration file just after the standard JBoss transaction manager MBean, but is commented out. To switch to using the Tyrex transaction manager you need simply to comment out or remove the default JBoss MBean and uncomment the Tyrex plugin. The Tyrex MBean configuration looks like the following:

```
<mbean code="org.jboss.tm.plugins.tyrex.TransactionManagerService"
  name="DefaultDomain:service=TransactionManager">
```

```
<attribute name="ConfigFileName">domain.xml</attribute>
</mbean>
```

To properly configure the Tyrex MBean service you need to setup the domain.xml configuration file correctly. See the tyrex.exolab.org website for the required configuration documentation.

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.

7. JBossCX – The JBoss Connector Architecture

JBossCX is an implementation of the J2EE Connector Architecture (JCA).

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 state of JCA in JBoss 2.4.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.

A graphical depiction of the JCA participants is given in

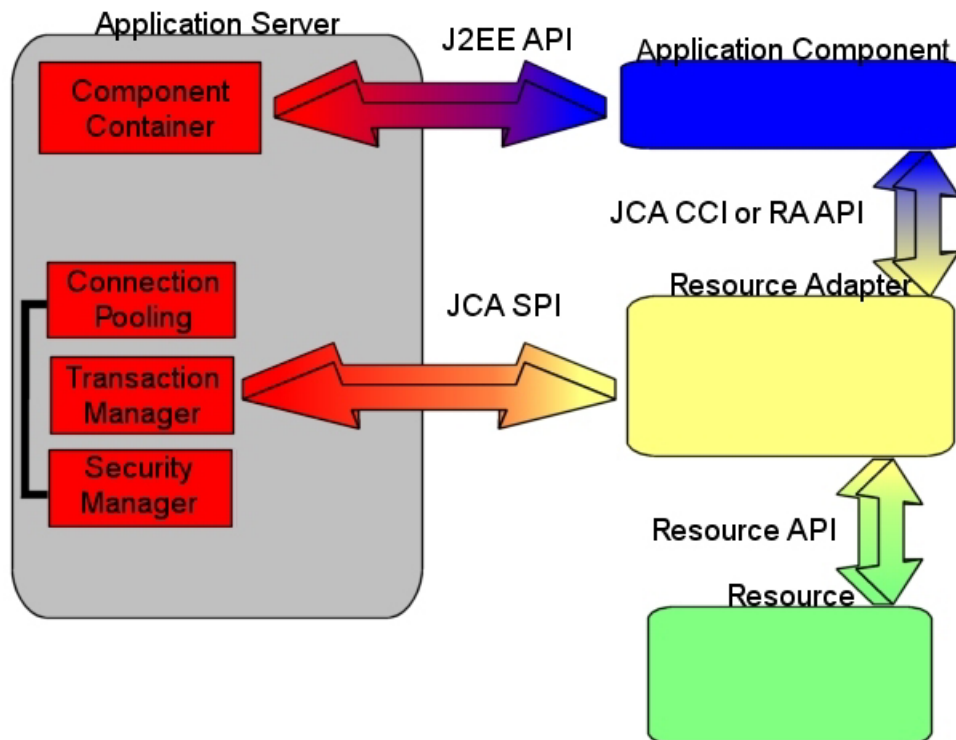


Figure 7-1.

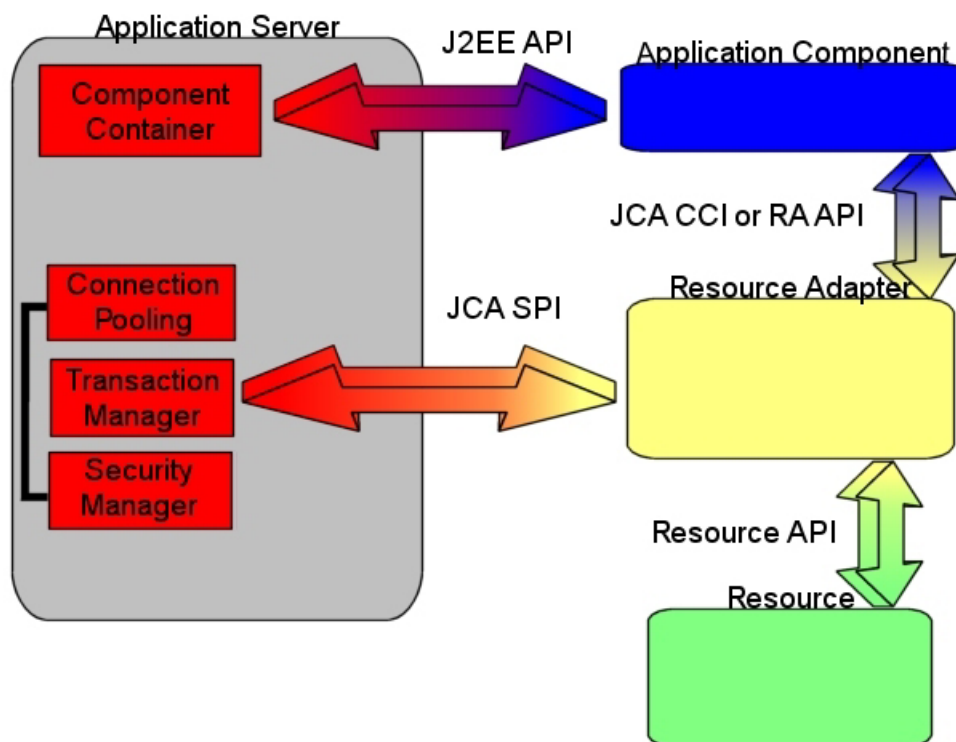


Figure 7-1, An overview of the interaction between the key participants defined by the JCA specification.

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 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 reproduced here as Figure 7-2.

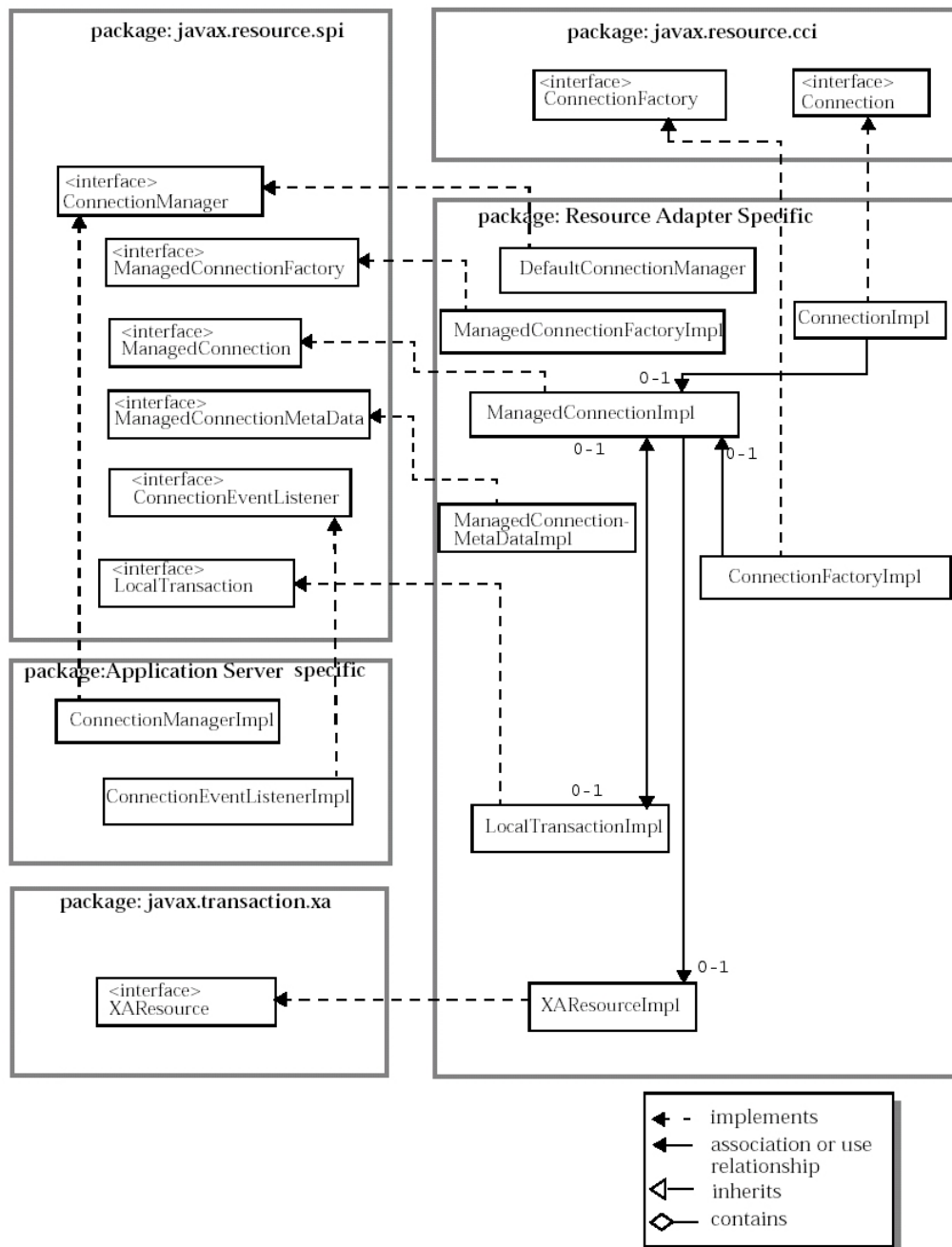


Figure 7-2, The JCA 1.0 specification figure 6.0 for the class diagram for the connection management architecture.

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`

The starting point for discussing the components in the JBoss server portion of Figure 7-3 are the two MBeans, org.jboss.resource.ConnectionManagerFactoryLoader and org.jboss.resource.ConnectionFactoryLoader. We'll start with the ConnectionManagerFactoryLoader MBean and finish with the ConnectionFactoryLoader MBean.

ConnectionManagerFactoryLoader MBean

The ConnectionManagerFactoryLoader is responsible for binding an org.jboss.resource.ConnectionManagerFactory instance into JNDI so that an org.jboss.resource.ConnectionFactoryLoader MBean can access it. A ConnectionManagerFactory implementation is responsible for creating org.jboss.resource.JBossConnectionManager instances for use by the JBoss server. The JBossConnectionManager interface is a subinterface of javax.resource.spi.ConnectionManager that adds support for re-enlisting connections under new transactions and shutting down the JBossConnectionManager. You define one or more ConnectionManagerFactoryLoader MBeans in the `jboss.jcml` file to define the types of ConnectionManager implementations that are available for use in the JBoss server. The configurable attributes of the ConnectionManagerFactoryLoader MBean are:

- **FactoryName:** The name of the connection manager factory. This is the name under which the connection manager factory will be bound in JNDI. Note that this name will be prefixed with "java:/" so that the binding will only be available inside of the JBoss server VM.
- **FactoryClass:** The fully qualified class name of the org.jboss.resource.JBossConnectionManager interface implementation to use as the connection manager factory.
- **Properties:** The properties to set on the connection manager factory instance. This is in the `java.util.Properties.load` format (one property per line in the form `name=value`).
- **TransactionManagerName:** The JNDI name of the javax.transaction.TransactionManager instance to use. If not specified this defaults to 'java:/TransactionManager'.

There are three implementations of the ConnectionManagerFactory interface bundled with the standard JBoss server distribution. They are:

- org.jboss.pool.connector.jboss.MinervaNoTransCMFactory. This is a factory for JBossConnectionManager instances suitable for use with resource adaptors that do not support transactions.

- org.jboss.pool.connector.jboss.MinervaSharedLocalCMFactory. This is a factory for JBossConnectionManager instances suitable for use with resource adaptors that support local transactions.
- org.jboss.pool.connector.jboss.MinervaXACMFactory. This is a factory for JBossConnectionManager instances suitable for use with resource adaptors that support JTA XA transactions.

The default JBoss `jboss.jcml` configuration file defines a ConnectionFactoryLoader for each of the three ConnectionFactory implementations. You would typically not need to define a ConnectionFactoryLoader MBean yourself unless you want to provide an alternate implementation of one of the JBossConnectionManagers.

As an example of how these classes are related, Figure 7.3 shows a ConnectionFactoryLoader associated with a MinervaXACMFactory, which is in turn associated with a MinervaXACM instance and a XAListener instance. The MinervaXACM is the JBossConnectionManager implementation that the MinervaXACMFactory creates. The XAListener instance is the `javax.resource.spi.ConnectionEventListener` callback interface implementation for MinervaXACM. Taken together, a ConnectionFactoryLoader and its classes associated through the MBean configuration make up the classes required by the JCA for the application server side of the system level contract.

The last piece of the resource adaptor integration puzzle is how a resource adaptor is loaded into the JBoss server and associated with a particular ConnectionFactory instance. There are two elements to this task. The first step is the creation of a resource adaptor archive (RAR) as defined by the JCA spec. The second step is to configure a ConnectionFactoryLoader MBean to specify the RAR deployment properties including the security settings, resource adaptor properties and the ConnectionFactory type to use. We'll look at the second step first and come back to the issue of creating a RAR for a skeleton resource adaptor.

ConnectionFactoryLoader MBean

In order for a resource adaptor's resource to be made available to application server components, a connection factory must be bound into JNDI. This is the standard pattern for all resources and it is a requirement of the JCA specification that resource adaptors provide a connection factory that can be bound into JNDI. The ConnectionFactoryLoader MBean is the JBoss deployment tool provided to allow a RAR administrator to configure all aspects of a resource adaptor deployment.

For each resource adaptor that is to be deployed in a JBoss server, one needs to define a ConnectionFactoryLoader MBean configuration and specify the attributes appropriate to the adaptor. The configurable attributes of the ConnectionFactoryLoader MBean are:

- **FactoryName:** The JNDI name of the connection factory. This is the name under which the connection factory will be bound in JNDI for access by application server components.
- **RARDeployerName:** The name of the MBean that will deploy the resource adaptor that this configuration relates to. Currently, JBoss lazily configures the adaptor separate from the RAR deployment step, and the two steps are tied together via a JMX notification. The ConnectionFactoryLoader MBean needs to know the name of the MBean that deploys RARs so that it can register for JMX notifications in order to configure the adaptor connection factory when the RAR is deployed. The default value of "JCA:service=RARDeployer" should only be changed if the RAR deployer MBean name is changed.
- **ResourceAdapterName:** The name of the resource adaptor for which this connection factory will create connections. This is the name given in the resource adaptor's <display-name> deployment descriptor element. It is used to filter JMX notifications to catch the deployment of the RAR.
- **Properties:** The properties to set on the resource adaptor to configure it to connect to a particular resource instance. This is in the java.util.Properties.load format (one property per line in the form name=value).
- **ConnectionFactoryName:** The name of the connection manager factory to use. This is the name given in a previously defined ConnectionFactoryLoader MBean. It should be one of the standard ConnectionManagerFactory implementations: MinervaNoTransCMFactory, MinervaSharedLocalCMFactory or MinervaXACMFactory.
- **ConnectionFactoryProperties:** The properties (in java.util.Properties.load format) to set on the connection manager for this connection factory. These properties control things such as connection pooling parameters.
- **PrincipalMappingClass:** The name of the class that implements the org.jboss.resource.security.PrincipalMapping interface. This class is responsible for mapping from the principal on behalf of whom the application component method is executing to the principal that will be used for validating access to the resource.

- **PrincipalMappingProperties:** The properties (in `java.util.Properties.load` format) to pass to the principal mapping implementation specified above via the `PrincipalMapping.setProperties` method.

RARDeployer MBean

The RARDeployer MBean service handles the deployment of archivesfiles containing resource adaptors (RARs). Deploying the rar file is the first step in making the resource adaptor available to application components. For each deployed, one or more connection factories must be configured and bound into JNDI, a task performed by the ConnectionFactoryLoader MBean service. The RARDeployer has no configurable attributes.

A Sample Skeleton JCA Resource Adaptor

To conclude our discussion of the JBossCX 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/example1` 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.

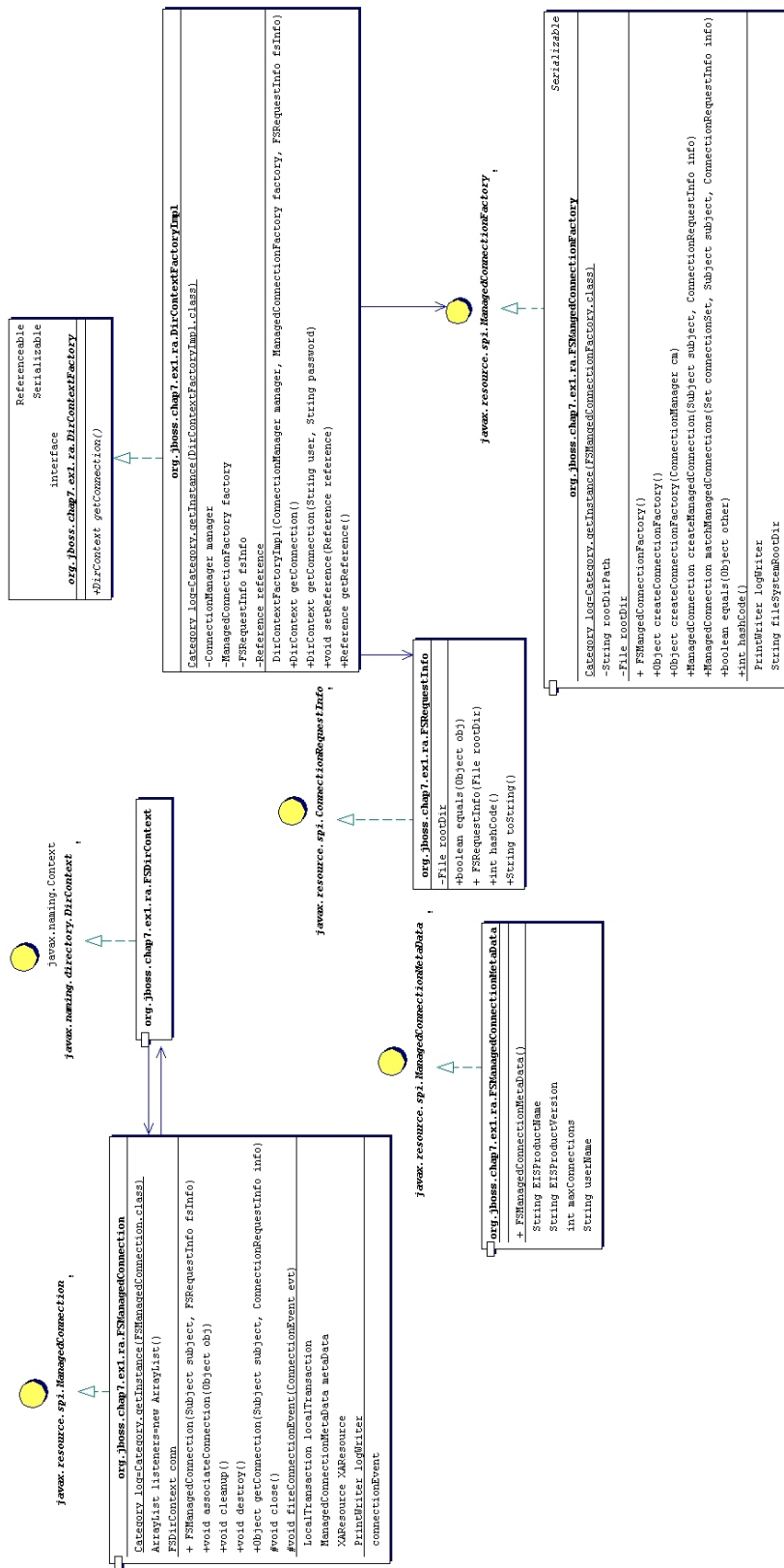


Figure 7-4, A class diagram for the example file system resource adaptor.

All of the resource adaptor classes in the diagram are from the `org.jboss.chap7.ex1` package. 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 JBossCX 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 808>ant -Dchap=7 build-chap
Buildfile: build.xml
...
chap7-ex1-rar:
    [jar] Building jar: D:\Scott\JBoss\SAMSBook\examples\build\chap7\ra.jar
    [jar] Building jar: D:\Scott\JBoss\SAMSBook\examples\build\chap7\chap7-ex1.rar

prepare:

chap7-ex1-jar:
    [jar] Building jar: D:\Scott\JBoss\SAMSBook\examples\build\chap7\chap7-ex1.jar
BUILD SUCCESSFUL
```

Next create a custom configuration of the JBoss server that includes a modified `jboss.jcml` file and `log4j.properties` file by executing the Chapter 7 configuration build as follows:

```
examples 1127>ant -Dchap=7 config
Buildfile: build.xml

config:

config:
    [copy] Copying 14 files to G:\JBoss-2.4.3\conf\chap7
    [copy] Copying 2 files to G:\JBoss-2.4.3\conf\chap7
BUILD SUCCESSFUL
```

This creates a chap7 configuration file set under the JBoss distribution conf directory. The example `jboss.jcml` file defines a `ConnectionFactoryLoader` MBean configuration for the example resource adaptor and this portion of the `jboss.jcml` file is presented in Listing 7-1.

Listing 7-1, the `jboss.jcml` `ConnectionFactoryLoader` MBean configuration fragment for the example resource adaptor.

```
<!-- The chap7 example1 filesystem resource adaptor -->
<mbean code="org.jboss.resource.ConnectionFactoryLoader"
      name="JCA:service=ConnectionFactoryLoader,name=NoTransFS">
  <attribute name="FactoryName">NoTransFS</attribute>
```

```

<attribute name="RARDeployerName">JCA:service=RARDeployer</attribute>
<attribute name="ResourceAdapterName">File System Adapter</attribute>
<attribute name="ConnectionFactoryName">
  MinervaNoTransCMFactory
</attribute>
<attribute name="ConnectionFactoryProperties" />
<!-- Principal mapping configuration -->
<attribute name="PrincipalMappingClass">
  org.jboss.resource.security.ManyToOnePrincipalMapping
</attribute>
<attribute name="PrincipalMappingProperties">
  userName=jduke
  password=theduke
</attribute>
</mbean>

```

The key attributes are:

- **FactoryName=NoTransFS**, this means that the adaptor factory will be bound into JNDI under the name "java:/NoTransFS".
- **ResourceAdapterName=File System Adapter**, this is the name of the display-name element of the resource adaptor rar.xml descriptor. These must match in order for the JMX notification to be received by the ConnectionFactoryLoader.
- **ConnectionFactoryName=MinervaNoTransCMFactory**, since this is a non-transaction capable resource adaptor we must choose the corresponding ConnectionFactory type.
- **PrincipalMappingClass= org.jboss.resource.security.ManyToOnePrincipalMapping**, we are choosing the default implementation of the PrincipalMapping interface that maps all caller principals onto the principal name and password given by the PrincipalMappingProperties set.
- **PrincipalMappingProperties=username=jduke;password=theduke**, these are the properties for the PrincipalMappingClass which assign the resource principal the name "jduke" with a password of "theduke" as the principal credentials.

Now startup the JBoss server using the chap7 configuration file set and the console should show output similar to the following:

```

bin 1426>run.bat chap7
JBoss_CLASSPATH=;run.jar;../lib/crimson.jar
jboss.home = G:\JBoss-2.4.5
Using JAAS LoginConfig: file:/G:/JBoss-2.4.5/conf/chap7/auth.conf
Using configuration "chap7"
[root] Started Log4jService, config=
      file:/G:/JBoss-2.4.5/conf/chap7/log4j.properties

```

```

...
[AutoDeployer] Auto deploy of
  file:/G:/JBoss-2.4.5/deploy/lib/chap7-ex1.rar
[RARDeployer] Attempting to deploy RAR at
  'file:/G:/JBoss-2.4.5/deploy/lib/chap7-ex1.rar'
[RARMetaData] License terms present. See deployment descriptor.
[NoTransFS] Not setting config property 'Password'
[NoTransFS] Not setting config property 'UserName'
[NoTransFS] Using default value '/tmp/db/fs_store' for config
  property 'FileSystemRootDir'
[NoTransFS] Bound connection factory for resource adaptor
  'File System Adapter' to JNDI name 'java:/NoTransFS'
[AutoDeployer] Started

```

This indicates that the resource adaptor has been successfully deployed and its connection factory has been bound into JNDI under the name 'java:/NoTransFS'. 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 7-2.

Listing 7-2, 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 814>ant -Dchap=7 -Dex=1 run-example
Buildfile: build.xml
...
run-example1:
  [copy] Copying 1 file to G:\JBoss-2.4.3\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 JBossCX layer. The output is in the log/chap7.log file of the JBoss server distribution. We'll summarize the events seen in the log using a sequence diagram. Figure 7-5 shows the events that occur during the deployment of the resource adaptor.

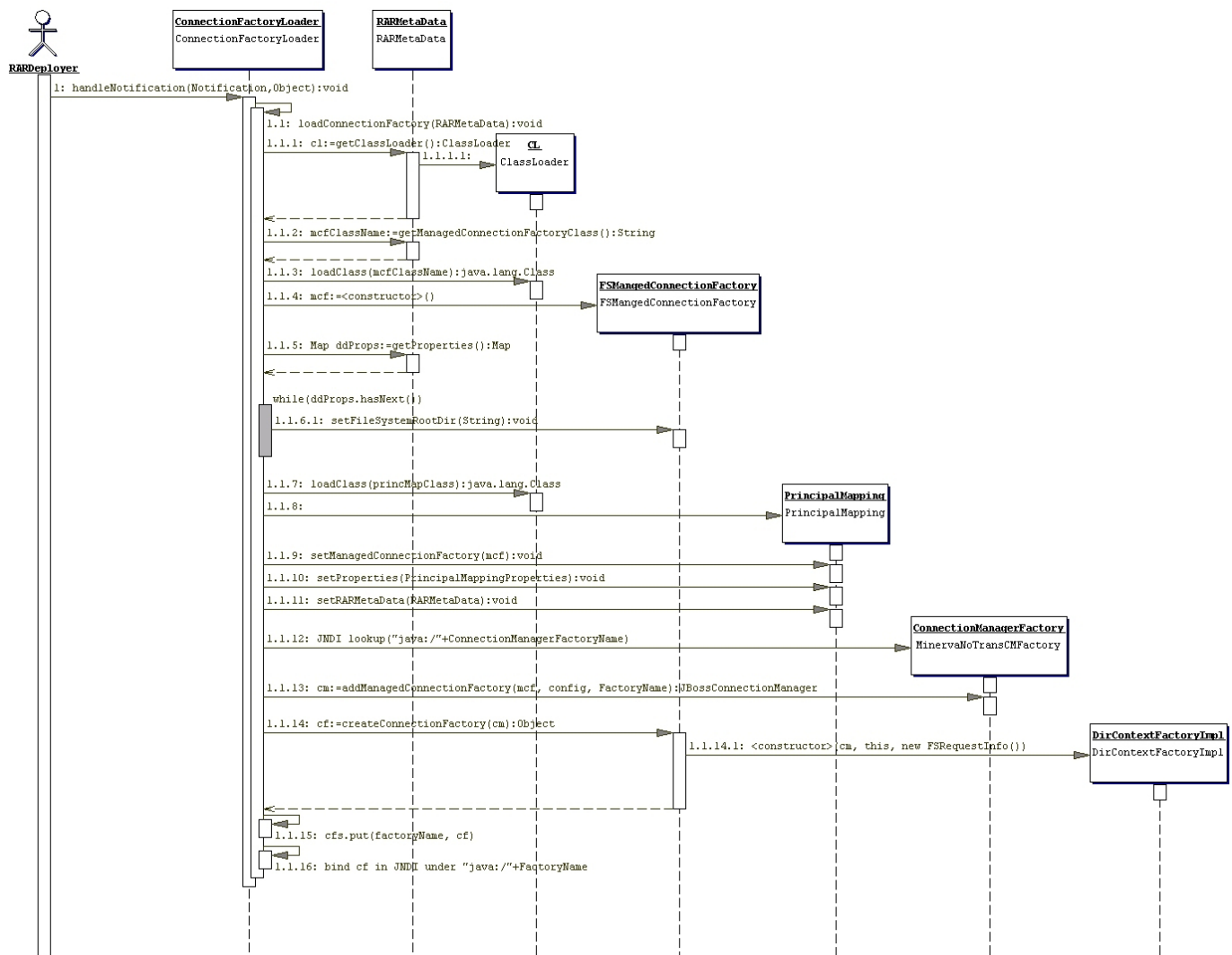


Figure 7-5, A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor during deployment.

The process starts when the RARDeployer MBean sends a notification that it has installed the resource adaptor found in the chap7-ex1.rar. The notification is processed by the handleNotification method of the ConnectionFactoryLoader MBean. The events that occur within the handleNotification method are as follows:

- 1.1, if the notification is a deployment event, the loadConnectionFactory method is called with the org.jboss.resource.RARMetaData object that was passed in with the event notification.

- 1.1.1, the ClassLoader object associated with the RARMetaData is obtained. This will be used to load the resource adaptor classes. This ClassLoader is assigned by the RARDeployer and has access to all of the classes in the RAR deployment.
- 1.1.2, the name of the ManagedConnectionFactory implementation class is obtained from the RARMetaData.
- 1.1.3, the ManagedConnectionFactory implementation class is loaded using the 1.1.1 ClassLoader instance.
- 1.1.4, the ManagedConnectionFactory implementation class is instantiated by invoking newInstance on the implementation class loaded in 1.1.3. This means that the implementation class must have a public no-arg constructor.
- 1.1.5, a java.util.Map of the properties defined in the RAR ra.xml descriptor is obtained from the RARMetaData instance. This represents the collection of config-property elements that were declared in the ra.xml descriptor.
- 1.1.6.1, an iteration over the properties loaded in 1.1.5 is performed and JavaBean style reflection is used to determine which properties have setters methods in the ManagedConnectionFactory instance. The diagram is illustrating that the FSManagedConnectionFactory class only has one setter for the FileSystemRootDir property. In general, any number of properties can be set by defining config-property elements in the ra.xml descriptor and implementing a JavaBean setter style method.
- 1.1.7, the class defined in the PrincipalMappingClass attribute of the ConnectionFactoryLoader MBean is loaded using the ClassLoader from 1.1.1. This is a class that implements the org.jboss.resource.security.PrincipalMapping interface.
- 1.1.8, the PrincipalMapping implementation class is instantiated by invoking newInstance on the class loaded in 1.1.7. This means that the implementation class must have a public no-arg constructor.
- 1.1.9, the ManagedConnectionFactory created in 1.1.4 is made available to the PrincipalMapping instance via the setManagedConnectionFactory method.
- 1.1.10, the value of the ConnectionFactoryLoader MBean PrincipalMappingProperties attribute is passed to the PrincipalMapping instance via the setProperties method.
- 1.1.11, the RARMetaData value is passed to the PrincipalMapping instance via the setRARMetaData method.

- 1.1.12, the `org.jboss.resource.ConnectionManagerFactory` instance to be used with the resource adaptor is located in JNDI. This is done using the name defined as the `ConnectionFactoryLoader` MBean `ConnectionManagerFactoryName` attribute value. In this case this locates an `org.jboss.pool.connector.jboss.MinervaNoTransCMFactory` instance since the `jboss.jcml` configuration specifies 'MinervaNoTransCMFactory' as the `ConnectionManagerFactoryName` value.
- 1.1.13, a `JBossConnectionManager` instance is created by invoking the `addManagedConnectionFactory` method on the `ConnectionManagerFactory` located in 1.1.12. This is the `javax.resource.spi.ConnectionManager` instance that will be associated with the resource adaptor.
- 1.1.14, a resource adaptor connection factory is created by invoking the `createConnectionFactory` method on the `ManagedConnectionFactory` instance. The `ConnectionManager` instance created in 1.1.13 is passed along with the method invocation.
- 1.1.14.1, the `FSManagedConnectionFactory` creates an instance of the example resource adaptor connection factory. This is a `DirConectFactoryImpl` instance. The `DirConectFactoryImpl` is provided the `ConnectionManager` instance as an argument to its constructor. The `DirConectFactoryImpl` will need this for creating connections as we will see later.
- 1.1.15, the resource adaptor connection factory is placed into a static `HashMap` of connection factory instances using the JNDI name of the factory as the key. This is used by the `javax.naming.spi.ObjectFactory.getObjectInstance` method implementation when a component does a lookup on the resource adaptor connection factory location.
- 1.1.16, the resource adaptor connection factory is bound into JNDI under the name 'java:/' + `FactoryName` where `FactoryName` is the value of the `ConnectionFactoryLoader` MBean `FactoryName` attribute. For this example, the full JNDI name is 'java:/NoTransFS' since 'NoTransFS' is the value of the `FactoryName` attribute. The actual value bound into JNDI is a `javax.naming.Reference` with the `ConnectionFactoryLoader` as the `ObjectFactory` implementation.

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 7-6 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.

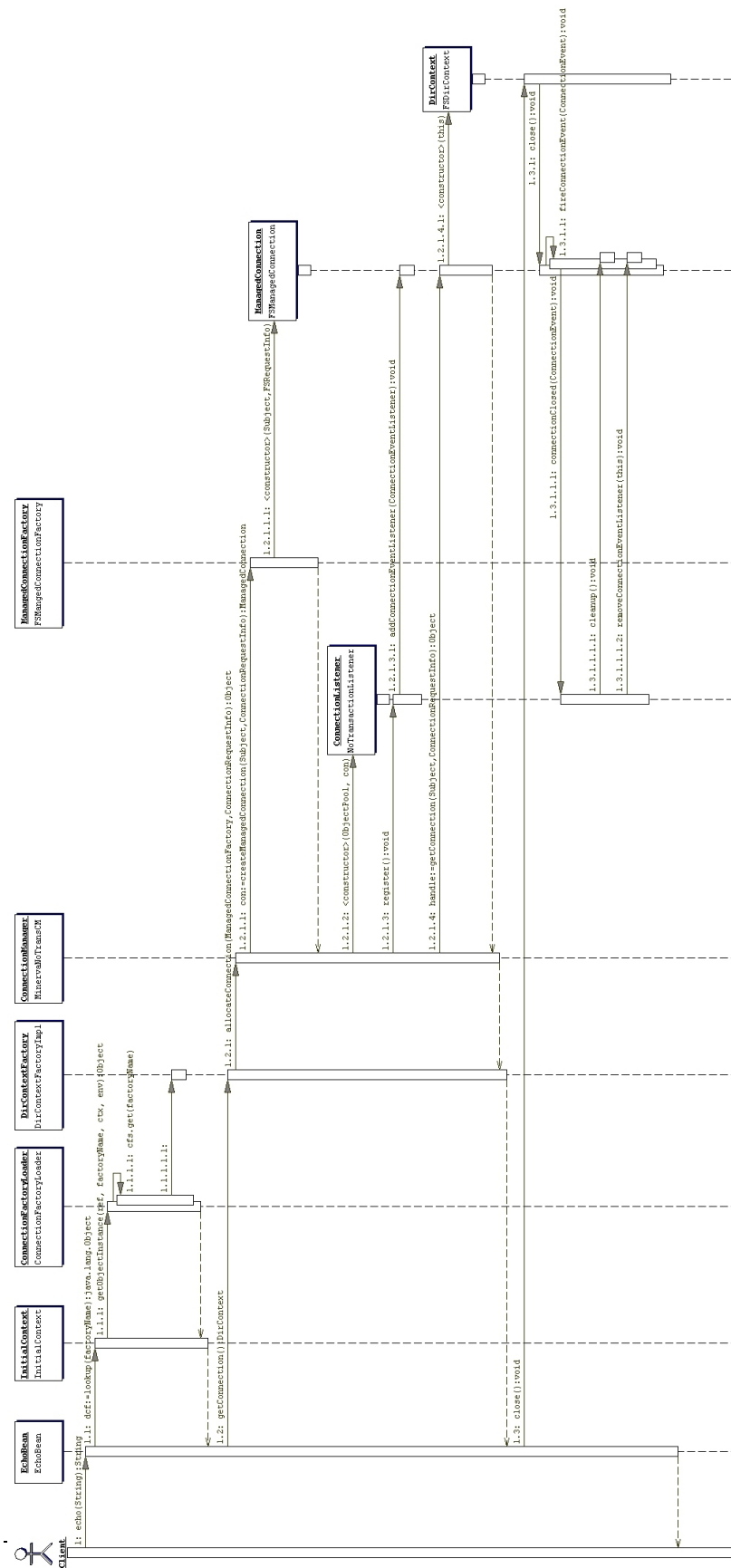


Figure 7-6, A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor that result when the EchoBean accesses the resource adaptor connection factory.

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 EchoBean 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:

- 1.1, the echo method performs a JNDI lookup of the resource adaptor connection factory.
- 1.1.1, the JNDI framework requests that value associated with the reference bound into JNDI using the reference ObjectFactory. The ObjectFactory is the ConnectionFactoryLoader MBean configured for the resource adaptor.
- 1.1.1.1, the ConnectionFactoryLoader locates the resource adaptor connection factory from the class map of connection factories using the name passed into the getObjectInstance method.
- 1.1.1.1.1, this signifies the location of the resource adaptor connection factory.
- the resulting Object returned by the lookup of 1.1 is cast to the DirContextFactory type so that it can be used to create a connection.

After the EchoBean has obtained the DirContextFactory for the resource adaptor, it creates a connection to the resource manager. This is illustrated by the 1.2 sequences of events, which are:

- 1.2, the getConnection method is invoked on the DirContextFactory obtained from the JNDI lookup.
- 1.2.1, 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.
- 1.2.1.1, the ConnectionManager asks its object pool for a connection object. 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

javax.security.auth.Subject associated with the pool as well as the FSRequestInfo data are passed as arguments to the createManagedConnection method invocation.

- 1.2.1.1.1, the FSManagedConnectionFactory creates a new FSManagedConnection instance and passes in the Subject and FSRequestInfo data.
- 1.2.1.2, 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.pool.connector.BaseConnectionManager\$NoTransactionListener instance.
- 1.2.1.3, the listener is asked to register with its connection.
- 1.2.1.3.1, the listener registers as a javax.resource.spi.ConnectionEventListener with the ManagedConnection instance created in 1.2.1.1.
- 1.2.1.4, the ManagedConnection is asked for the underlying resource manager connection. The Subject and FSRequestInfo data are passed as arguments to the getConnection method invocation.
- 1.2.1.4.1, the FSManagedConnection creates the example resource adaptor connection type by creating a FSDirContext instance.
- 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 is illustrated by the 1.3 sequences of events, which are:

- 1.3, the EchoBean invokes close on the DirContext to indicate the no further use of the connection will occur.
- 1.3.1, the FSDirContext instance invokes close on its associated FSManagedConnection to allow it to inform its listeners that close has been invoked.
- 1.3.1.1, the FSManagedConnection creates a javax.resource.spi.ConnectionEvent and invokes its fireConnectionEvent method to inform all registered ConnectionEventListeners of the event.
- 1.3.1.1.1, the NoTransactionListener is notified of the close event via its connectionClosed method.

- 1.3.1.1.1, the NoTransactionListener places the ManagedConnection back into the connection pool. This results in a cleanup invocation of the FSManagedConnection instance.
- 1.3.1.1.2, the NoTransactionListener removes itself as a ConnectionEventListener of the FSManagedConnection instance.

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.

Summary

This chapter introduced the JCA architecture and the JBoss implementation of the application server system contract. The JBossCX MBeans that support the integration of JCA resource adaptors was presented, and an example file system resource adaptor was created. This illustrated both the usage of the JBossCX MBeans and the interaction between the JBossCX JCA components and a resource adaptor.

You will next cover the JBoss security architecture and the default implementation provided by the JBossSX framework.

8. JBossSX – The JBoss Security Extension Framework

The JBossSX security extension provides support for both the declarative EJB 1.1 security model as well as integration of custom security via a security proxy layer

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.

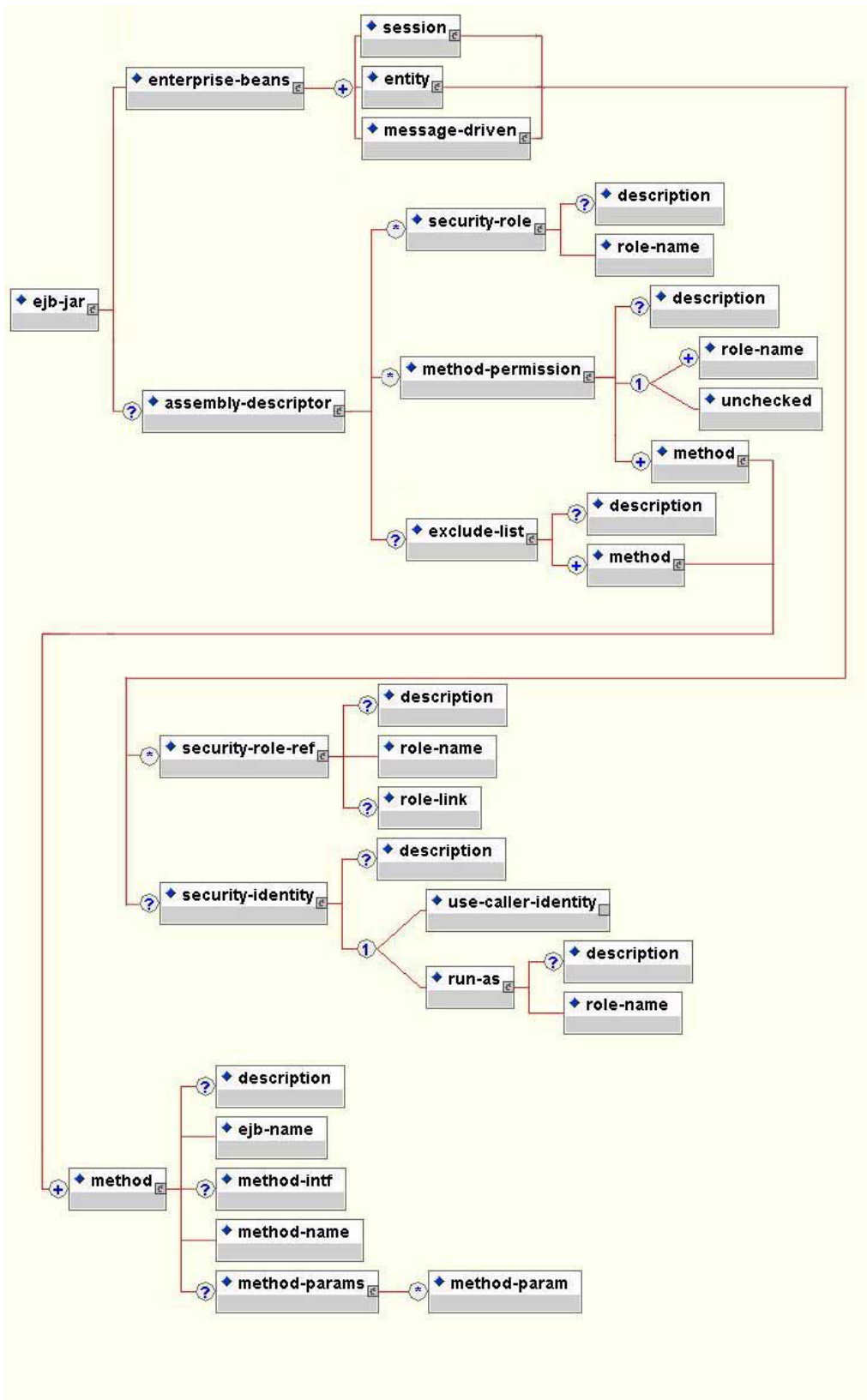


Figure 8-1, A subset of the EJB 2.0 deployment descriptor content model that shows the security related elements.

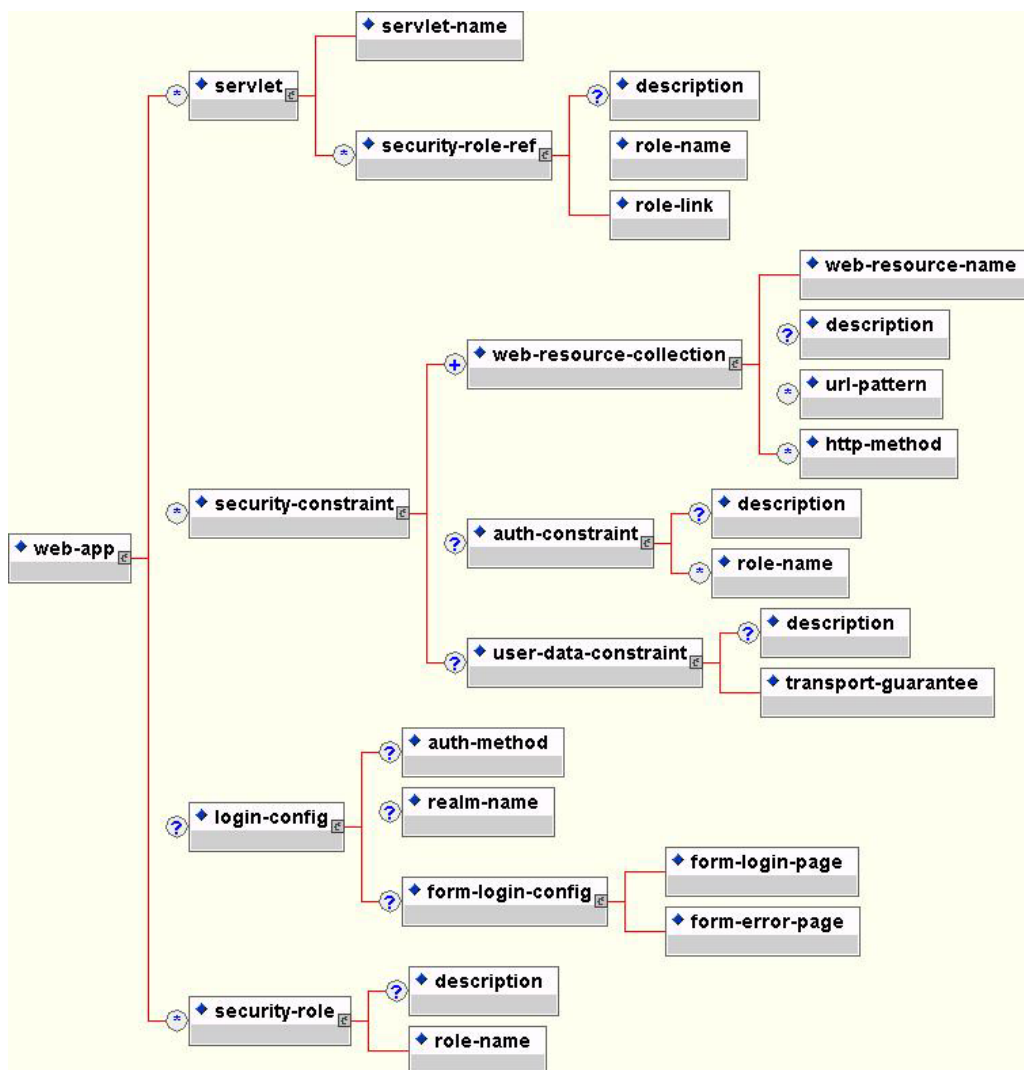


Figure 8-2, A subset of the Servlet 2.2 deployment descriptor content model that shows the security related elements.

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 8-1.

Listing 8-1, 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 internal 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 2.4.

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, 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>
      <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 "Enabling Declarative Security in JBoss Revisited".

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) framework 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 Model," later in this chapter.

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.

During 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 final class Subject implements java.io.Serializable
{
    ...
    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.RealmMapping, and org.jboss.security.SecurityProxy. Figure 8-3 shows a class diagram of the security interfaces and their relationship to the EJB container architecture.

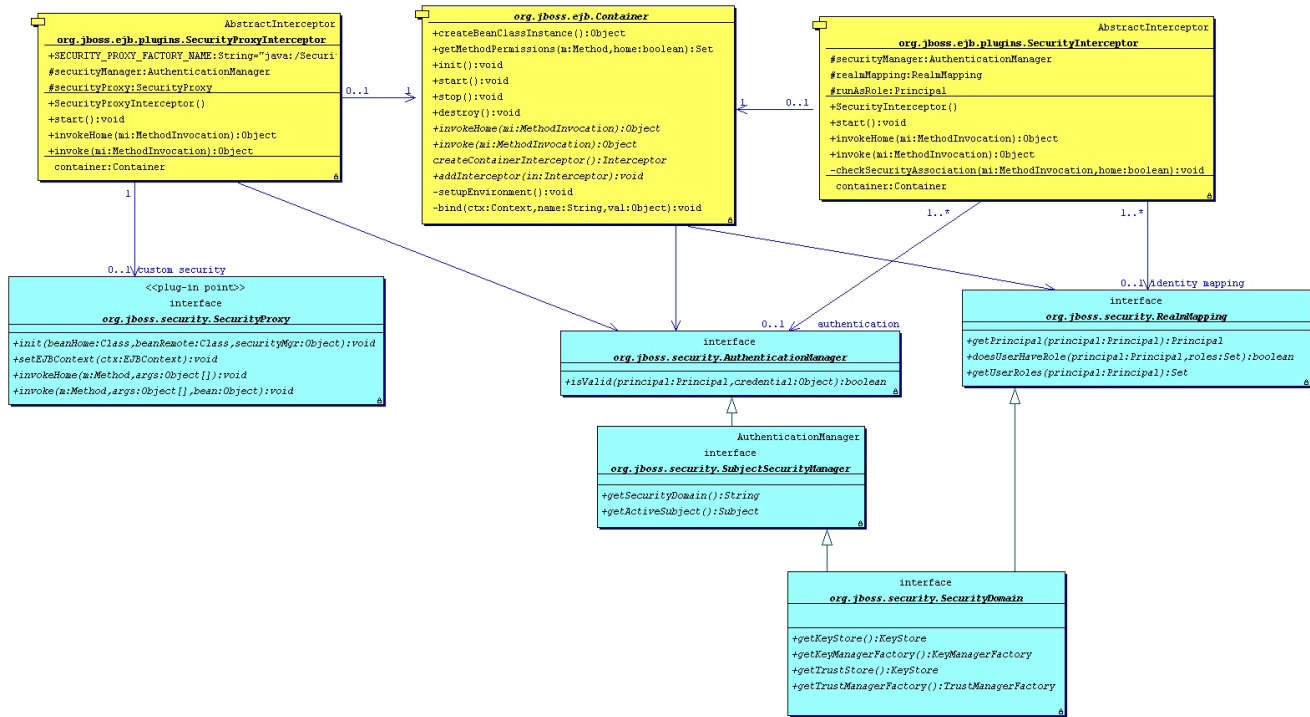


Figure 8-3, The key security model interfaces and their relationship to the JBoss server EJB container elements.

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 the section titled "The JBossSX security extension architecture".

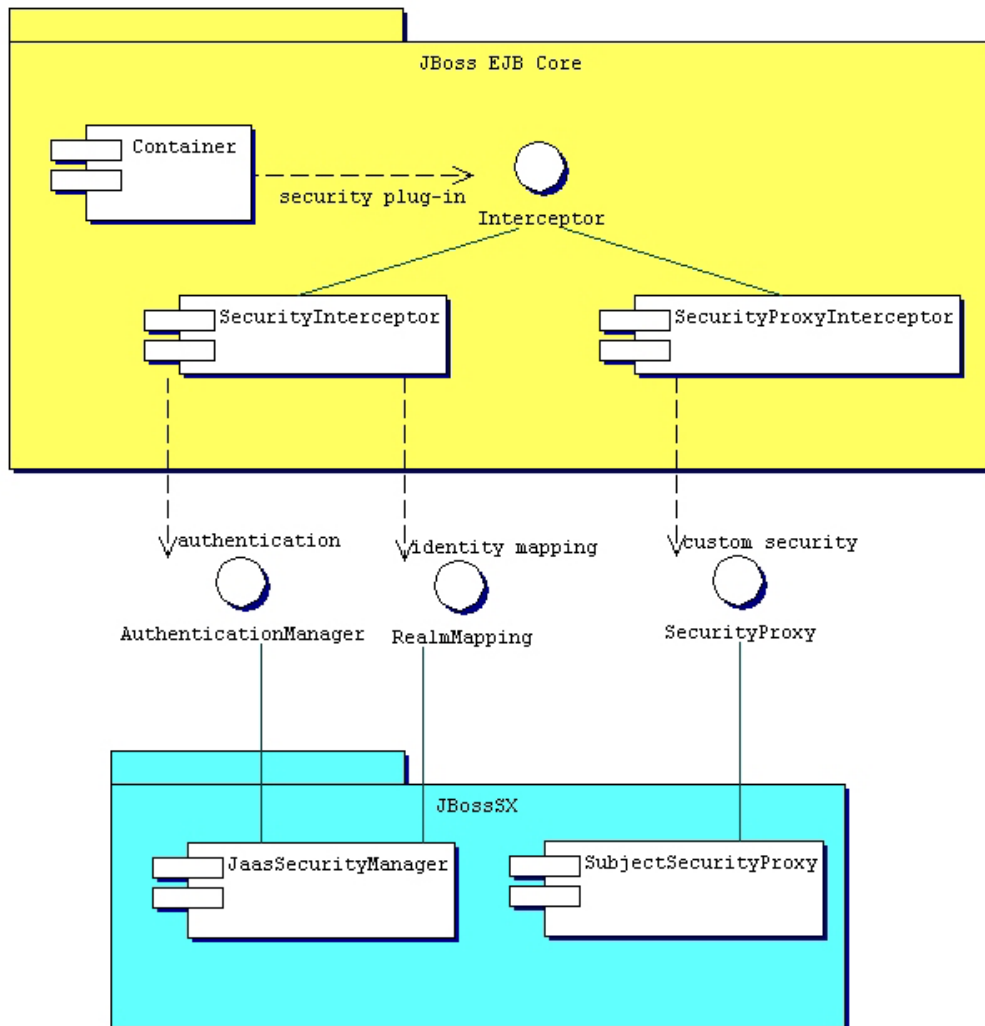


Figure 8-4, The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.

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.

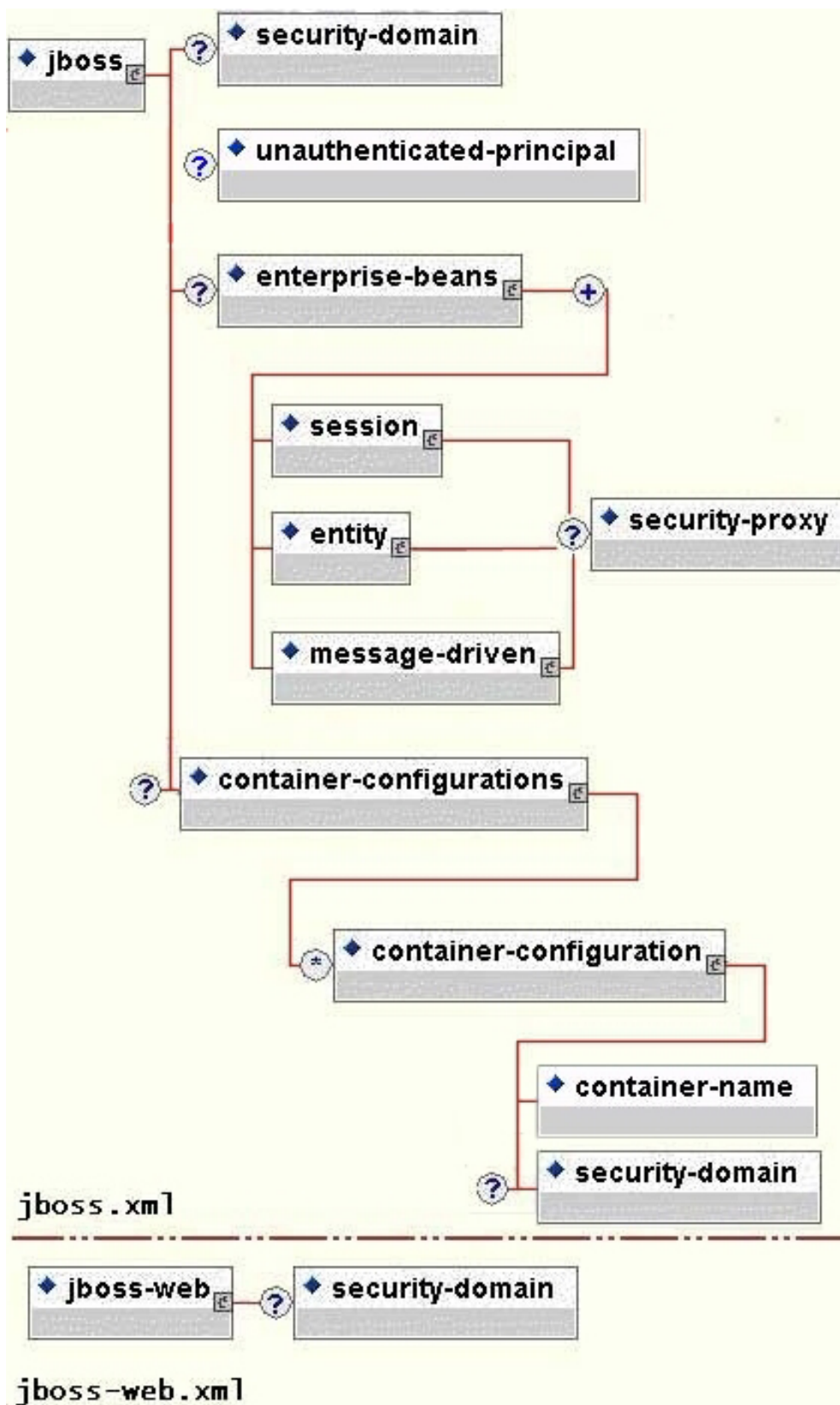


Figure 8-5, The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.

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 unsecured 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 find 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.

The first step to testing the custom security proxy is to configure the JBoss server with a configuration file set that enables unauthenticated users access to unsecured beans. To execute this step run the following Ant command from the book examples root directory:

```

examples 1015>ant -Dchap=8 config
Buildfile: build.xml

config:

config:
    [copy] Copying 14 files to G:\JBoss-2.4.5\conf\chap8

```



```
[copy] Copying 3 files to G:\JBoss-2.4.5\conf\chap8
BUILD SUCCESSFUL
Total time: 1 second
```

This creates a chap8 configuration file set under the JBoss distribution conf directory. Next, start the JBoss server using the chap8 configuration. You do this using either the run.bat or run.sh script, and pass the chap8 configuration name to the script as follows:

```
bin 1116>run.bat chap8
JBoss_CLASSPATH=;run.jar;../lib/crimson.jar
jboss.home = G:\JBoss-2.4.5
Using JAAS LoginConfig: file:/G:/JBoss-2.4.5/conf/chap8/auth.conf
Using configuration "chap8"
...
```

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
    {
        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();
        System.out.println("Created Echo");
        System.out.println("Echo.echo('Hello') = "+echo.echo("Hello"));
        System.out.println("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:

```
examples 1144>ant -Dchap=8 -Dex=1 run-example
Buildfile: build.xml

...
chap8-ex1-jar:

run-example1:
[copy] Copying 1 file to G:\JBoss-2.4.5\deploy
[echo] Waiting for 5 seconds for deploy...
[java] Created Echo
[java] Echo.echo('Hello') = Hello
```

```

[java] java.rmi.ServerException: RemoteException occurred in
server thread; nested exception is:
[java]     javax.transaction.TransactionRolledbackException:
SecurityProxy.invoke exception, principal=null, msg=No 4 letter words;
nested exception is:
[java]     java.lang.SecurityException: SecurityProxy.invoke
exception, principal=null, msg=No 4 letter words; nested exception is:
[java]     java.rmi.RemoteException: SecurityProxy.invoke exception,
principal=nullmsg=No 4 letter words; nested exception is:
...
[java]     at $Proxy1.echo(Unknown Source)
[java]     at org.jboss.chap8.ex1.ExClient.main(ExClient.java:20)
[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 so you should see quite a bit more than is shown here. 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 JBossSX 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.

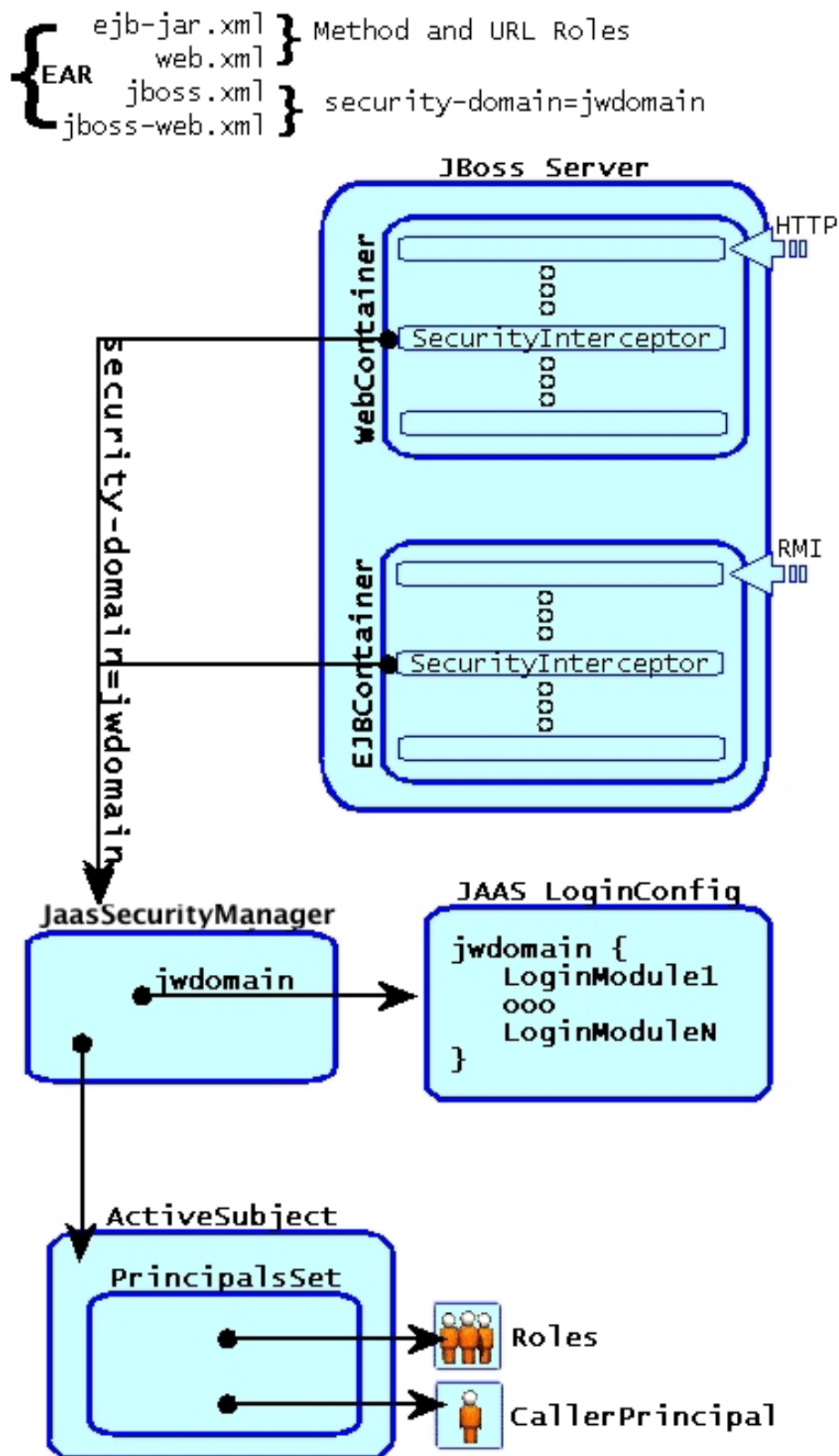


Figure 8-6, The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.

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.

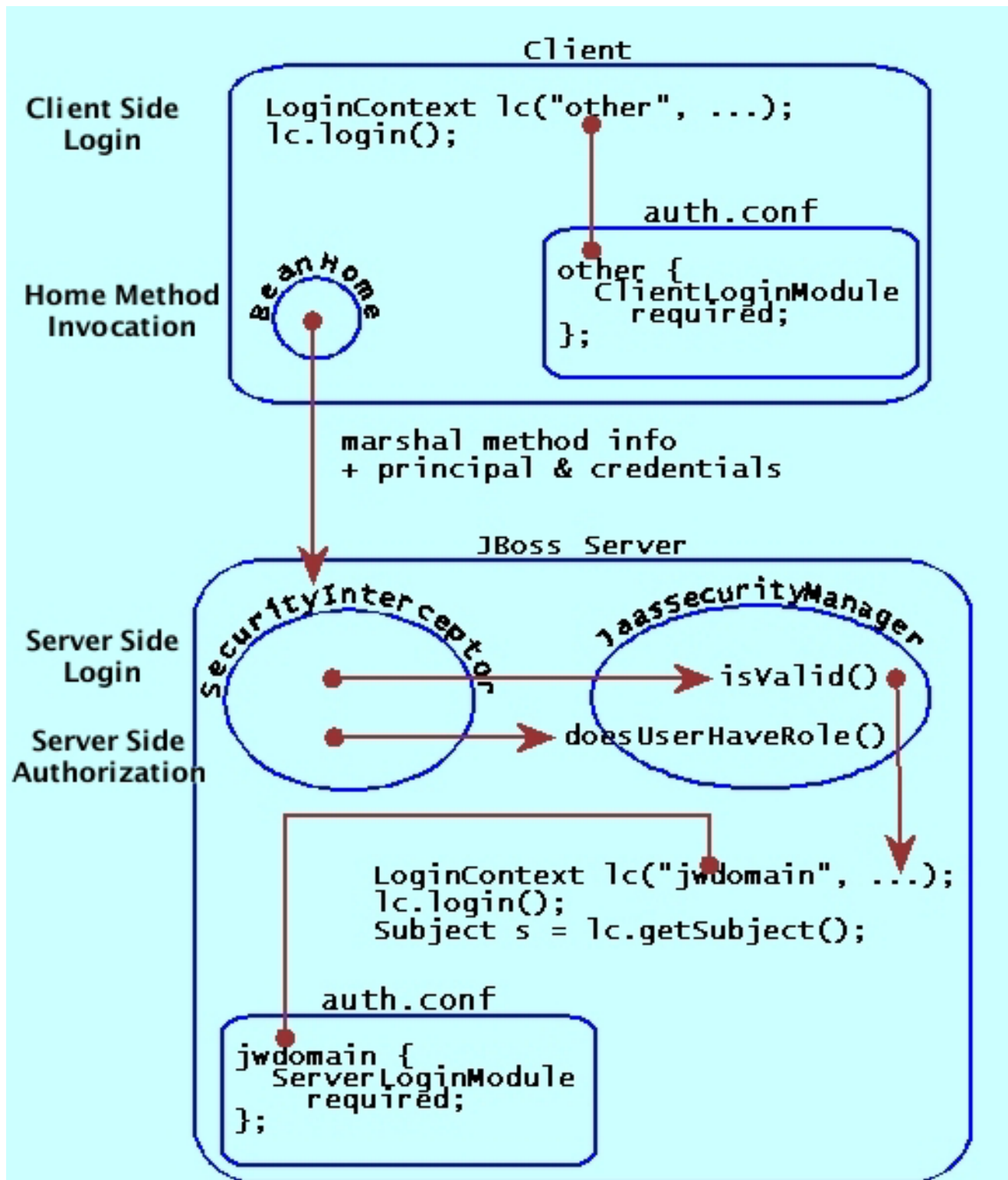


Figure 8-7, An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.

Figure 8-7 provides a view of the client to server communication we will discuss. The numbered steps shown in Figure 8-7 are:

- 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.
- 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.
- 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. This event is labeled as *Server Side Login* in Figure 8-7. 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 titled “Using and Writing JBossSX Login Modules” presented later.

- 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 Figure 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 synch 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.

- **LoginConfig:** Specifies the JMX ObjectName string of the that provides the default JAAS login configuration. When the `JaasSecurityManagerService` is started, this mean is queried for its `javax.security.auth.login.Configuration` by calling its `getConfiguration(Configuration currentConfig)` operation. The default configuration MBean returns the default Sun `Configuration` implementation described in the `Configuration` javadocs. In the future the default will be an XML based implementation.

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 `JaasSecurityManagerService`, 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

KeyStore.load(InputStream, char[]) method. If this is not specified null will be used and the integrity of the database will not be checked.

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 "How the `JaasSecurityManager` Uses JAAS" section. 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 username to password mappings. This defaults to `users.properties`.
- **rolesProperties=string**: (2.4.5+) The name of the properties resource containing the username 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 distinguished name to the context to search for user roles.
- **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 8-8. This figure corresponds to the testLdap login configuration just shown.

Server = ldap://ldaphost.jboss.org:1389

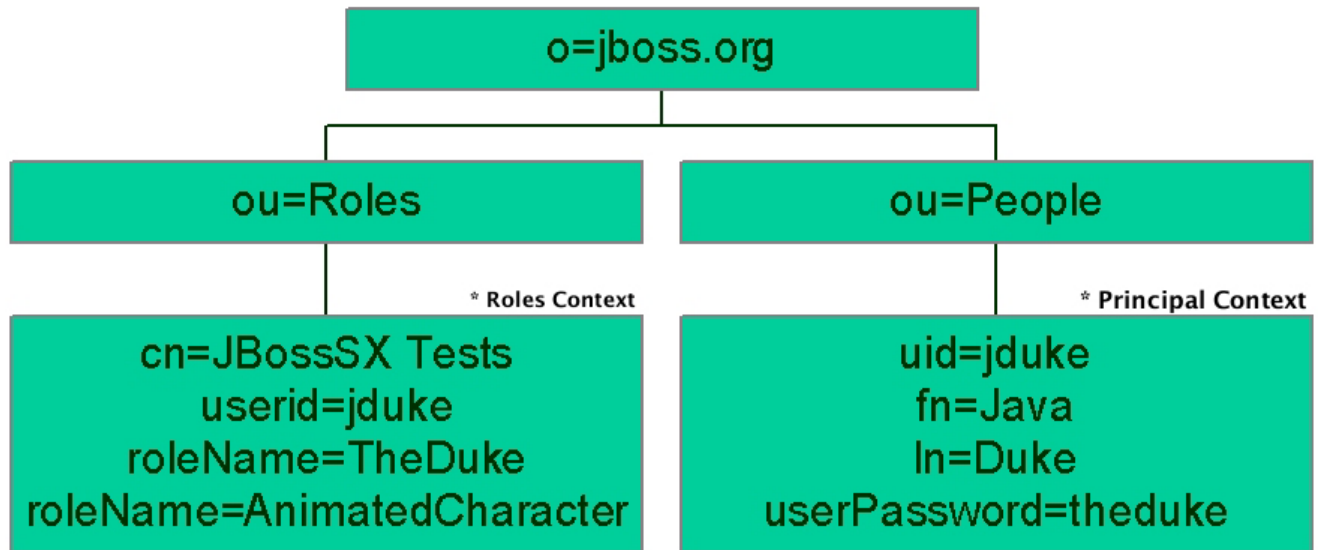


Figure 8-8, An LDAP server configuration compatible with the testLdap sample configuration.

Take a look at the testLdap login module configuration in comparison to the Figure 8-8 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 8-8 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 8-8 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 8-8, the roles that will be assigned to the user 'jduke' are 'TheDuke' and 'AnimatedCharacter'.

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.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.AbstractLoginModule 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;

    ...
    /** 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.
@param subject, the Subject to update after a successful login.
@param callbackHandler, the CallbackHandler that will be used
to obtain the user identity and credentials.
@param sharedState, a Map shared between all configured login
module instances
@param options,
    @option password-stacking: if true, the login identity will
be taken from the javax.security.auth.login.name value of
the sharedState map, and the proof of identity from the
javax.security.auth.login.password value of the sharedState
map.
*/
    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. Subclasses should override to perform the required
credential validation steps.
```

```

    */
    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;
}

```

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 AbstractLoginModule 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 AbstractLoginModule.

The steps you are required to perform when writing a custom login module are summerized 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 AbstractLoginModule 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 AbstractLoginModule, 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.
- `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.
- `String getUsersPassword();` to return the expected password for the current username available via the `getUsername()` method. The `getUsersPassword()` method is called from within `login()` after the CallbackHandler returns the username and candidate password.
- `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.

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 clients 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 8-9 gives a diagram of the key components involved in the JBossSX implementation of the SRP client/server framework.

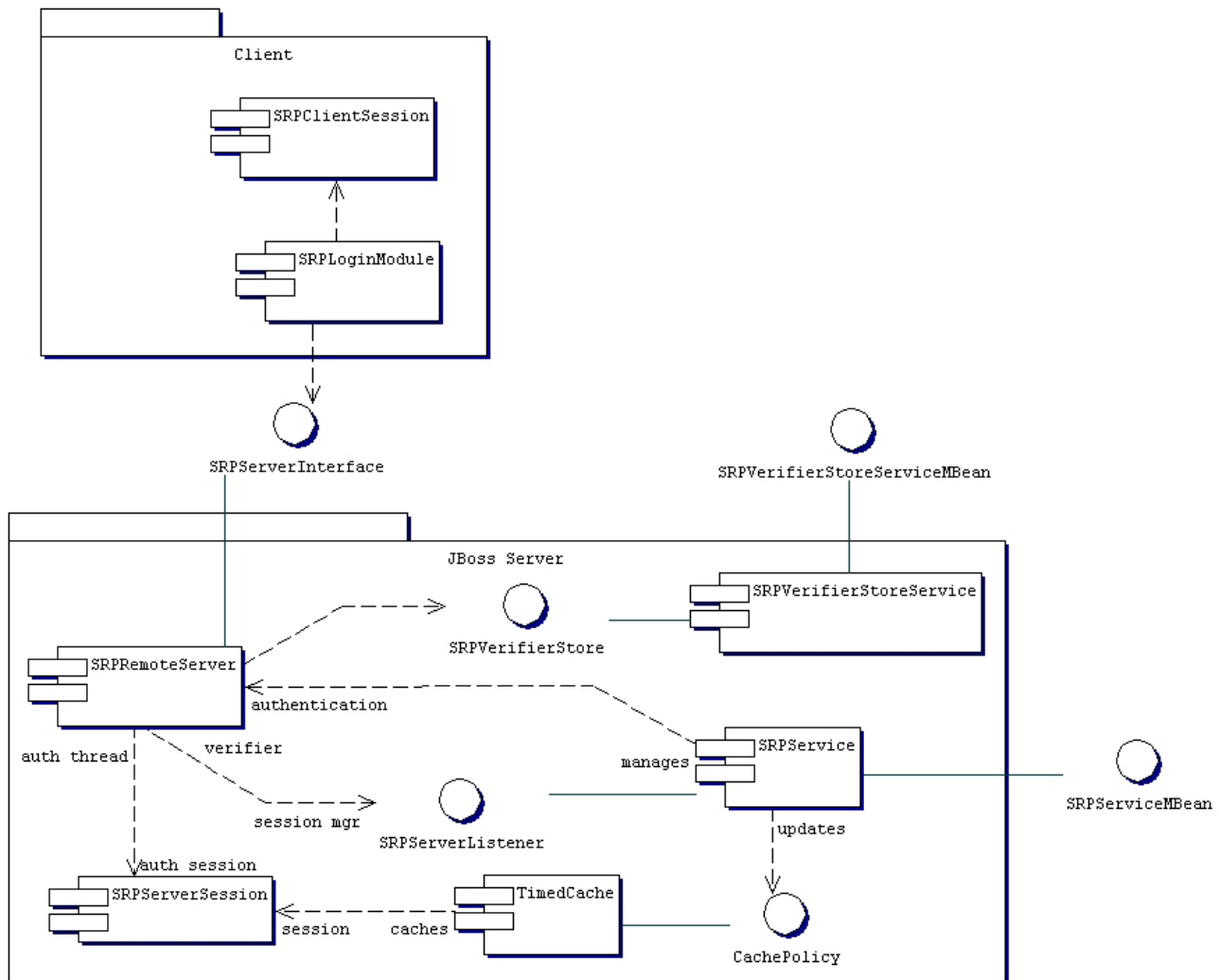


Figure 8-9, The JBossSX components of the SRP client-server framework.

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 configuring a login configuration entry that includes the org.jboss.security.srp.jaas.SRPLoginModule. This module supports the following configuration options:

- **principalClassName:** The fully qualified class name of the java.security.Principal implementation to use. The implementation must provide a public constructor that accepts a single String argument representing the name of the principal. If not specified this defaults to org.jboss.security.SimplePrincipal.

- **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.

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.SRPService 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 SRPService MBean attributes include the following:

- **JndiName:** The JNDI name from which the SRPServerInterface proxy should be available. This is the location where the SRPService 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 SRPService. 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 "An SRP example".

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 8-10 presents a sequence diagram of the authentication protocol as implemented by JBossSX.



Figure 8-10, The SRP client-server authentication algorithm sequence diagram.

The highlights of what is taking place for the key message exchanges presented in Figure 8-10 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 username 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 username.
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 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 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.
6. 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.

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. SSL can be used to prevent such masquerade uses of the opaque credential..
- 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.
- 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. The current association of the SRP session to the username limits one session per username. In the future this may be extended to support multiple user sessions.

At the end of this authentication protocol, the SRPServerSession has been placed into the SRPService authentication cache for subsequent use by the SRPCacheLoginModule. 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 8-11 illustrates the operation of the SRPCacheLoginModule.login method implementation.

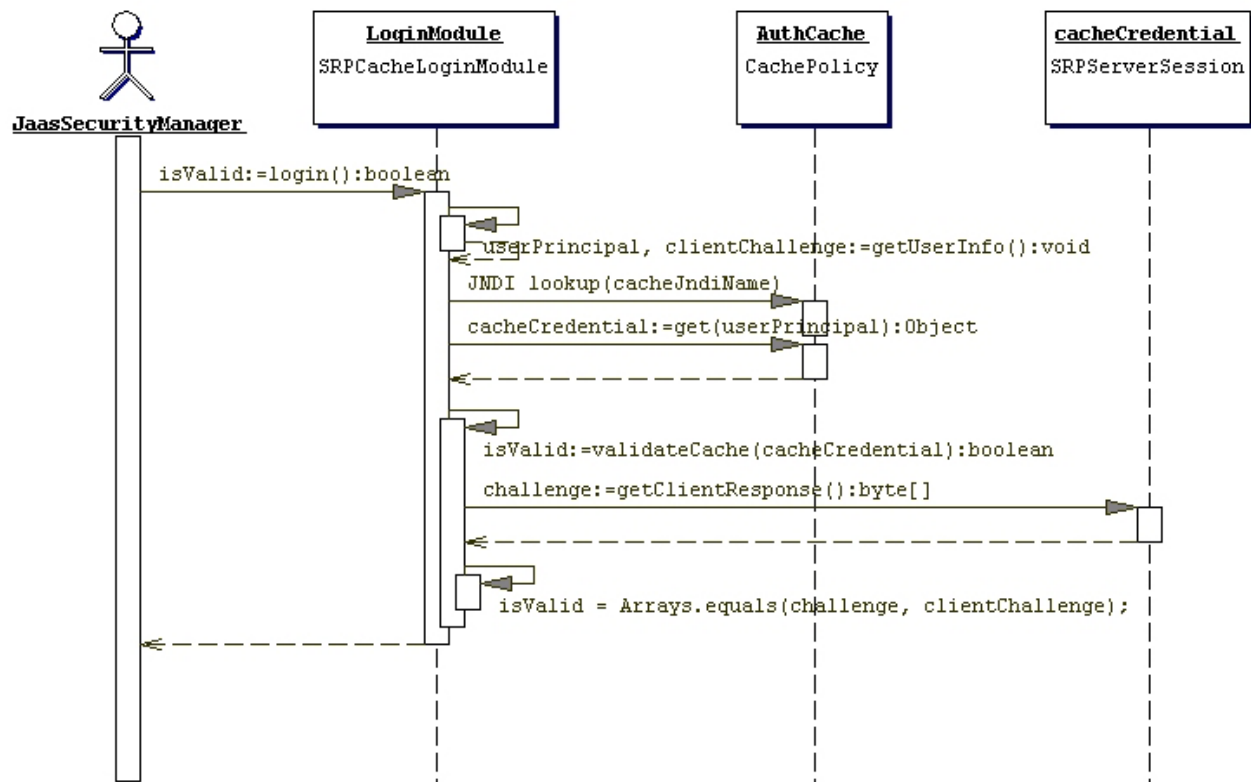


Figure 8-11, A sequence diagram illustrating the interaction of the *SRPCacheLoginModule* with the SRP session cache.

An SRP example

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. We'll use Ant to configure the JBoss server to use SRP as well as deploy and test the secured EJB and then look at the SRP related configuration changes that were performed.

The first step required is to configure the JBoss server with a configuration file set that enables the SRP MBeans we have discussed and adds a login configuration entry for the *SRPCacheLoginModule*. To execute this step, run the following Ant command from the book examples root directory:

```
examples 1015>ant -Dchap=8 config
Buildfile: build.xml

config:
```

```

config:
  [copy] Copying 14 files to G:\JBoss-2.4.5\conf\chap8
  [copy] Copying 3 files to G:\JBoss-2.4.5\conf\chap8

BUILD SUCCESSFUL

Total time: 1 second

```

This creates a chap8 configuration file set under the JBoss distribution conf directory. Next, start the JBoss server using the chap8 configuration. You need to do this using the `run_chap8.bat` or `run_chap8.sh` script as appropriate for your platform. These scripts were placed into the JBoss bin directory by the configuration step:

```

bin 1116>run_chap8.bat
JBoss_CLASSPATH=;run.jar;../lib/crimson.jar
jboss.home = G:\JBoss-2.4.5
Using JAAS LoginConfig: file:/G:/JBoss-2.4.5/conf/chap8/auth.conf
Using configuration "chap8"
...
[SRPVerifierStoreService] Starting
[SRPVerifierStoreService] Created SerialObjectStore at:
  G:\JBoss-2.4.5\bin\chap8_store.ser
[SRPVerifierStoreService] Started
[SRPService] Starting
[SRPRemoteServer] setVerifierStore, ...
[SRPService] Bound SRPServerProxy at srp/SRPServerInterface
[SRPService] Bound AuthenticationCache at srp/AuthenticationCache
[SRPService] Started
[Service Control] Started 48 services
[Default] JBoss 2.4.5 Started in 0m:6s

```

You can see at the end of the console output that the SRPVerifierStoreService and SRPService have been started. Run the example 2 client by executing the following command:

```

examples 1018>ant -Dchap=8 -Dex=2 run-example
Buildfile: build.xml
...
chap8-ex2-jar:
run-example2:
  [java] Accessing the Security:service=SRPVerifierStore MBean server
  [java] Creating username=jduke, password=theduke
  [java] User jduke added
  [copy] Copying 1 file to G:\JBoss-2.4.5\deploy
  [echo] Waiting for deploy...
  [java] Logging in using the 'srp' configuration
  [java] Created Echo
  [java] Echo.echo()#1 = This is call 1

```

```
[java] Echo.echo()#2 failed with exception:
      Authentication exception, principal=jduke
BUILD SUCCESSFUL
Total time: 31 seconds
```

In the examples directory you will find a file called `ex2-trace.log`. This is a detailed trace of the client side of the SRP algorithm. The tail of the JBoss `server.log` contains the server side trace 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. 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.

Listing 8-9 describes the configuration changes that were required to enable the use of the SRP authentication protocol. If you look at the bottom of the `conf/chap8/jboss.jcml` file, under the JBoss server root directory you will see the following entries shown in Listing 8-9.

Listing 8-9, the jboss.jcml SRP MBean service configuration entries used with example 2.

```
<!-- ===== -->
<!-- Add your custom MBeans here -->
<!-- ===== -->

<mbean code="org.jboss.security.srp.SRPVerifierStoreService"
  name="Security:service=SRPVerifierStore">
  <attribute name="JndiName">srp/Chap8VerifierStore</attribute>
  <attribute name="StoreFile">chap8-store.ser</attribute>
</mbean>

<mbean code="org.jboss.security.srp.SRPService"
  name="Security:service=SRPService">
  <attribute name="JndiName">srp/SRPServerInterface</attribute>
  <attribute name="VerifierSourceJndiName">
    srp/Chap8VerifierStore
  </attribute>
  <attribute name="AuthenticationCacheJndiName">
    srp/AuthenticationCache
  </attribute>
  <attribute name="AuthenticationCacheTimeout">10</attribute>
  <attribute name="AuthenticationCacheResolution">5</attribute>
```

```
<attribute name="ServerPort">12345</attribute>
</mbean>
```

This configures the SRPVerifierStoreService and SRPService MBean services using a number of non-default values for the supported attributes to demonstrate their use. The one item of note is that the `JndiName` attribute of the SRPVerifierStoreService is equal to the `VerifierSourceJndiName` attribute of the SRPService. This is required because the SRPService needs an implementation of the SRPVerifierStore interface for accessing user password verification information. Providing this implementation is the purpose of the SRPVerifierStoreService.

The other JBoss server configuration change was the addition of a login module configuration entry that used the SRPCacheLoginModule. If you look at the `conf/chap8/auth.conf` file you will see the two entries given in Listing 8-10.

Listing 8-10, the JBoss server `auth.conf` JAAS login configuration file used with example 2.

```
srp {
    org.jboss.security.auth.spi.ProxyLoginModule required
    moduleName=org.jboss.security.srp.jaas.SRPCacheLoginModule
    cacheJndiName="srp/AuthenticationCache"
    ;

    org.jboss.security.auth.spi.UsersRolesLoginModule required
    password-stacking=useFirstPass
    ;
};

// The default server login module
other {
    org.jboss.security.auth.spi.UsersRolesLoginModule required
    unauthenticatedIdentity="nobody";
};
```

It is the "srp" login configuration entry that is of interest. There are three issues to note about this configuration. First, note that you had to use the ProxyLoginModule to bootstrap the SRPCacheLoginModule. The reason for this is that the SRPCacheLoginModule used the org.jboss.util.TimedCache class located in the `lib/ext/jboss.jar` archive. This jar is not loaded by the system class loader, and thus would not be available if the SRPCacheLoginModule was loaded by the system class loader.

Also, the `cacheJndiName="srp/AuthenticationCache"` configuration option tells the SRPCacheLoginModule the location of the CachePolicy that contains the SRPServerSession for users who have authenticated against the SRPService.

Finally, the configuration includes a UsersRolesLoginModule with the `password-stacking=useFirstPass` configuration option. It is required to use a second login module with

the SRPCacheLoginModule because SRP is only an authentication technology. A second login module needs to be configured to accept the authentication credentials validated by the SRPCacheLoginModule to set the principal's roles that determines the principal's permissions.

The example entity bean you want to secure using this configuration includes a `jboss.xml` descriptor with a `security-domain` element value of `srp`. It also includes a `user.properties` file that is ignored and a `roles.properties` file that is used by the UsersRolesLoginModule to establish the roles for the user after authentication by the SRPCacheLoginModule.

An appropriate `auth.conf` file must also be created on the client side. Listing 8-11 shows the client side "srp" login configuration entry.

Listing 8-11, the JBoss client `auth.conf` JAAS login configuration file used with example 2.

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="srp/SRPServiceInterface"
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
    ;
};
```

The client configuration makes use of the SRPLoginModule with a `srpServerJndiName` option value that corresponds to the JBoss server component SRPService `JndiName` attribute value. Also needed is the ClientLoginModule configured with the `password-stacking="useFirstPass"` value to propagate the user authentication credentials to the EJB invocation layer.

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 8-12 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 8-12, 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 8-13 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 8-13, 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

A prerequisite to using SSL as described in this section is that you have the JSSE package installed as define by the JSSE installation guide. If you don't have the JSSE package you can download it from: <http://java.sun.com/products/jsse/index.html>. To test that you have JSEE installed as a VM extension, you need to be able to compile and run this simple installation test case:

```

import java.net.*;
import javax.net.ServerSocketFactory;
import javax.net.ssl.*;

public class JSSE_install_check
{
    public static void main(String[] args) throws Exception
    {
        ServerSocketFactory factory =
            SSLServerSocketFactory.getDefault();
        SSLServerSocket sslSocket = (SSLServerSocket)
            factory.createServerSocket(12345);

        String [] cipherSuites = sslSocket.getEnabledCipherSuites();
        for(int i = 0; i < cipherSuites.length; i++)
        {
            System.out.println("Cipher Suite " + i +
                " = " + cipherSuites[i]);
        }
    }
}

```

An example of a successful run is:

```

bin 1218>java -version
java version "1.3.1_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_01)
Java HotSpot(TM) Client VM (build 1.3.1_01, mixed mode)
bin 1219>java JSSE_install_check
Cipher Suite 0 = SSL_DHE_DSS_WITH_DES_CBC_SHA
Cipher Suite 1 = SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
Cipher Suite 2 = SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
Cipher Suite 3 = SSL_RSA_WITH_RC4_128_MD5
Cipher Suite 4 = SSL_RSA_WITH_RC4_128_SHA
Cipher Suite 5 = SSL_RSA_WITH_DES_CBC_SHA
Cipher Suite 6 = SSL_RSA_WITH_3DES_EDE_CBC_SHA
Cipher Suite 7 = SSL_RSA_EXPORT_WITH_RC4_40_MD5

```

If you don't want to, or cannot install JSSE as a VM extension, you can use JSSE with JBoss by placing the JSSE jars (jcert.jar, jnet.jar, jsse.jar) into the JBoss lib/ext directory. You will also need to configure a JaasSecurityDomain service as described in the following to ensure that the JSSE security provider is properly registered.

Once you have JSSE properly configured, you need a public key/private key pair in the form of an X509 certificate for use by the JBoss server. 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 configuration 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
=WA, C=US> correct?
    [no]:  yes

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 conf/chap8/chap8.keystore file contents using the keytool shows one self-signed certificate:

```

bin 1281>keytool -list -v -keystore ../conf/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 installed 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 container 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.RMICSSLClientSocketFactory 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.jcml` file in the `chap8` configuration file set directory already includes the following MBean entry:

```

<!-- 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 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 JBoss server configuration using the indicated password.

With this setup you are now able to perform the second step, which is to define an EJB container configuration that uses the JBossSX RMI socket factories that support SSL.

To do this you need to define a custom EJB container configuration using the `jboss.xml` descriptor. This requires knowledge that is not fully discussed until Chapter 9, “Advanced JBoss configuration using `jboss.xml`”. The configuration required to enable RMI over SSL access to stateless session bean is provided for you in Listing 8-14. You will use this configuration in a stateless session bean example.

Listing 8-14, The `jboss.xml` container configuration to enable SSL with stateless session beans.

```
<?xml version="1.0"?>
<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Standard Stateless SessionBean</container-name>
      <!-- Override the container socket factories -->
      <container-invoker-conf>
        <Optimized>true</Optimized>
        <RMIObjectPort>4445</RMIObjectPort>
        <RMIClientSocketFactory>
          org.jboss.security.ssl.RMISSLClientSocketFactory
        </RMIClientSocketFactory>
        <RMIServerSocketFactory>
          org.jboss.security.ssl.RMISSLServerSocketFactory
        </RMIServerSocketFactory>
        <ssl-domain>java:/jaas/RMI+SSL</ssl-domain>
      </container-invoker-conf>
    </container-configuration>
  </container-configurations>
</jboss>
```

It is the `container-invoker-conf` element that is doing all of the work. The key elements are as follows:

- **RMIObjectPort[md]** This specifies that the server listening port should be 4445 rather than the default of 4444. This must be changed so that both SSL and non-SSL stateless session beans may coexist.
- **RMIClientSocketFactory[md]** This specifies the client side of the RMI server socket factory. The `org.jboss.security.ssl.RMISSLClientSocketFactory` class is one that knows how to obtain the required JSSE information from a JaasSecurityDomain KeyStore.
- **RMIServerSocketFactory[md]** This specifies the server side of the RMI server socket factory. The `org.jboss.security.ssl.RMISSLServerSocketFactory` class is one that knows how to obtain the required JSSE information from a JaasSecurityDomain KeyStore.

- `ssl-domain[md]` This specifies the JNDI name of the JaasSecurityDomain that the SSL socket factories should use to obtain the KeyStore information needed for JSSE.

The example 3 code is located under the `src/main/org/jboss/chap8/ex3` 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 3 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 3a as follows:

```
examples 1130>ant -Dchap=8 -Dex=3a run-example
Buildfile: build.xml
...
run-example3a:
    [copy] Copying 1 file to G:\JBoss-2.4.5\deploy
    [echo] Waiting for 15 seconds for deploy...
java.rmi.MarshalException: Error marshaling transport header;
    nested exception is:
        javax.net.ssl.SSLException: untrusted server cert chain
javax.net.ssl.SSLException: untrusted server cert chain
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a
    at com.sun.net.ssl.internal.ssl.ClientHandshaker.a
    at com.sun.net.ssl.internal.ssl.ClientHandshaker.processMessage
    at com.sun.net.ssl.internal.ssl.Handshaker.process_record
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a
    at com.sun.net.ssl.internal.ssl.AppOutputStream.write
    at java.io.BufferedOutputStream.flushBuffer
    at java.io.BufferedOutputStream.flush
    at java.io.DataOutputStream.flush
    at sun.rmi.transport.tcp.TCPChannel.createConnection
    at sun.rmi.transport.tcp.TCPChannel.newConnection
    at sun.rmi.server.UnicastRef.invoke
    at org.jboss.ejb.plugins.jrmp.server.JRMPContainerInvoker_Stub.invokeHome
    at org.jboss.ejb.plugins.jrmp.interfaces.HomeProxy.invokeHome
    at org.jboss.ejb.plugins.jrmp.interfaces.HomeProxy.invoke
    at $Proxy0.create(Unknown Source)
    at org.jboss.chap8.ex3.ExClient.main(ExClient.java:25)
Exception in thread "main"
Java Result: 1
```

The resulting exception is expected, and is the purpose of the 3a 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 3 using -Dex=3 rather than -Dex=3a to pass the location of the correct truststore using the javax.net.ssl.trustStore system property:

```
examples 1118>ant -Dchap=8 -Dex=3 run-example
Buildfile: build.xml
...
run-example3:
  [copy] Copying 1 file to G:\JBoss-2.4.5\deploy
  [echo] Waiting for 5 seconds for deploy...
  [java] DEBUG [Thread-0] (RMISSSLClientSocketFactory.java:69) - SSL handshake
Completed, cipher=SSL_RSA_WITH_RC4_128_SHA, peerHost=172.17.66.54
  [java] 0 [Thread-0] DEBUG org.jboss.security.ssl.RMISSSLClientSocketFactory
- 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: 15 seconds
```

This time the only indication that an SSL socket is involved is because of the "SSL handshakeCompleted" message. This is coming from the RMISSSLClientSocketFactory 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 3a, 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.

Summary

J2EE Declarative Security Overview including the deployment descriptor security related elements for both EJBs and Web components. The use of the jboss.xml and jboss-web.xml descriptors to enable security was also discussed.

The generic JBoss security plug-in architecture was introduced. The interfaces that supported the J2EE declarative security model were discussed. Also discussed and demonstrated was the custom security interface that allowed one to enforce security constructs that cannot be described by the J2EE declarative model.

An introduction to JAAS that highlighted the authentication classes was given. The default JAAS based security manager plug-in implementation was discussed. This included its architecture, use of JAAS and MBean configuration. The standard supporting JAAS login module implementations shipped with JBoss were presented and their options detailed. You were also shown two abstract login modules supplied with JBoss that help in writing your own custom login modules.

The SRP protocol was also described. Both the algorithm and the JBoss implementation were discussed. An example was presented and a critique of the current JBoss implementation was given.

Lastly, the use of SSL with EJBs was described. A sample configuration was presented and an example of using SSL with a stateless session bean was given.

Next you will cover the jboss.xml descriptor. A complete description of the advanced configuration capabilities of the jboss.xml descriptor will be presented.

9. Advanced JBoss configuration using jboss.xml

The jboss.xml file is the JBoss specific EJB container configuration descriptor.

Although the jboss.xml and standardjboss.xml descriptors have been examined in many contexts in the book, the descriptor content model hasn't been described in its entirety yet. This chapter will focus on the descriptor content model along with all of the associated advanced configuration capabilities afforded by the jboss.xml descriptor that have not yet been discussed. Chapter 2, "JBoss Server Architecture Overview", showed that the jboss.xml and standardjboss.xml descriptors act together to form a two-level configuration. The first level is the standardjboss.xml file contained in the configuration file set directory, which acts as the default settings file. The second level is at the ejb-jar deployment unit. 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.

The jboss.xml Descriptor

The jboss.xml descriptor is the mechanism by which all configurable aspects of the JBoss EJB container are specified. In addition to the allowing a deployer to specify the EJB ENC and security bindings that have been covered in previous chapters, the jboss.xml descriptor allows for customization of most of the EJB container behavior. You will begin your introduction to the JBoss advanced configuration options by viewing the complete jboss.xml descriptor DTD. This is split across Figures 9.1 and 9.2 due to the size of the DTD image.

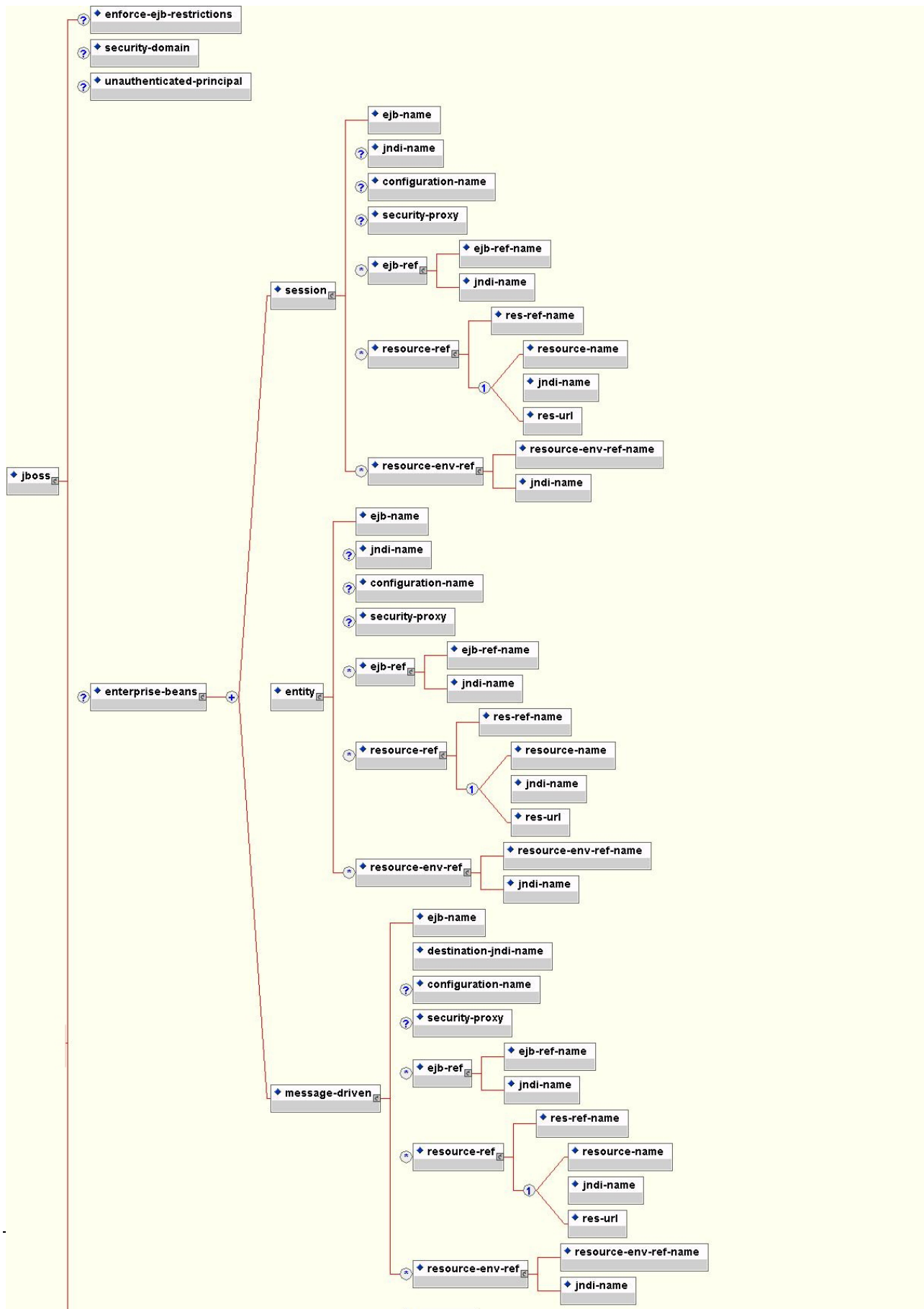


Figure 9-1, The jboss.xml descriptor DTD EJB related elements.

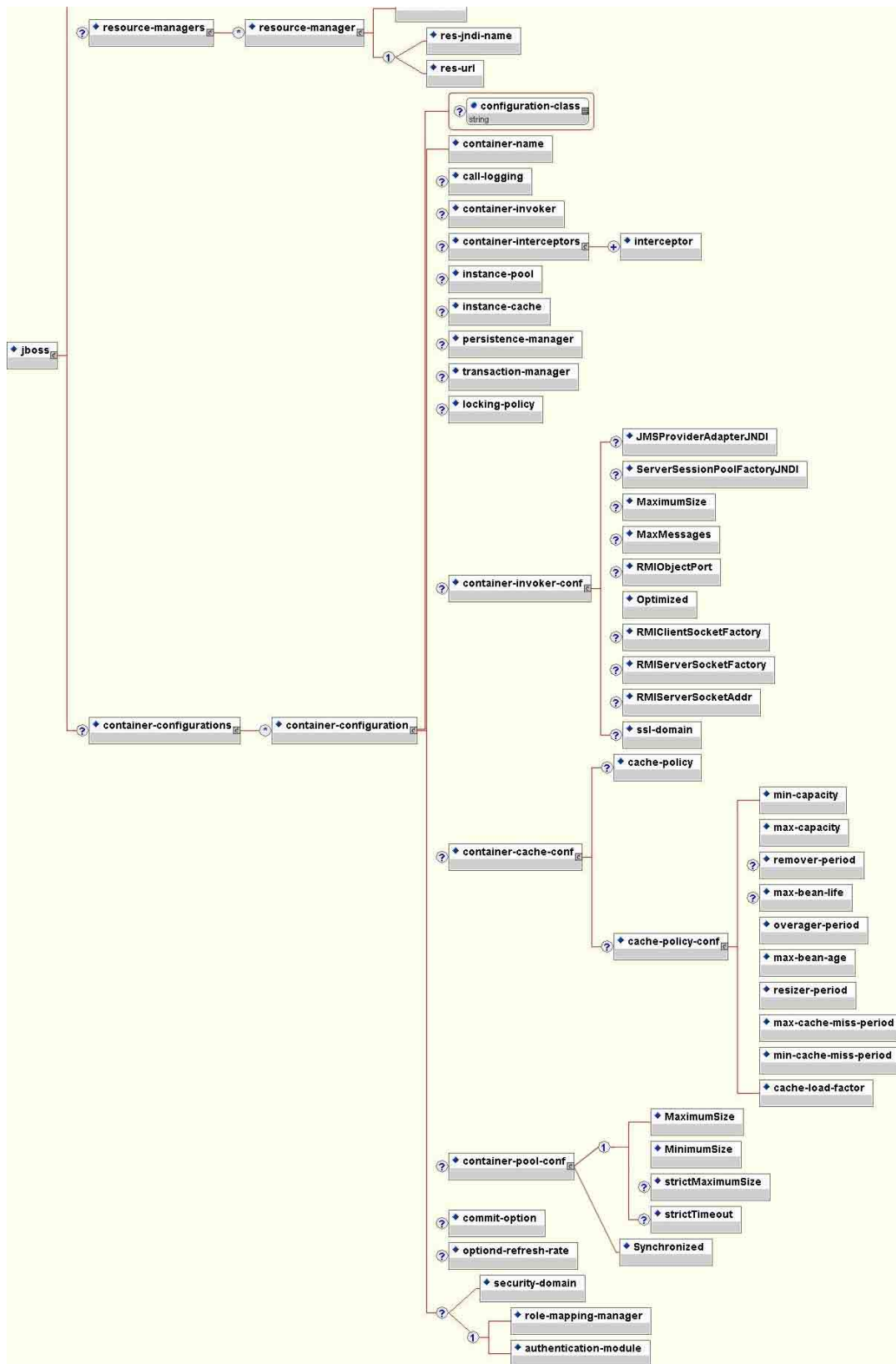


Figure 9-2, The *jboss.xml* descriptor DTD EJB container configuration related elements.

We have already seen nearly every element of the *jboss.xml* descriptor shown in Figure 9-1. The exceptions are the top-level [enforce-ejb-restrictions](#) element and the [configuration-name](#) element under each of the EJB element types. The [enforce-ejb-restrictions](#) element is a true/false flag that indicates whether the EJB programming restrictions defined in the EJB specification should be enforced by the JBoss EJB container. This is currently unsupported and setting it has no affect. The JBoss container does not enforce the EJB programming restrictions.

The [configuration-name](#) element is a link to a [container-configurations/container-configuration](#) element in Figure 9-2. 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 first specifying the name of an existing *standardjboss.xml* [container-configuration/container-name](#) as the value for the EJB [configuration-name](#) element. The desired *standardjboss.xml* descriptor [container-configuration](#) elements are then overridden by including a *jboss.xml* [container-configuration](#) that specifies the elements that are to be overridden. We have already seen an example of this in Chapter 8 "JBossSX – The JBoss Security Extension Framework", when we setup custom RMI socket factories that supported SSL. Listing 9-1 reproduces the *jboss.xml* descriptor that was used to do this.

Listing 9-1, the Chapter 8 *jboss.xml* container configuration to enable SSL with stateless session beans.

```
<?xml version="1.0"?>
<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Standard Stateless SessionBean</container-name>
      <!-- Override the container socket factories -->
      <container-invoker-conf>
        <Optimized>true</Optimized>
        <RMIObjectPort>4445</RMIObjectPort>
        <RMIClientSocketFactory>
          org.jboss.security.ssl.RMISSLClientSocketFactory
        </RMIClientSocketFactory>
        <RMIServerSocketFactory>
          org.jboss.security.ssl.RMISSLServerSocketFactory
        </RMIServerSocketFactory>
        <ssl-domain>java:/jaas/RMI+SSL</ssl-domain>
      </container-invoker-conf>
    </container-configuration>
  </container-configurations>
```

```
</jboss>
```

The first thing you might notice is that there is no EJB section containing a configuration-name element as per the first step of the procedure just described. For example, you might have expected the following at the start of Listing 9-1:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <configuration-name>Standard Stateless SessionBean
    </configuration-name>
    </session>
  </enterprise-beans>
  ...
```

The reason that this was not necessary is due to the fact that we were overriding the EJB's standard container configuration for stateless session beans. The JBoss EJB container factory needs a container configuration when it deploys an EJB. If the EJB has not provided 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 = 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 are overriding the default for 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 later in this chapter.

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 obsolete with the change to log4j, which provides a fine-grained logging API. We'll look at the log4j API in Chapter 11, "Using JBoss".

The container-invoker and container-invoker-conf elements

The container-invoker element specifies the class name of the org.jboss.ejb.ContainerInvoker implementation to use. The ContainerInvoker implementation is responsible for receiving remote method invocations for EJBs and forwarding the requests to the EJB container with which it is associated. Basically, the container invoker is the object that exposes a particular method invocation protocol. Currently there are two ContainerInvoker implementation choices available – org.jboss.ejb.plugins.jrmp.server.JRMPCContainerInvoker for RMI/JRMP access to session and entity beans, and org.jboss.ejb.plugins.jms.JMSContainerInvoker for JMS access to message driven beans.

The container-invoker class is configured using the container-invoker-conf element, provided that the class that implements the 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. The child elements of the container-invoker-conf element break down into two groups, which correspond to the two current ContainerInvoker

implementation classes. For the JRMPContainerInvoker, the following container-invoker-conf child elements are meaningful:

- Optimized, controls the bean method call argument and return value copy semantics within the Java VM in which JBoss is running. It can take the value true or false. If the value is false, then the container will behave per the EJB specification and all method arguments and return values of methods will be passed using RMI copy by value semantics, regardless of whether the client is running remotely or in the same VM as the JBoss server.
If the value of Optimized is true, then method arguments and return values will be passed by reference rather than by value. This is much more efficient, and can result in substantial performance improvements. However, it can also result in unexpected behavior if the objects passed to an EJB are stored as instance variables without first making a copy. Similarly, if a method return value is an instance variable of an EJB, a direct reference to the internal state of the EJB is returned and modifications to this value violate the security, transaction and multi-threaded contract the EJB container provides. Essentially the Optimized element allows any bean to behave as an EJB 2.0 local object and all caveats related to pass by reference semantics apply.
- 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.
- RMIClientSocketFactory, specifies a fully qualified class name for the java.rmi.server.RMIClientSocketFactory interface to use during export of the EJB home interface.
- RMI ServerSocketFactory, specifies a fully qualified class name for the java.rmi.server.RMIServerSocketFactory interface to use during export of the EJB home interface.
- RMIServerSocketAddr, 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 RMIServerSocketAddr value will be ignored.
- ssl-domain, 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 ssl-domain will be ignored.

For the JMSContainerInvoker, the following container-invoker-conf 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.

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 discussed the Interceptor interface in some detail in Chapter 2 when we talked 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 discussed the InstancePool interface in some detail in Chapter 2 when we talked 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 controlled 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. For further information on the requirements of the InstanceCache implementation see the "JBoss Server Architecture Overview" chapter discussion on 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 $\text{current-capacity} * 1/\text{cache-load-factor}$. 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".

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.

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 configured using the optiond-refresh-rate element.

The security-domain, role-mapping-manager and authentication-module elements

The security-domain element specifies the JNDI name of the object that implements the `org.jboss.security.AuthenticationManager` and `org.jboss.security.RealmMapping` interfaces. The role-mapping-manager and authentication-module elements are legacy notions that allowed one to specify the implementations of the `AuthenticationManager` and `RealmMapping` independently, but this is no longer supported.

Summary

Next you will be introduced to the JBoss support for integration third-partly servlet containers to support web components.



JBoss provides an abstract integration layer for third party web containers.

10. Integrating Servlet Containers

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 10-1 provides a complete view of the jboss-web.xml descriptor DTD for reference. The complete DTD with comments can be found in Appendix B in the section titled “The JBoss server jboss-web.xml descriptor DTD”.

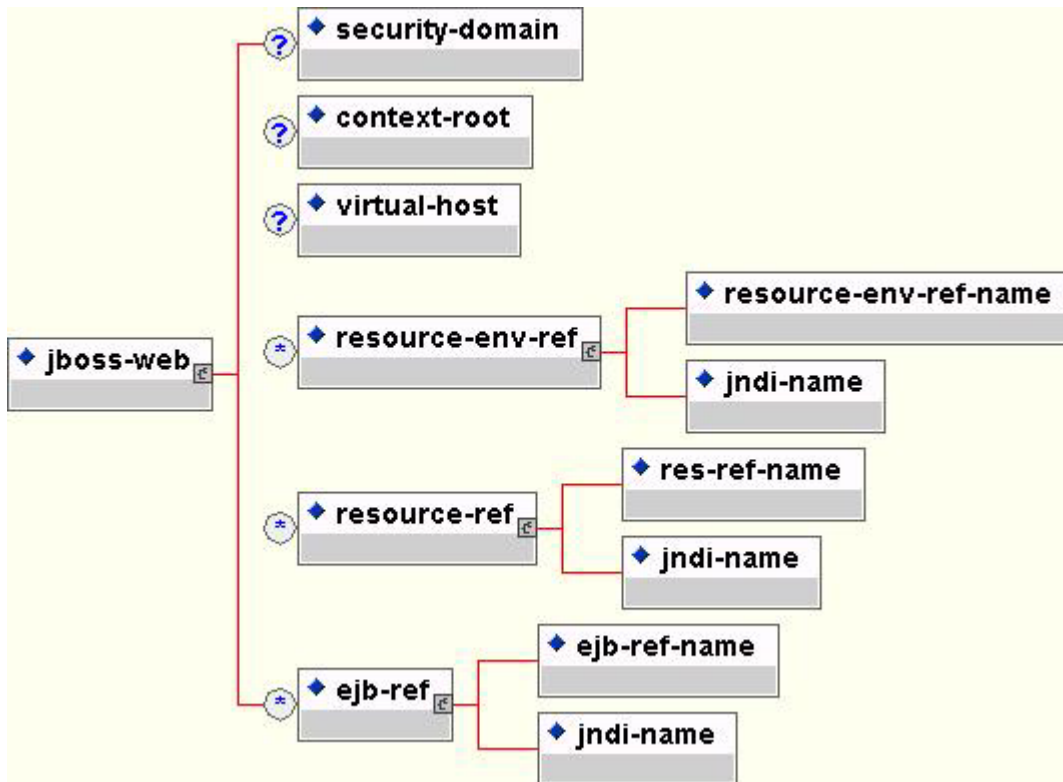


Figure 10-1, The complete *jboss-web.xml* descriptor DTD.

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 `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 10-1 presents some of the key `AbstractWebContainer` methods.

Listing 10-1, Key methods of the `AbstractWebContainer` class.

```
1:public abstract class AbstractWebContainer extends ServiceMBeanSupport
```



```

2:  implements AbstractWebContainerMBean
3:{
4:  public static interface WebDescriptorParser
5:  {
6:      public void parseWebAppDescriptors(ClassLoader loader,
7:          WebMetaData metaData) throws Exception;
8:  }
10: public void setConfig(Element config)
11: {
12:
13: }
15: public synchronized void deploy(String ctxPath, String warUrl)
16:     throws DeploymentException
17: {
18:     Thread thread = Thread.currentThread();
19:     ClassLoader appClassLoader = thread.getContextClassLoader();
20:     try
21:     {
22:         // Create a classloader for the war to ensure a unique ENC
23:         URL[] empty = {};
24:         URLClassLoader warLoader = URLClassLoader.newInstance(empty,
25:             appClassLoader);
26:         thread.setContextClassLoader(warLoader);
27:         WebDescriptorParser webAppParser = new DescriptorParser();
28:         // Parse the web.xml and jboss-web.xml descriptors
29:         WebMetaData metaData = parseMetaData(ctxPath, warUrl);
30:         WebApplication warInfo = new WebApplication(metaData);
31:         performDeploy(warInfo, warUrl, webAppParser);
32:         deploymentMap.put(warUrl, warInfo);
33:     }
34:     finally
35:     {
36:         thread.setContextClassLoader(appClassLoader);
37:     }
38: }
40: protected abstract void performDeploy(WebApplication webApp, String warUrl,
41:     WebDescriptorParser webAppParser) throws Exception;
43: public synchronized void undeploy(String warUrl) throws DeploymentException
44: {
45:     performUndeploy(warUrl);
46:     // Remove the web application ENC...
47:     deploymentMap.remove(warUrl);
48: }
50: protected abstract void performUndeploy(String warUrl) throws Exception;
52: protected WebMetaData parseMetaData(String ctxPath, String warUrl)
53:     throws MalformedURLException
54: {
55:     WebMetaData metaData = new WebMetaData();
56:     ...
57:     return metaData;
58: }
60: protected void parseWebAppDescriptors(ClassLoader loader, WebMetaData metaData)
61:     throws Exception
62: {

```

```

63:     try
64:     {
65:         // Create a java:comp/env environment unique for the web application
66:         Thread.currentThread().setContextClassLoader(loader);
67:         envCtx = (Context) iniCtx.lookup("java:comp");
68:         // Add a link to the global transaction manager
69:         envCtx.bind("UserTransaction", new LinkRef("UserTransaction"));
70:         envCtx = envCtx.createSubcontext("env");
71:     }
72:     finally
73:     {
74:         Thread.currentThread().setContextClassLoader(currentLoader);
75:     }
77:     addEnvEntries(envEntries, envCtx);
78:     linkResourceEnvRefs(resourceEnvRefs, envCtx);
79:     linkResourceRefs(resourceRefs, envCtx);
80:     linkEjbRefs(ejbRefs, envCtx);
81:     linkEjbLocalRefs(ejbLocalRefs, envCtx);
82:     linkSecurityDomain(securityDomain, envCtx);
83: }
85: protected void addEnvEntries(Iterator envEntries, Context envCtx)
86:     throws ClassNotFoundException, NamingException
87: {
88: }
90: protected void linkResourceEnvRefs(Iterator resourceEnvRefs, Context envCtx)
91:     throws NamingException
92: {
93: }
95: protected void linkResourceRefs(Iterator resourceRefs, Context envCtx)
96:     throws NamingException
97: {
98: }
100: protected void linkEjbRefs(Iterator ejbRefs, Context envCtx)
101:     throws NamingException
102: {
103: }
105: protected void linkEjbLocalRefs(Iterator ejbRefs, Context envCtx)
106:     throws NamingException
107: {
108: }
110: protected void linkSecurityDomain(String securityDomain, Context envCtx)
111:     throws NamingException
112: {
113: }
115: }

```

Lines 10-13 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.jcml` file. 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 15-38 correspond to the `deploy` method. This method is a template pattern method implementation. The arguments to the `deploy` method include `ctxPath`, which is the `context-root` element value from the J2EE `application.xml` descriptor. This may be null if the war is being deployed outside of an EAR(enterprise application archive). The `warUrl` argument is the URL string to the WAR that is to be deployed.

The first step of the `deploy` method is to save the current thread context `ClassLoader` and then create another `URLClassLoader` (`warLoader`) using the saved `ClassLoader` 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 18-25. Chapter 3, “JBossNS - The JBoss Naming Service,” 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 26, before the `performDeploy` call is made. Next, the `web.xml` and `jboss-web.xml` descriptors are parsed by calling `parseMetaData` on line 29. Next, the Web container-specific subclass is asked to perform the actual deployment of the WAR through the `performDeploy` call on line 31. The `WebApplication` object for this deployment is stored in the deployed application map using the `warUrl` as the key on line 32. The final step at line 36 is to restore the thread context `ClassLoader` to the one that existed at the start of the method.

Lines 40-41 give the signature for the abstract `performDeploy` method. This method is called by the `deploy` method and must be overridden by subclasses to perform the Web container specific deployment steps. A `WebApplication` is provided as an argument, and contains the metadata from the `web.xml` `web-app` 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 43-48 correspond to the undeploy method. This is a template pattern method implementation. Line 45 of this method calls the subclass performUndeploy method to perform the container-specific undeployment steps. Next, at line 47, the warUrl is unregistered from the deployment map. The warUrl argument is the string URL of the WAR as originally passed to the deploy method.

Line 50 gives the signature of the abstract performUndeploy method. This method is called as part of the undeploy() method template as shown on line 45. A call to performUndeploy asks the subclass to perform the Web container-specific undeployment steps.

Lines 52-58 correspond to the parseMetaData method. This is invoked as part of the deploy method and it parses the required WEB-INF/web.xml descriptor and the optional WEB-INF/jboss-web.xml descriptor and returns the deployment descriptor information as a org.jboss.metadata.WebMetData object.

Lines 60-83 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 77-82.

The addEnvEntries method on lines 85-88 creates the java:comp/env Web application env-entry bindings that were specified in the web.xml descriptor.

The linkResourceEnvRefs method on lines 90-93 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 95-98 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 100-103 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 105-108 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 110-113 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.

Creating an AbstractWebContainer Subclass

To integrate a web container into JBoss you need to create a subclass of AbstractWebContainer and implement the required performDeploy(String, 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 10-2 in quasi psuedo-code. Listing 10-3 provides the equivalent process for the authorization of a user.

Listing 10-2, a psuedo-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 10-3, a psuedo-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 = getContentRoles(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 . 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.
- **Java2ClassLoadingCompliance**, 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.
- **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 10-2, and the elements are described in the following section.

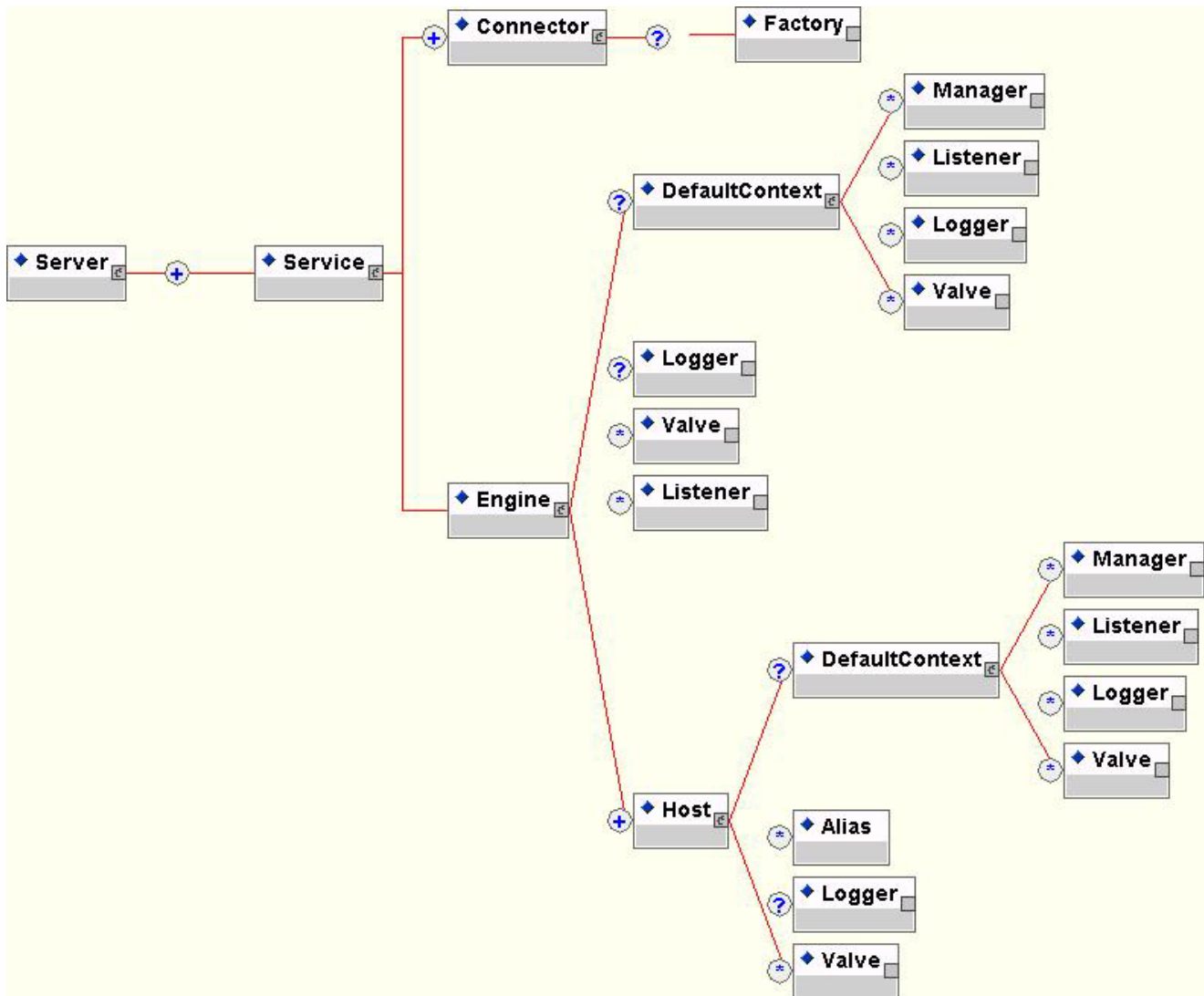


Figure 10-2, An overview of the Tomcat-4.0.3 configuration DTD supported by the *EmbeddedCatalinaServiceSX* Config attribute.

The Embedded Tomcat Configuration Elements

This section provides an overview of the Tomcat configuration elements that may appear as child elements of the EmbeddesCatalinaSX 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.
- **scheme**, sets the protocol name as accessed by the `ServletRequest.getScheme` method. The scheme defaults to http.
- **secure**, sets the `ServletRequest.isSecure` method value flag to indicate whether or not the transport channel is secure. This flag defaults to false.

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.apache.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 specifies 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 specifies 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 elment 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 three 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 KeyStore 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 `jboss.jcml` configuration file fragment that illustrates the setup of SSL via this approach, and which uses the same JaasSecurityDomain setup as Chapter 8 is given in Listing 10-4.

Listing 10-4, the JaasSecurityDomain and EmbeddedCatalinaSX MBean configurations for setting up Tomcat-4.x to use SSL as its primary connector protocol.

```
<server>
...
<!-- 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="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 = "../jboss/log" />
                        <DefaultContext cookies = "true" crossContext = "true" override = "true" />
                    </Host>
                </Engine>

                <!-- SSL/TLS Connector configuration -->
                <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>

```

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 **jboss.jcml** configuration file fragment that illustrates such a setup of SSL is given in Listing 10-5.

Listing 10-5, the JaasSecurityDomain and EmbeddedCatalinaSX MBean configurations for setting up Tomcat-4.x to use both non-SSL and SSL enabled HTTP connectors.

```

<server>
...
<!-- 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="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 = "../jboss/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 -->
        <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>

```

All 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 will likely 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 10-3. Figure 10-4 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 10-3, The Internet Explorer 5.5 security alert dialog.

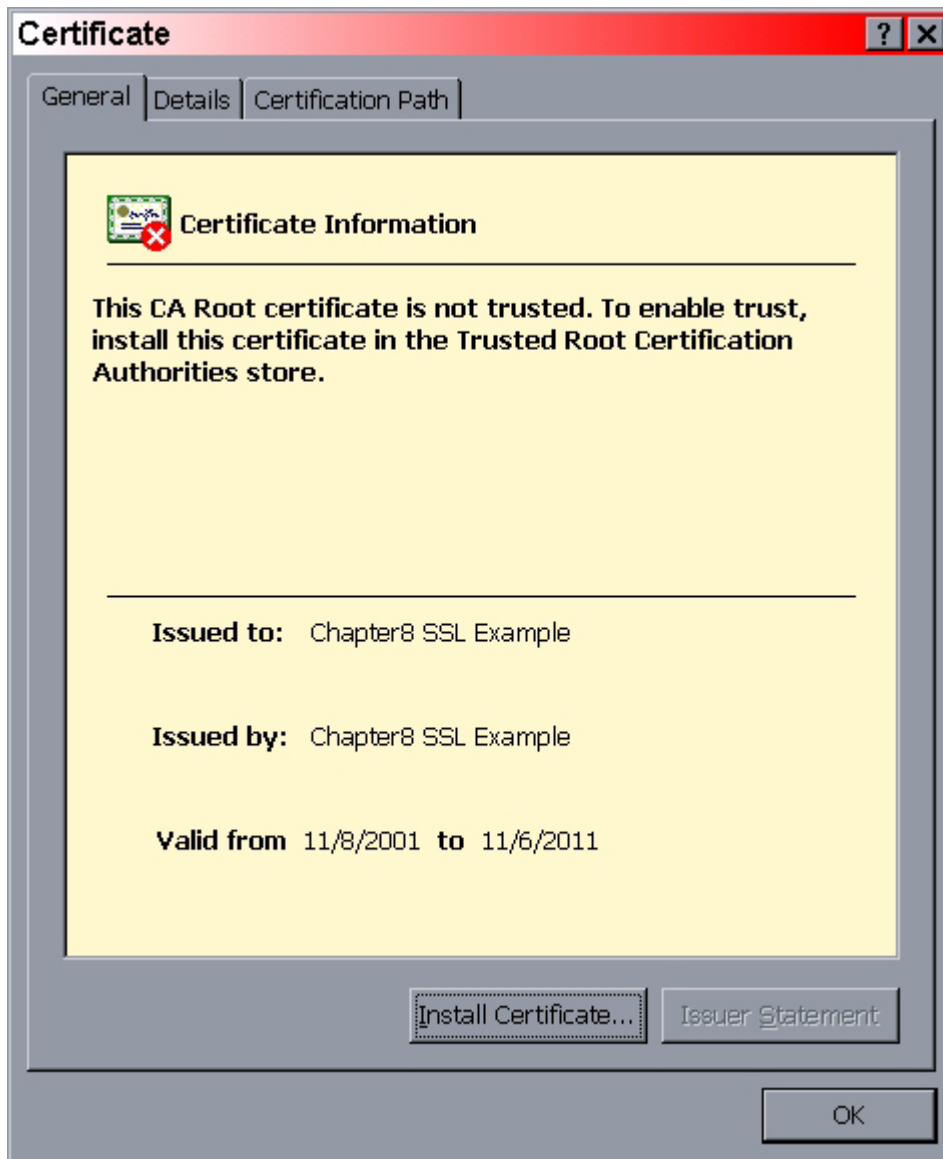


Figure 10-4, The Internet Explorer 5.5 SSL certificate details dialog.

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 `jboss.jcml` configuration fragment given in Listing 10-6. This configuration defines a default host named `localhost` and a second host named `banshee.starkinternational.com`. The `banshee.starkinternational.com` also has the aliases `www.starkinternational.com` associated with it.

Listing 10-6, 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 = "../jboss/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 = "../jboss/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 10-7, 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 10-8, Output from the `www.starkinternational.com` Host component when the Listing 10-7 WAR is deployed.

```
[INFO,AutoDeployer] Auto deploy of file:/tmp/JBoss-2.4.5RC3_Tomcat-
4.0.3/jboss/deploy/banshee.war
[INFO,J2eeDeployer] Deploy J2EE application: file:/tmp/JBoss-2.4.5RC3_Tomcat-
4.0.3/jboss/deploy/banshee.war
[INFO,J2eeDeployer] Create application banshee.war
[INFO,J2eeDeployer] inflate and install WEB module banshee.war
[INFO,ContainerFactory] Deploying:file:/tmp/JBoss-2.4.5RC3_Tomcat-
4.0.3/jboss/tmp/deploy/Default/banshee.war
[INFO,ContainerFactory] Deployed application: file:/tmp/JBoss-2.4.5RC3_Tomcat-
4.0.3/jboss/tmp/deploy/Default/banshee.war
[INFO,J2eeDeployer] Starting module banshee.war
[INFO,EmbeddedCatalinaServicesX] deploy, ctxPath=, warUrl=file:/tmp/JBoss-2.4.5RC3_Tomcat-
4.0.3/jboss/tmp/deploy/Default/banshee.war/web1003/
[INFO,Host=www] WebappLoader[:]: Deploying class repositories to work directory /tmp/JBoss-
2.4.5RC3_Tomcat-4.0.3/catalina/work/banshee.starkinternational.com/_
[INFO,Host=www] StandardManager[:]: Seeding random number generator class
java.security.SecureRandom
[INFO,Host=www] StandardManager[:]: Seeding of random number generator has been completed
[INFO,Host=www] ContextConfig[:]: Added certificates -> request attribute Valve
[INFO,EmbeddedCatalinaServicesX] Using Java2 parent classloader delegation: true
[INFO,Host=www] StandardWrapper[:default]: Loading container servlet default
[INFO,Host=www] default: init
[INFO,Host=www] StandardWrapper[:invoker]: Loading container servlet invoker
[INFO,Host=www] invoker: init
[INFO,Host=www] jsp: init
[INFO,J2eeDeployer] J2EE application: file:/tmp/JBoss-2.4.5RC3_Tomcat-
4.0.3/jboss/deploy/banshee.war is deployed.
```

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 10-9.

Listing 10-9, 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 = "../jboss/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 10-9 setup with a WAR deployed with a context root of “/jboss/test” 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        /jboss/test/* ajp13
</IfModule>

```

Other Apache to Tomcat configurations would follow the same pattern. All that would change it the Connector element definition that is placed into the EmbeddedCatalinaSX MBean configuration.

JBoss/Tomcat-3.2.3 Bundle Notes

In this section we'll discuss configuration issues specific to the JBoss/Tomcat-3.2.3 integration bundle. The Tomcat-3.2.3 servlet container supports the Servlet 2.2/JSP 1.1 specifications. The JBoss/Tomcat-3.2.3 integration layer is deprecated and no longer supported as of the 2.4.5 release of JBoss. The configuration of this bundle is done primarily configured through the external Tomcat `server.xml` file located in the `conf` directory of the tomcat distribution. The MBean used to embed the Tomcat-3.2.3 series of web containers is the `org.jboss.tomcat.EmbeddedTomcatServiceSX` service. It is a subclass of the `AbstractWebContainer` class. Its only configurable attribute is:

- **ConfigFile**, the path to the location of the tomcat container XML configuration file. If not specified the default value is `conf/server.xml`.

Using SSL with the JBoss/Tomcat bundle

Configuring the Tomcat-3.2.x servlet container to use SSL is performed as documented in the Tomcat and SSL howto on the Apache site, and can be found here:

<http://jakarta.apache.org/tomcat/tomcat-3.2-doc/tomcat-ssl-howto.html>. There has not been any work done to provide a custom SSL socket factory implementation that allows one to obtain the JSSE server certificate information from a JBossSX `SecurityDomain`.

JBoss/Jetty-4.0.0 Bundle Notes

In this section we'll discuss configuration issues specific to the JBoss/Jetty-3.1.3 integration bundle. The Jetty-4.0.0 servlet container supports the Servlet 2.3/JSP 1.2 specifications. The JBoss/Jetty integration layer is primarily configured through the `jboss.jcml` Jetty mbean configuration. The MBean used to embed the Jetty-4.x series of web containers is the `org.jboss.jetty.JettyService` service. It is a subclass of the `AbstractWebContainer` class. Its configurable attributes include:

- **WebDefault**, the default web-app deployment descriptor settings, for example, `webdefault.xml` for the standard JBoss/Jetty bundle.
- **UnpackWars**, flag indicating if war archives should be unjared.
- **Java2ClassLoadingCompliance**, 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.

- **ConfigurationElement**, the embedded configuration for the Jetty servlet container. It is a subset of the standard Jetty `jetty.xml` configuration.

Summary

You were introduced to the AbstractWebContainer integration MBean that handles the JBoss specific servlet container integration tasks. You were also introduced to the default subclass implementations of the AbstractWebContainer that are used to create the Tomcat-4.x and Jetty-4.x bundled releases of JBoss.

Next, you will be introduced to two key tools used by JBoss, the Ant build tool and the Log4j logging API. You will also investigate two enterprise application deployments to gain experience with all of the step required for deploying application under JBoss.



11. Using JBoss

Introductions to Ant and Log4j as well as using JBoss to run applications

Until this point the focus of this book has been largely on the architecture and configuration of the various JBoss server components. The focus of this chapter switches to running J2EE applications using JBoss. The chapter starts with a discussion of a sample mail management application, then dicusses porting the J2EE blueprints 1.1.2 Java Pet Store application to JBoss, and concludes by covering the JBossTest unit test suite.

Building and running enterprise applications with JBoss

In this section we will go through the details of creating an email forwarding application that uses a number of J2EE components as well as a custom MBean service. This will provide a detailed example of building, configuring, and deploying a non-trivial application under JBoss. The concept of the application is to monitor an IMAP mail account for new mail that matches a set of criteria, and applies a new mail action to the matching messages. The implementation that we will discuss will simply forward an abbreviated transcript of any new mail messages to another email address through an SMTP gateway using JavaMail. This would allow a traveling user to receive text message summaries of new email on their cell phone, for example, and this particular use case will be demonstrated.

An overview of the email forwarding application architecture is given in Figure 11-1. The source code for the application is located in the `src/main/org/jboss/chap11` directory of the book examples.

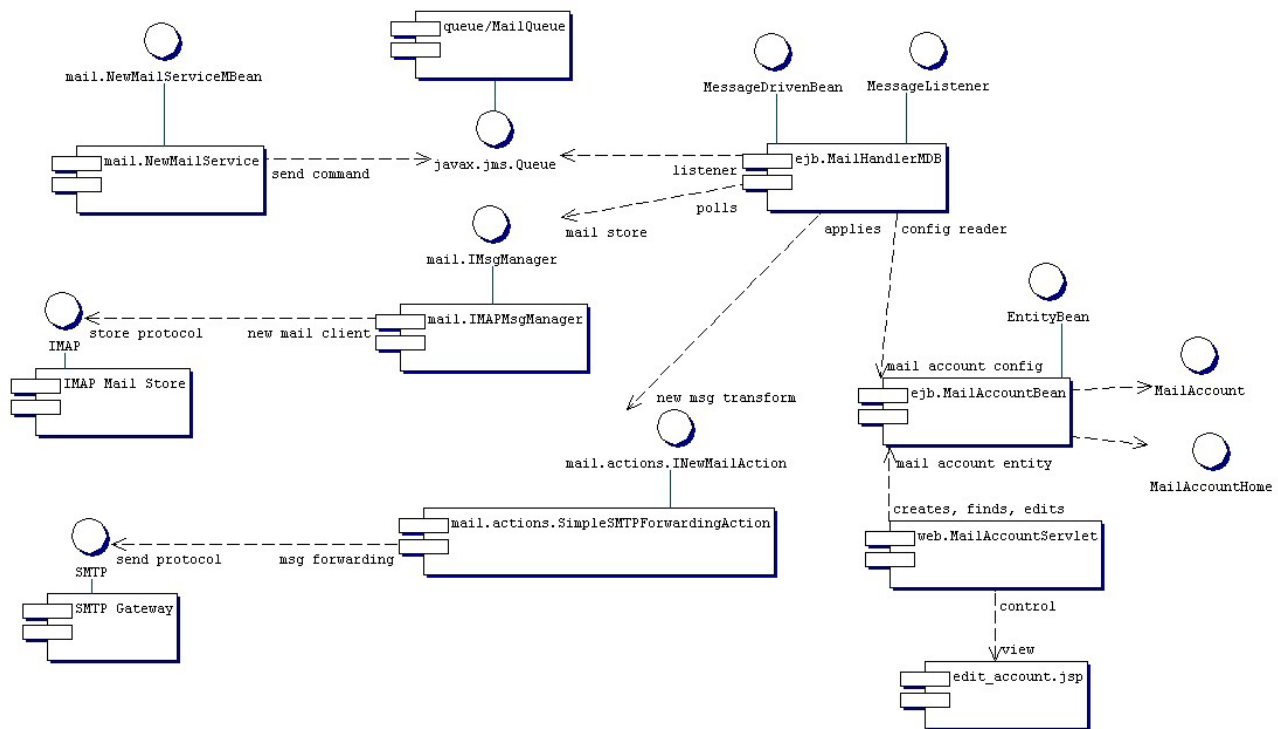


Figure 11-1, An overview of the email forwarding application architecture.

We will give a quick overview of the interactions illustrated in Figure 11.3 and then take a deeper look at the key components of the application. Starting with the NewMailService, this is a custom MBean that acts as a new mail check command generator. It sends a JMS message to the "queue/MailQueue" destination at a timed interval specified by one of its attributes. The MailHandlerMDB is the listener of the "queue/MailQueue" destination. On receipt of a new mail check command message it queries for all MailAccounts that are due for a check of their IMAP mail stores for new messages. For each MailAccount instance needing a new mail check, the MailHandlerMDB creates a javax.mail.Session and javax.mail.Store using the account information. The MailHandlerMDB then polls the mail store for new messages using an IMsgManager instance. The check for new messages includes any search criterion filter associated with the INewMailAction for the MailAccount. If new messages exist, the INewMailAction is then invoked to apply its action logic to the matching messages.

For this example application, the implementation of [INewMailAction](#) is called [SimpleSMTPFowardingAction](#). The [SimpleSMTPFowardingAction](#) takes any messages that match the search filter that has been associated with the action and creates a summary text message that includes two lines per message. The two lines are the message sender's email address and the message subject. This summary text message is then forwarded to the action forwarding email address using the action SMTP gateway server property.

The purpose of the IMsgManager abstraction is to hide the mechanism by which new messages are identified in a mail store. Different mail store protocols have varying levels of support for determining new mail messages. For example, IMAP has direct support for the notion of new messages, while POP3 does not. Therefore, an abstraction above the mail store is necessary to allow for the identification of new mail messages, and the IMsgManager interface provides this abstraction. An implementation of the IMsgManager interface access the associated mail store using the JavaMail APIs. This is combined with mail store specific logic to support identification of new mail messages. The IMsgManager interface supports limiting the set new mail messages to messages that match a search criterion. For example, one may only want to be notified of new mail from a particular user or that contains certain key words in the subject.

The management of MailAccount entity beans is the function of the MailAccountServlet. The MailAccountServlet provides a web interface that allows users to create, find and edit mail forwarding account information. It is a simple model/view/control pattern implementation where the MailAccount entity bean serves as the model data, the MailAccountServlet is the controller, and the edit_account.jsp page serves as the application view.

One question you may be asking is why use a message driven bean (MDB) instead of a stateless session bean. There are two reasons. The first is that we wanted to demonstrate the utility of message driven beans as a mechanism for loose coupling between services. A common problem users face is how to package a custom MBean service with one more EJBs from an enterprise application. The problem is that MBean services are loaded using the JBoss server application main class loader, and EJBs are loaded using a class loader that delegates to the main class loader. In order for an MBean to be a client of an EJB it must have access to the home and remote interfaces of the EJB. This means that these classes must be available to the main class loader. One can split up the EJB jar to place its client facing classes into a separate jar that is made available to the main class loader, but this can prevent redeployment of the EJBs. Sometimes this limitation is acceptable and sometimes it is not. If it is not acceptable, one way around the problem is to introduce a weakly typed coupling between the service and the EJBs. Using JMS to drive an MDB in a command pattern is one way to do this.

The second reason for using and MDB is scalability. If you were to offer a mail forwarding service based on this application you could find yourself needing to manage many client accounts. Checking for new mail is a relatively slow operation, and you would need to employ many servers to handle large number of clients. This can be done easily with JMS even in the absence of support for clustering by JBoss. One way to do this would be to configure several JBoss servers running JMS with the MailHandlerMDB deployed in each server. Each server would use a slightly different deployment descriptor for the MailHandlerMDB jar that varied in its topic message selector. Another JBoss server would

serve as the master that runs the custom NewMailService MBean. When the NewMailService awakes to send the new mail message command, it would create several JMS messages that specified a range of user accounts to poll. These messages would then be sent to a JMS topic and be distributed to the various JBoss servers where the MailHandlerMDBs are deployed. This is an example of using the JMS topic one to many message distribution model to achieve a simple clustering implementation.

The MailAccount, MailHandlerMDB and NewMailService component details

With a broad overview and rational for the type of architecture used for the mail forwarding application behind us, let's look at the three main components in more detail. We'll start with the MailAccount CMP entity bean, then discuss the MailHandlerMDB message driven bean, and finish with the NewMailService controller.

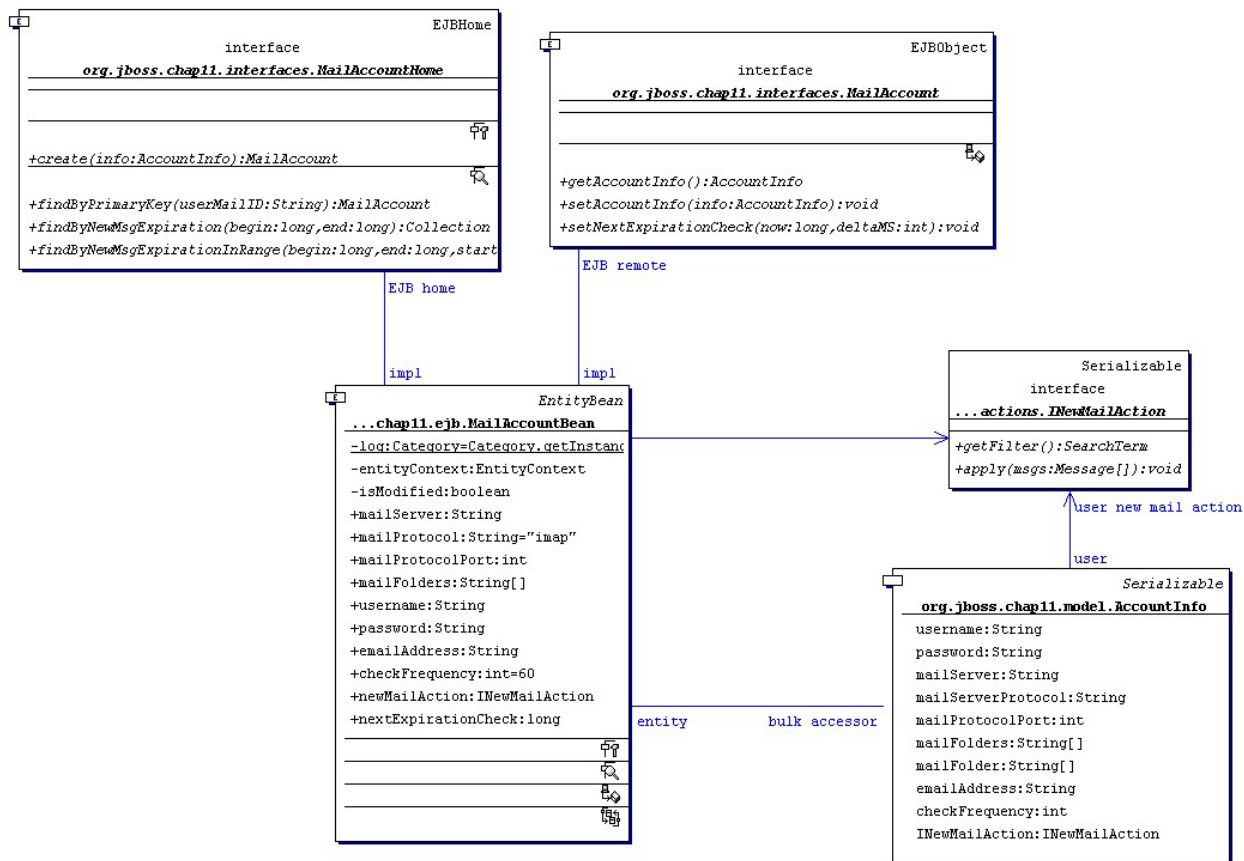


Figure 11-2, The MailAccount CMP entity bean and the AccountInfo bulk accessor representation classes.

Figure 11.4 presents a class diagram of the MailAccount CMP entity bean, its home and remote interfaces, and the AccountInfo bulk data accessor object. The MailAccount bean is a rather trivial persistent view of mail account information for a user. A simple CMP

persistence model works well here because we have no interest in a relational view of the account information. The CMP fields of the MailAccount bean are:

- **mailServer** : the account mail server where messages are stored.
- **mailProtocol** : the protocol used by the mail server. Only imap is supported in this version of the application.
- **mailProtocolPort** : the port number the mail server is listening on if not the default for the protocol.
- **mailFolders** : a list of mail folder names for the account. If null then all subscribed folders will be used.
- **username** : the user name to use when accessing the mail server.
- **password** : the password for username on the mail server.
- **emailAddress** : the email address associated with username. This is used as the from email address on forwarded messages.
- **checkFrequency** : the mail check frequency in seconds. This can only extend the interval between check for new mail. An account is checked at the maximum of the periods of the NewMailService MBean and the account specified period.
- **newMailAction** : the mail action is the action that is applied to any new mail messages. The action may include a javax.mail.search.SearchTerm that further filters new mail for arbitrary criteria. The current version of the application only supports the use of the SimpleSMTPFowardingAction described earlier.
- **nextExpirationCheck** : the time at which the account should next be checked for new messages. The units of the nextExpirationCheck field are the same as the System.currentTimeMillis method value, milliseconds, between the current time and midnight, January 1, 1970 UTC.

The only methods of the remote interface are the bulk setter and getter of the MailAccount fields in the form of the AccountInfo object, and the setNextExpirationCheck method that updates the next time at which the account should be checked for new mail. The home interface defines the required findByPrimaryKey finder that locates a MailAccount by the username field. It also defines and two custom finders that allow one to locate MailAccounts by nextExpirationCheck values in a given range as well as by nextExpirationCheck and username range. The current version of the application only uses the findByNewMsgExpiration(begin, end) form of the custom finder.

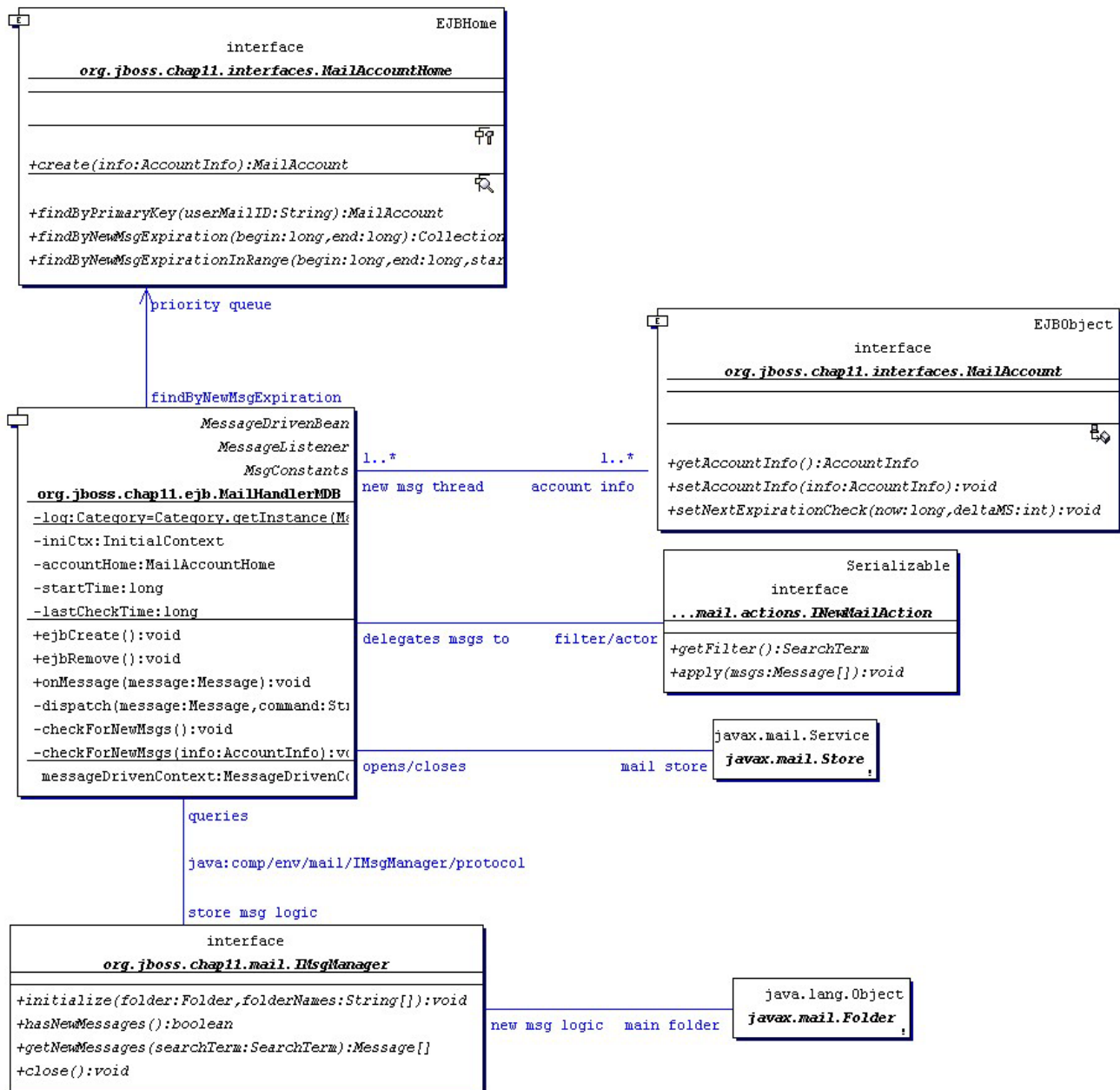


Figure 11-3, The MailHandlerMDB and associated classes.

Figure 11-3 presents the **MailHandlerMDB** class and the classes it directly interacts with. The **MailHandlerMDB** is driven by JMS messages sent to the JMS queue the MDB listens to. The **onMessage** method looks for a command property and invokes the **dispatch** method with the name of the command and the message. In the current application the only command that is handled is a request to check for new messages. The logic for this command handler is coded in the MDB as the **checkForNewMsgs** method. A more flexible

implementation of the MDB command pattern would externalize this logic using a stateless session bean to allow the command dispatch logic to be separated from the command implementation.

The checkForNewMsgs method queries for all MailAccount instances with nextExpirationCheck values that lie between the current time and the last time onMessage method was invoked by querying the MailAccountHome interface findByNewMsgExpiration finder. For all accounts meeting this criterion, the AccountInfo data is obtained and the checkForNewMsg(AccountInfo) method called to poll the associated account mail store. After opening the account mail store an IMsgManager instance for the mail store protocol is obtained by doing a lookup against the "java:comp/env/mail/IMsgManager" context with a subcontext name equal to the protocol name as specified by the account information. We'll see that the NewMailService is responsible for installing a protocol-based factory to support this. The IMsgManager instance is initialized with mail store default folder and any folder names from the account info. A lightweight (how lightweight depends on the mail store protocol) check for new messages is performed by invoking the IMsgManager.hasNewMessages method. If the mail store indicates that new messages are present, then a query for all news messages contained in the specified folders that also meet the search criterion associated with account INewMailAction is performed using the IMsgManager.getNewMessage(SearchTerm) method. Any messages returned are delegated to the account INewMailAction.apply(Message[]) method. All connections to the mail store are then closed.

On return from the checkForNewMsg(AccountInfo) method, either due to a normal completion of the method or a failure due to an exception, the MailAccount nextExpirationCheck value is updated to the next scheduled check. Note that this could mean that new messages will be missed. An attempt could be made to make this as robust as possible, but in general, in the absence of a JCA connector for the mail store the new message check is not part of the MDB JMS transaction and so, cannot be made 100% reliable. Generally this is acceptable as mail is itself a somewhat unreliable protocol that lacks guaranteed message delivery semantics.



Figure 11-4, The NewMailService MBean and associated classes.

The final major component from the application we'll discuss in detail is the NewMailService MBean. The class diagram for the NewMailService is shown in Figure 11-4. The NewMailService is responsible for driving the processing of the new mail message checks and binding the implementation of the IMsgManager interface for each supported mail store protocol into JNDI. In the current application implementation the task of driving the MailHandlerMDB processing is a trivial timer based task that uses the java.util.Timer and java.util.TimerTask classes. As discussed previously in the rationale for choosing an MDB based architecture, the logic behind this task could be considerably more complicated and entail partitioning the client accounts into ranges for distributed processing by a cluster of MailHandlerMDB deployments listening to a JMS topic.

The mechanism for providing implementations of the IMsgManager interface for each supported mail store protocol is a custom JNDI ObjectFactory binding under "java:/IMsgManager". The IMsgManagerObjectFactory provides the ObjectFactory implementation that supports this. This is a common pattern that allows a custom service to provide either a static or dynamic implementation of an interface to a J2EE component. It works well because of the standardized ENC notion supported by all J2EE components. An application assembler defines a link to a resource factory or resource environment reference

that is linked to the actual context that supports the desired object by the application deployer using a JBoss server deployment descriptor. The IMsgManagerObjectFactory is an example that allows the use of the "java:/IMsgManager" context as a dynamic factory for IMsgManager instances based on the mail store protocol. The usage construct is that one performs a lookup against the "java:/IMsgManager" context for a binding using the name of the desired protocol. When the "java:/IMsgManager" context is accessed, the IMsgManagerObjectFactory.getObjectInstance method is invoked by the JNDI framework to obtain the javax.naming.Context implementation. The implementation returned is a dynamic java.lang.reflect.Proxy that implements that Context interface using the IMsgManagerObjectFactory as the java.lang.reflect.InvocationHandler implementation. The only Context methods that are supported by the implementation are the lookup, list and toString methods. When one does a lookup against the Context implementation, the name passed to lookup is treated as the name of the mail store protocol and the appropriate IMsgManager is constructed and returned as the Context binding for the protocol name.

Hopefully this discussion on the MailAccount, MailHandlerMDB and NewMailService will give you sufficient background to tackle the application code on your own. The MailAccountServlet is a rather simple controller implementation that we'll talk a bit about when we go through a demo of the application using the web interface.

Building and assembling the mail forwarding application

The complete mail forwarding application consists of a jar containing the NewMailService MBean, and the EJB jar and web application archive combined into an EAR. To build and package the application components we need to create an appropriate Ant build file, write the deployment descriptors and define the MBean configuration. The compilation phase for all of the book examples is defined in the build.xml in the examples directory, so we'll take a look at its classpath definitions for compiling and running clients to see the scope of required JBoss jars.

Listing 11-1, the book examples build.xml file build.path and client.path classpaths demonstrating the standard requirements for compiling and running with JBoss.

```
<project name="JBossBook examples" default="build-all" basedir=".">

  <!-- Allow override from local properties file -->
  <property file=".ant.properties" />
  <!-- Override with your JBoss/Web server bundle dist root -->
  <property name="dist.root" value="G:/JBoss-2.4.5_Tomcat-4.0.3" />
  <property name="jboss.dist" value="${dist.root}/jboss"/>
  <property name="jboss.deploy.dir" value="${jboss.dist}/deploy"/>
  <!-- Change if your not using tomcat -->
  <property name="servlet.jar"
    value="${dist.root}/tomcat/lib/servlet.jar"/>
```

```

<property name="src.dir" value="${basedir}/src/main"/>
<property name="src.resources" value="${basedir}/src/resources"/>
<property name="build.dir" value="${basedir}/build"/>
<property name="build.classes.dir" value="${build.dir}/classes"/>

<path id="build.path">
  <pathelement location="${jboss.dist}/client/jboss-j2ee.jar"/>
  <pathelement location="${jboss.dist}/client/jaas.jar"/>
  <pathelement
    location="${jboss.dist}/client/jbosssx-client.jar"/>
  <pathelement location="${jboss.dist}/client/jboss-client.jar"/>
  <pathelement location="${jboss.dist}/client/jnp-client.jar"/>
  <pathelement location="${jboss.dist}/client/log4j.jar"/>
  <pathelement
    location="${jboss.dist}/client/oswego-concurrent.jar"/>
  <pathelement location="${jboss.dist}/lib/jmxri.jar"/>
  <pathelement location="${jboss.dist}/lib/ext/activation.jar"/>
  <pathelement location="${jboss.dist}/lib/ext/jboss.jar"/>
  <pathelement location="${jboss.dist}/lib/ext/mail.jar"/>
  <pathelement location="${servlet.jar}"/>
  <pathelement location="${build.classes.dir}"/>
</path>

<path id="client.path">
  <pathelement location="${jboss.dist}/client/jboss-j2ee.jar"/>
  <pathelement location="${jboss.dist}/client/jaas.jar"/>
  <pathelement
    location="${jboss.dist}/client/jbossmq-client.jar"/>
  <pathelement
    location="${jboss.dist}/client/jbosssx-client.jar"/>
  <pathelement
    location="${jboss.dist}/client/jboss-client.jar"/>
  <pathelement location="${jboss.dist}/client/jnp-client.jar"/>
  <pathelement location="${jboss.dist}/client/log4j.jar"/>
  <pathelement
    location="${jboss.dist}/client/oswego-concurrent.jar"/>
  <pathelement location="${build.classes.dir}"/>
  <pathelement location="${src.resources}"/>
</path>

...
<!-- Compile all java source under src/main -->
<target name="compile" depends="init">
  <mkdir dir="${build.classes.dir}"/>
  <javac srcdir="${src.dir}"
    destdir="${build.classes.dir}"
    classpathref="${classpath_id}"
    debug="on"
    deprecation="on"
    optimize="off"
    includes="org/jboss/**"
  />
</target>

...

```


Note that since we are compiling all of the examples in a single step this requires more jars than a standard client because the functionality of the examples covers all aspects of the standard J2EE components as well as custom JBoss extensions. You might ask why we don't just include all classes into a single jar or at least reduce the number of jars. There are a few reasons for this. In terms of the JBoss jars, the modularity of JBoss allows you to replace any of the standard frameworks with versions of your own, at least in theory. Therefore, aggregating all these framework classes into a single jar would hinder such plug-and-play replacement. For the non-JBoss jars, the packages are the original distributions from Sun or the third-party that produced them and must remain in their original form as per the distribution license.

In Listing 11-1, the book examples build.xml file build.path and client.path classpaths demonstrating the standard requirements for compiling and running with JBoss., the path element with the id="build.path" value defines the classpath used by the compile target to build all book example code. The jars included in the build.path definition are:

- client/jboss-j2ee.jar : this jar includes the standard J2EE interfaces and classes from the javax package namespace. This include the standard EJB, JMS, JCA, and JTA packages.
- client/jaas.jar : this jar includes the JAAS javax.security extension classes.
- client/jbosssx-client.jar : this jar includes the JBossSX classes required by client using the JAAS login mechanism as well as the client side SRP classes.
- client/jboss-client.jar : this jar includes all of the JBoss EJB client classes.
- client/jnp-client.jar : this jar includes the JBossNS client classes.
- client/log4j.jar : this jar includes the Apache log4j classes.
- client/oswego-concurrent.jar : this jar includes the concurrency classes from Doug Lea of "Concurrent Programming in Java" book fame.
- lib/jmxri.jar : this is the JMX reference implementation jar and contains the javax.management package classes.
- lib/ext/activation.jar : this jar includes the JavaBean activation extension package(javax.activation) classes. It is need for use with JavaMail.
- lib/ext/jboss.jar : this is the JBoss server core jar. It contains all core JBoss classes and is needed by custom services.

- **lib/ext/mail.jar** : this is the JavaMail extension package which includes the javax.mail classes.
- **servlet.jar** : this is the Servlet extension package which includes the javax.servlet classes. It is needed when compiling servlets.

Additional common jars that may be required for compilation include:

- **lib/jboss-jdbc_ext.jar** : this jar contains the JDBC 2.0 extension classes from the javax.sql package. These classes are separated from the jboss-j2ee.jar due to the restriction that JAAS 1.0 requires login modules to be on the system classpath and the `org.jboss.security.auth.spi.DatabaseServerLoginModule` uses the javax.sql classes.

So, with all these jars how do you know which ones to include? Either you learn, include them all, or start with a basic collection and when compilation errors occur due to missing classes, search the jars for the needed package. It is not really that difficult. The classpath needed to run a Java client is a subset of the build path because the build includes components that run inside of the JBoss server. The one exception to this is the `client/jbossmq-client.jar` needed by JBossMQ client applications. The reason this jar is needed at runtime but not during compilation is due to the fact that the JMS API consists almost entirely of interfaces. The classes that appear in a JMS client are all from the standard javax.jms package and so compilation only requires these standard interfaces. However, running the client requires an implementation of these interfaces and so a JMS provider specific jar is required.

After compiling the application classes we need to package them for deployment to the JBoss server. This entails creating the standard J2EE deployment descriptors and any JBoss specific deployment descriptors required to map application component references to the corresponding deployment environment binding. Listing 11-2, the `ejb-jar.xml` descriptor for the mail forwarding application enterprise beans gives the `ejb-jar.xml` descriptor for the MailAccount CMP entity bean as well as the MailHandlerMDB message driven bean.

Listing 11-2, the ejb-jar.xml descriptor for the mail forwarding application enterprise beans.

```
<?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>
    <entity>
      <ejb-name>MailAccountBean</ejb-name>
      <home>org.jboss.chap11.interfaces.MailAccountHome</home>
```

```

<remote>org.jboss.chap11.interfaces.MailAccount</remote>
<ejb-class>org.jboss.chap11.ejb.MailAccountBean</ejb-class>
<persistence-type>Container</persistence-type>
<prim-key-class>java.lang.String</prim-key-class>
<reentrant>False</reentrant>
<cmp-field>
  <field-name>mailServer</field-name>
</cmp-field>
<cmp-field>
  <field-name>mailProtocol</field-name>
</cmp-field>
<cmp-field>
  <field-name>mailProtocolPort</field-name>
</cmp-field>
<cmp-field>
  <field-name>mailFolders</field-name>
</cmp-field>
<cmp-field>
  <field-name>username</field-name>
</cmp-field>
<cmp-field>
  <field-name>password</field-name>
</cmp-field>
<cmp-field>
  <field-name>emailAddress</field-name>
</cmp-field>
<cmp-field>
  <field-name>checkFrequency</field-name>
</cmp-field>
<cmp-field>
  <field-name>newMailAction</field-name>
</cmp-field>
<cmp-field>
  <field-name>nextExpirationCheck</field-name>
</cmp-field>
<primkey-field>username</primkey-field>
</entity>

<message-driven>
  <ejb-name>MailHandlerMDB</ejb-name>
  <ejb-class>org.jboss.chap11.ejb.MailHandlerMDB</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>
  <ejb-ref>
    <ejb-ref-name>ejb/MailAccountHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>org.jboss.chap11.interfaces.MailAccountHome</home>
    <remote>org.jboss.chap11.interfaces.MailAccount</remote>
    <ejb-link>MailAccountBean</ejb-link>
  </ejb-ref>
  <resource-env-ref>

```

```

        <resource-env-ref-name>mail/IMsgManager
      </resource-env-ref-name>
      <resource-env-ref-type>org.jboss.chap11.mail.IMsgManager
    </resource-env-ref-type>
  </resource-env-ref>
</message-driven>
</enterprise-beans>

</ejb-jar>

```

The MailAccountBean entity bean definition has no external references to resource or other EJBs that require jboss.xml settings. The MailHandlerMDB message driven bean does require jboss.xml settings for both the location of the message-driven-destination and the resource-env-ref location. Note that the ejb-ref element does not require a separate deployment setting because the reference is to the MailAccountBean and so can be handled by the ejb-link element. Listing 11-3 presents the jboss.xml descriptor that provides the MailHandlerMDB deployment settings.

Listing 11-3, the jboss.xml descriptor required for specification of the MailHandlerMDB deployment settings.

```

<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>MailHandlerMDB</ejb-name>
      <destination-jndi-name>queue/MailQueue</destination-jndi-name>
      <resource-env-ref>
        <resource-env-ref-name>mail/IMsgManager</resource-env-ref-name>
        <jndi-name>java:/IMsgManager</jndi-name>
      </resource-env-ref>
    </message-driven>
  </enterprise-beans>
</jboss>

```

The destination-jndi-name element gives the JNDI name for the javax.jms.Queue destination the MailHandlerMDB will listen to for messages. Since the value is not one of the JBossMQ queue destinations found in the standard jboss.jcml configuration, we will need to add its definition. We'll look at the corresponding MBean configuration for the queue when we view the jboss.jcml configuration in the context of the NewMailService MBean. The resource-env-ref/jndi-name element gives the JNDI name to which the "java:comp/env/mail/IMsgManager" ENC binding will point to.

Although the MailAccountBean does not require a jboss.xml descriptor, it does require a jaws.xml descriptor to define the custom findByNewMsgExpiration and findByNewMsgExpirationInRange finder methods. The MailAccountBean is a CMP entity bean and its persistence and home interface methods are handled by the JBossCMP framework. The JBossCMP implementation can only handle simple finders that are based

on equality of a CMP field to the finder argument. The custom finders of the MailAccountHome interface are not of this form. Therefore, we must specify the appropriate SQL using a jaws.xml descriptor, and this is given in Listing 11-4.

Listing 11-4, the JBossCMP jaws.xml descriptor that provides the SQL statements for the custom MailAccountHome interface finders.

```
<?xml version="1.0"?>
<jaws>
  <enterprise-beans>
    <entity>
      <ejb-name>MailAccountBean</ejb-name>
<!--  findByNewMsgExpiration(long begin, long end); -->
      <finder>
        <name>findByNewMsgExpiration</name>
        <query>nextExpirationCheck &gt;= {0} AND
          nextExpirationCheck &lt;= {1}</query>
        <order>nextExpirationCheck</order>
      </finder>
<!--  findByNewMsgExpirationInRange(long begin, long end,
    String startPrefix, String endPrefix); -->
      <finder>
        <name>findByNewMsgExpirationInRange</name>
        <query>nextExpirationCheck &gt;= {0} AND
          nextExpirationCheck &lt;= {1}
          AND username &gt;= '{2}' AND username &lt;= '{3}'
        </query>
        <order>username</order>
      </finder>
    </entity>
  </enterprise-beans>
</jaws>
```

Moving to the web application components, we have the MailAccountServlet controller and its edit_account.jsp view. The web.xml descriptor is given in Listing 11-5. The only external reference is an ejb-ref to the MailAccount entity bean. Since we will bundle the web application with the entity bean jar into an EAR, we can link the ejb-ref to the MailAccountBean using the ejb-link element. Therefore, we do not need a jboss-web.xml deployment descriptor for the web application.

Listing 11-5, the web.xml descriptor for the web components of the mail forwarding application.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
  <display-name>Mail forwarding application</display-name>
  <description>Mail forwarding application</description>
```

```

<servlet>
  <servlet-name>MailAccountServlet</servlet-name>
  <servlet-class>org.jboss.chap11.web.MailAccountServlet
  </servlet-class>
</servlet>
<servlet>
  <servlet-name>MailAccountView</servlet-name>
  <jsp-file>edit_account.jsp</jsp-file>
</servlet>

<servlet-mapping>
  <servlet-name>MailAccountServlet</servlet-name>
  <url-pattern>/MailAccountServlet</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>

<ejb-ref>
  <ejb-ref-name>ejb/MailAccountHome</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>org.jboss.chap11.interfaces.MailAccountHome</home>
  <remote>org.jboss.chap11.interfaces.MailAccount</remote>
  <ejb-link>MailAccountBean</ejb-link>
</ejb-ref>
</web-app>

```

The final deployment descriptor is the `application.xml` descriptor for the mail application EAR, and this is given in Listing 11.10. This descriptor simply says that the EAR consists of the `mailer.jar` EJB jar and the `mailer.war` WAR, and that the WAR should be deployed under the root context name "mailer".

Listing 11-6, the `application.xml` descriptor for the mail application EAR.

```

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application
  PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
  'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  <display-name>Mail forwarding application</display-name>
  <description>Mail forwarding application</description>
  <module>
    <ejb>mailer.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>mailer.war</web-uri>
      <context-root>mailer</context-root>
    </web>
  </module>
</application>

```

```

    </web>
  </module>
</application>

```

The next step is to create the application deployment packages. The Ant build.xml for the mailer application is located in the src/main/org/jboss/chap11 directory of the book examples. The relevant fragments of this file that create that application EAR and MBean service jar are given in Listing 11-7.

Listing 11-7, the mail application Ant build.xml file targets for the creation of the application packages.

```

<!-- Build script for the chapter 11 mailer ear -->
<project name="Chapter 11 build" default="build-all">

  <property name="src.root" value="src/main/org/jboss/chap11" />
  <property name="chapter.dir" value="${build.dir}/chap11" />
  ...
  <target name="mailer-jar" depends="prepare">
    <jar jarfile="${chapter.dir}/mailer.jar">
      <metainf dir="${src.root}/ejb" includes="*.xml"/>
      <fileset dir="${build.classes.dir}">
        <include name="org/jboss/chap11/ejb/*" />
        <include name="org/jboss/chap11/interfaces/*" />
        <include name="org/jboss/chap11/mail/*" />
        <include name="org/jboss/chap11/model/*" />
        <exclude name="org/jboss/chap11/mail/NewMailService*" />
        <exclude name="org/jboss/chap11/mail/IMAPMsgManager*" />
        <exclude name="org/jboss/chap11/mail/IMsgManager*" />
      </fileset>
    </jar>
  </target>

  <target name="mailer-war" depends="prepare">
    <war warfile="${chapter.dir}/mailer.war"
        webxml="${src.root}/web/web.xml">
      <fileset dir="${src.root}/web">
        <include name="*.jsp" />
        <include name="*.html" />
        <include name="images/*" />
      </fileset>
      <classes dir="${build.classes.dir}">
        <include name="org/jboss/chap11/interfaces/MailAccount*" />
        <include name="org/jboss/chap11/model/AccountInfo*" />
        <include name="org/jboss/chap11/web/*" />
      </classes>
    </war>
  </target>

  <target name="mailer-ear" depends="mailer-jar,mailer-war">
    <ear earfile="${chapter.dir}/mailer.ear"
        appxml="${src.root}/application.xml">
      <fileset dir="${chapter.dir}" includes="*.jar,*.war"/>
    </ear>
  </target>

```

```

    </ear>
</target>

<target name="mailer-sar" depends="prepare">
  <jar jarfile="${chapter.dir}/mailer-service.jar">
    <fileset dir="${build.classes.dir}">
      <include name="org/jboss/chap11/mail/NewMailService*" />
      <include name="org/jboss/chap11/mail/IMAPMsgManager*" />
      <include name="org/jboss/chap11/mail/IMsgManager*" />
    </fileset>
  </jar>
</target>

</project>

```

Running the build-all target builds and deploys the application packages. Run the following command from the book examples directory to execute the build-all target:

```

examples 1975>ant -Dchap=11 build-chap
Buildfile: build.xml
...
mailer-jar:
[jar] Building jar: ...\\examples\\build\\chap11\\mailer.jar
mailer-war:
[war] Building war: ...\\examples\\build\\chap11\\mailer.war
mailer-ear:
[ear] Building ear: ...\\examples\\build\\chap11\\mailer.ear
...
mailer-sar:
[jar] Building jar: ...\\build\\chap11\\mailer-service.jar
[copy] Copying 1 file to G:\\JBoss-2.4.5_Tomcat-4.0.3\\jboss\\lib\\ext
[copy] Copying 1 file to G:\\JBoss-2.4.5_Tomcat-4.0.3\\jboss\\deploy
BUILD SUCCESSFUL

```

The final step is to configure the JBoss server to load the custom NewMailService MBean. We also need to include the definition for the MailQueue destination we declared in the jboss.xml descriptor as the destination used by the MailHandlerMDB. Listing 11-8 gives the two required jboss.jcml entries.

Listing 11-8, the jboss.jcml MBean configuration entries for the MailQueue destination and custom NewMailService.

```

<server>
...
  <!-- Configure the mailer MailQueue -->
  <mbean code="org.jboss.mq.server.QueueManager"
    name="JBossMQ:service=Queue,name=MailQueue"/>
...
  <!-- ===== -->
  <!-- Add your custom MBeans here -->

```



```

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

<!-- Configure our custom NewMailService MBean -->
<mbean code="org.jboss.chap11.mail.NewMailService"
      name=":service=NewMailService" />

</server>

```

The installation of this configuration file and the creation of a custom chap11 configuration file set can be performed using the chapter config target. Run the following command from the book examples directory to perform this step:

```

examples 1981>ant -Dchap=11 config
Buildfile: build.xml

config:

config:
[copy] Copying 14 files to ....4.3_Tomcat-4.0.3\jboss\conf\chap11
[copy] Copying 1 file to ...-2.4.5_Tomcat-4.0.3\jboss\conf\chap11
[copy] Copying 2 files to G:\JBoss-2.4.5_Tomcat-4.0.3\jboss\bin

BUILD SUCCESSFUL

```

We are now ready to run the mail forwarding application.

Testing the mail forwarding application

The previous section went through the steps required to build and deploy the mail forwarding application. It also created a custom configuration file set named chap11 to use when running the JBoss/Tomcat server bundle. To test the mail forwarding application you need to start the JBoss server using the chap11 configuration. Use one of the run_mailer.bat or run_mailer.sh start scripts that were copied to the JBoss bin directory by the configuration build step to start with the proper configuration. As an example, the following is the console output that results when running the run_mailer.bat script on a Windows 2000 system:

```

bin 1496>run_mailer.bat
JBOSS_CLASSPATH=D:/usr/local/Java/jdk1.3.1/lib/tools.jar;run.jar;
jboss.home = G:\JBoss-2.4.5_Tomcat-4.0.3\jboss
Using JAAS LoginConfig: ...omcat-3.2.3/jboss/conf/chap11/auth.conf
Using configuration "chap11"
...
[AutoDeployer] Auto deploy of ...cat-3.2.3/jboss/deploy/mailear.ear
[J2EE Deployer Default] Deploy J2EE application: ...loy/mailear.ear
[J2eeDeployer] Create application mailear.ear
[J2eeDeployer] install EJB module mailear.jar
[J2eeDeployer] inflate and install WEB module mailear.war

```

```

[J2eeDeployer] add all ejb jar files to the common classpath
[Container factory] Deploying:...oss/tmp/deploy/Default/mailer.ear
[Verifier] Verifying file:...deploy/Default/mailer.ear/ejb1001.jar
[Container factory] Deploying MailAccountBean
[Container factory] Deploying MailHandlerMDB
[JAWS] Created table 'MailAccountBean' successfully.
[Bean Cache] Cache policy scheduler started
[ContainerManagement] Initializing
[ContainerManagement] Initialized
[ContainerManagement] Starting
[ContainerManagement] Started
[ContainerManagement] Initializing
[ContainerManagement] Initialized
[ContainerManagement] Starting
[ContainerManagement] Started
[Container factory] Deployed application: ...oy/Default/mailer.ear
[J2EE Deployer Default] Starting module mailer.war
[EmbeddedTomcatServiceSX] deploy, ctxPath=/mailer,warUrl=...eb1002
[J2EE Deployer Default] J2EE application:...mailer.ear is deployed
...
[NewMailService] Starting
[NewMailService] startService, bound java:/IMsgManager
[NewMailService] Started
[Service Control] Started 40 services
[Default] JBoss 2.4.5 Started in 0m:11s

```

If the first line you see is:

```
JAVA_HOME is not defined, JSP pages may not compile correctly
```

then you need to set your JAVA_HOME environment variable to point to the root of your JDK installation. The javac compiler is needed to compile JSP pages and the start scripts add the JAVA_HOME/lib/tools.jar jar to the system classpath to support this. You can achieve this in any fashion you wish, but a javac compiler must be available to the web container.

To test the application, direct your browser to <http://localhost:8080/mailer/> and you should see the mailer application home page. This should look that the page presented in Figure 11-5.



Figure 11-5, The mail forwarding application home page.

Follow the "Create/Edit your mail forwarding account" link to go to the account management page. Figure 11-6 shows the page filled out for some fictional user.

Mail Forwarding Setup Form - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites History Print Mail Forwarding Setup

Address http://localhost:8080/mailler/edit_account.jsp Go Links »

Mail Forwarding Setup (New Account Mode)

MailServer Hostname/IP: ?

MailServer Type: ☐ POP3 ☒ IMAP ?

MailServer Username: ?

MailServer Password: ?

Mail Check Frequency: ?

Email Address: ?

SMTPServer Hostname/IP: ?

Forwarding Address: ?

Done Local intranet

Figure 11-6, The mail forwarding application account management page filled out with some fictional user information.

The account management page allows one to create a new account setup, search for an existing account setup, and edit existing account setups. Hovering your cursor over the question marks next to each field will give a brief description of the field purpose. The information for the fictional user is as follows:

- **MailServer Hostname/IP:** specifies that the mail server that stores the user mail is mail.somedot.com.

- **MailServer Type:** specifies that the type of access protocol supported by the mail server is IMAP. This is the only protocol supported by this version of the application.
- **MailServer Username:** specifies that the mail server account username is mail_user.
- **MailServer Password:** specifies the password associated with the mail_user account name.
- **Mail Check Frequency:** specifies the account level frequency for new mail checks. This will only be used if the frequency is greater than the NewMailService MBean frequency.
- **Email Address:** specifies the email address to use as the from address for mail forwarded from the mail_user account.
- **SMTPServer Hostname/IP:** specifies the SMTP gateway server through which forwarded mail should be routed is smtp.somedot.com.
- **Forwarding Address:** specifies the address to which new mail should be forwarded. Here is an example of forwarding mail as text messages to a Nextel phone account.

To demonstrate that the application does work, I entered information to forward mail from one of my accounts to my Nextel phone. An image of the first message that was forwarded to the phone is given in Figure 11-7.



Figure 11-7, An example text message forwarded from a mail account message to a cell phone.

Experiment with setting up forwarding for your account and browser the server logs to gain insight into the behavior of the application.

Securing the mail forwarding application

In this section we'll create a secured variation of the mail forwarding application. If one were offering the mail forwarding service in an application service provider (ASP) environment, only authorized users of the service would be allowed to create and edit forwarding accounts. We will develop a partial solution that only allows authorized users access to edit their existing accounts. The creation of accounts is not allowed as part of the web application itself. This is really just a pedagogical grafting of security onto the previous unsecured application, not a representative example of creating secured e-commerce applications. A properly secured application requires that security be considered as a design criterion from the start.

The only code changes required to the unsecured application are contained in the controller servlet and its JSP view. The EJBs and mailer service require no code changes. The EJB deployment descriptor does need to be updated to restrict access to their functionality. The code for the secured version of the mailer is a combination of the `org/jboss/chap11` and `org/jboss/chap11s` packages. The web content and deployments descriptors come from the `chap11s` package. The mailer EJBs and MBean service code from the unsecured `chap11` package version will be used as is.

The unsecured version of the `MailAccountServlet` took the username to use as the `MailAccount` primary key from the `edit_account.jsp` post information. In the secured version of the `MailAccountServlet`, this information is taken from the authenticated caller's identity using the JAAS `Subject` caller principal information. This is an example of an application that is running in a security context where the username and password information used to gain access to the application do not have any meaning in the application domain. Rather, the application domain notion of the caller is obtained from the authenticated subject, and this is used as the key to obtain the user account information. The creation of the ASP username and password will be handled as a separate configuration step. We will create a mail forwarding service user with a username "duke" and a password "javaman", that is assigned the caller principal value "jduke" and a role of "MailServiceUser". This process will also create a dummy `MailAccount` entity bean for the "jduke" username. Once this configuration step is performed, the only user who will have access to the `MailAccount` edit page will be the "duke" ASP user. Anyone will still be able to view the mail forwarding home page, but only "duke" will be able to traverse the edit account link.

The bulk of the changes required to secure the application occur at the deployment descriptor level. We need to declare the web forms that allow account access as secured

content, and we need to restrict what MailAccount entity bean operation can be performed based on user roles. Let's start with the changes required for the `ejb-jar.xml` descriptor. Listing 11-9 gives the revised descriptor, and the elements with bold font indicate the revisions.

Listing 11-9, the revised `ejb-jar.xml` descriptor used by the secured version of the mail forwarding application.

```
<?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>
    <entity>
      <ejb-name>MailAccountBean</ejb-name>
      <home>org.jboss.chap11.interfaces.MailAccountHome</home>
      <remote>org.jboss.chap11.interfaces.MailAccount</remote>
      <ejb-class>org.jboss.chap11.ejb.MailAccountBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field>
        <field-name>mailServer</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>mailProtocol</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>mailProtocolPort</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>mailFolders</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>username</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>password</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>emailAddress</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>checkFrequency</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>newMailAction</field-name>
      </cmp-field>
      <cmp-field>
```

```

    <field-name>nextExpirationCheck</field-name>
  </cmp-field>
  <primkey-field>username</primkey-field>
</entity>

<message-driven>
  <ejb-name>MailHandlerMDB</ejb-name>
  <ejb-class>org.jboss.chap11.ejb.MailHandlerMDB</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>
  <ejb-ref>
    <ejb-ref-name>ejb/MailAccountHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>org.jboss.chap11.interfaces.MailAccountHome</home>
    <remote>org.jboss.chap11.interfaces.MailAccount</remote>
    <ejb-link>MailAccountBean</ejb-link>
  </ejb-ref>
  <security-identity>
    <run-as>
      <role-name>InternalUser</role-name>
    </run-as>
  </security-identity>
  <resource-env-ref>
    <resource-env-ref-name>mail/IMsgManager
    </resource-env-ref-name>
    <resource-env-ref-type>org.jboss.chap11.mail.IMsgManager
    </resource-env-ref-type>
  </resource-env-ref>
</message-driven>
</enterprise-beans>

<assembly-descriptor>
  <method-permission>
    <role-name>MailServiceUser</role-name>
    <method>
      <ejb-name>MailAccountBean</ejb-name>
      <method-name>findByPrimaryKey</method-name>
    </method>
    <method>
      <ejb-name>MailAccountBean</ejb-name>
      <method-name>getAccountInfo</method-name>
    </method>
    <method>
      <ejb-name>MailAccountBean</ejb-name>
      <method-name>setAccountInfo</method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>MailServiceAdmin</role-name>
    <role-name>InternalUser</role-name>
    <method>

```



```

        <ejb-name>MailAccountBean</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>

<method-permission>
    <description>MDBs deployed in a security domain currently
    need to allow access to their onMessage because of how
    run-as is currently handled by the SecurityInterceptor.
    </description>
    <unchecked/>
    <method>
        <ejb-name>MailHandlerMDB</ejb-name>
        <method-name>onMessage</method-name>
    </method>
</method-permission>
</assembly-descriptor>
</ejb-jar>

```

All revisions are related to securing access to the MailAccountBean entity bean. We have added a number of method-permission statements that restrict access to the "MailServiceUser", "MailServiceAdmin", and "InternalUser" roles. A user with the role "MailServiceUser" is allowed to call the findByPrimaryKey, getAccountInfo and setAccountInfo methods. These are the only methods a user needs to maintain their mail forwarding account information. The "MailServiceAdmin" and "InternalUser" roles are allowed access to all MailAccountBean methods. The configuration program that creates the "duke" user account will use a login with the "MailServiceAdmin" role. The purpose of the "InternalUser" role is to allow the MailHandlerMDB access to the MailAccountBean. Since an MDB does not have any caller identity associated with its onMessage method invocation, it has no ability to invoke other J2EE components using caller identity propagation. Therefore, when an MDB interacts with secured components it either needs to explicitly establish its caller identity the same as any client does, or it can be configured to use the security-identity/run-as construct. This moves the task of setting the security identity out of the MDB code and into the hands of the application deployer where it belongs. Listing 11-9 shows that the MailHandlerMDB descriptor element has been augmented with a run-as element that says any time the MailHandlerMDB interacts with another EJB, it should be assigned the role "InternalUser". There is one caveat to using a run-as identity with an MDB that is deployed under a security domain. Because of how the run-as identity is handled at the security interceptor level, there must be an authenticated caller that has permission to invoke the MDB onMessage method. However, since JMS has no support for propagating a message producer's identity, an unauthenticated identity needs to be assigned to the security domain login module configuration. This unauthenticated identity then either needs to be assigned a role that can access the onMessage method, or the onMessage method needs to be declared as accessible by anyone. We have chosen the later approach in this example, and that is the purpose of the last method-permission element that declares the MailHandlerMDB to be unchecked.

In order to have the `ejb-jar.xml` descriptor security changes be effective, a security domain must be assigned using the `jboss.xml` descriptor. Listing 11-10 highlights the revised `jboss.xml` descriptor. The only change is the addition of the `security-domain` element to assign the security domain to be used as "secure-mailer". Recall from Chapter 8 that the value of the `security-domain` element is the JNDI location of the security manager implementation. Only the portion of the JNDI name after the "java:jaas" prefix is used as the security domain name.

Listing 11-10, the revised `jboss.xml` descriptor used by the secured version of the mail forwarding application.

```
<?xml version="1.0"?>
<jboss>
  <security-domain>java:/jaas/secure-mailer</security-domain>

  <enterprise-beans>
    <message-driven>
      <ejb-name>MailHandlerMDB</ejb-name>
      <destination-jndi-name>queue/MailQueue</destination-jndi-name>
      <resource-env-ref>
        <resource-env-ref-name>mail/IMsgManager</resource-env-ref-name>
        <jndi-name>java:/IMsgManager</jndi-name>
      </resource-env-ref>
    </message-driven>
  </enterprise-beans>
</jboss>
```

Moving to the web component changes, we'll first show the revised `web.xml` deployment descriptor and the now required `jboss-web.xml` descriptor. Listing 11-11 shows the updated `web.xml` descriptor with revisions highlighted in bold along with the new `jboss-web.xml` descriptor.

Listing 11-11, the revised `web.xml` descriptor and new `jboss-web.xml` descriptor used by the secured version of the mail forwarding application.

```
<!-- The web.xml descriptor -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
  <display-name>Mail forwarding application</display-name>
  <description>Mail forwarding application</description>

  <servlet>
    <servlet-name>MailAccountServlet</servlet-name>
    <servlet-class>org.jboss.chap11s.web.MailAccountServlet</servlet-class>
  </servlet>
```

```

<servlet>
  <servlet-name>MailAccountView</servlet-name>
  <jsp-file>edit_account.jsp</jsp-file>
</servlet>

<servlet-mapping>
  <servlet-name>MailAccountServlet</servlet-name>
  <url-pattern>/restricted/MailAccountServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>MailAccountView</servlet-name>
  <url-pattern>/restricted/edit_account</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>MailForwarding</web-resource-name>
    <url-pattern>/restricted/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>MailServiceUser</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>MailForwarding</realm-name>
</login-config>

<ejb-ref>
  <ejb-ref-name>ejb/MailAccountHome</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>org.jboss.chap11.interfaces.MailAccountHome</home>
  <remote>org.jboss.chap11.interfaces.MailAccount</remote>
  <ejb-link>MailAccountBean</ejb-link>
</ejb-ref>
</web-app>

<!-- The jboss-web.xml descriptor -->
<?xml version="1.0"?>
<jboss-web>
  <security-domain>java:/jaas/secure-mailer</security-domain>
</jboss-web>

```

The first set of changes to the web.xml descriptor move the URL location of the MailAccountServlet and MailAccountView JSP page to under the "/restricted" prefix. This prefix is identified as protected by the new security-constraint element which indicates that all content under the "/restricted" path requires an authenticated user with a role of

"MailServiceUser". The other change to the web.xml descriptor is the addition of the login-config element. This defines the mechanism by which users must present their authentication information. For this example we have chosen to use HTTP Basic authentication. The jboss-web.xml descriptor simply defines the security domain that will be used to handle authentication and authorization. Note that we have chosen to use the same "secure-mailer" security domain as was used for the EJBs. This is typical of components that are collected together into an EAR as this allows seamless interaction between the components.

The one significant web application code change was in the mechanism by which the MailAccountServlet obtains the username to use as the MailAccount entity bean primary key. The relevant code is given in Listing 11-12, and in particular, the getCallerPrincipal method is where all of the logic is centered.

Listing 11-12, the MailAccountServlet logic for obtaining the application domain identity of the authenticated caller.

```
public class MailAccountServlet extends HttpServlet
{
    ...
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        String username = getCallerPrincipal();
        if( username == null )
            throw new ServletException("Authentication is required");
    ...
        RequestDispatcher rd =
            request.getRequestDispatcher("edit_account");
        rd.forward(request, response);
    }

    /** Use the custom JBoss java:comp/env/security/subject binding
    to access the authenticated subject caller principal mapping.
    */
    private String getCallerPrincipal()
        throws ServletException
    {
        String callerPrincipal = null;
        try
        {
            InitialContext iniCtx = new InitialContext();
            String name = "java:comp/env/security/subject";
            Object ref = iniCtx.lookup(name);
            Subject subject = (Subject) ref;
            if( subject == null )
                throw new ServletException("No valid Subject found");
            Group callerPrincipalGrp = null;
            Set groups = subject.getPrincipals(Group.class);
```

```

        Iterator iter = groups.iterator();
        while( iter.hasNext() )
        {
            callerPrincipalGrp = (Group) iter.next();
            String grpName = callerPrincipalGrp.getName();
            if( grpName.equals("CallerPrincipal") )
                break;
        }
        if( callerPrincipalGrp == null )
        {
            // Use the first Principal
            Set pset = subject.getPrincipals(Principal.class);
            Principal user = (Principal) pset.iterator().next();
            callerPrincipal = user.getName();
        }
        else
        {
            // Get the sole member of the CallerPrincipal group
            Enumeration pset = callerPrincipalGrp.members();
            Principal caller = (Principal) pset.nextElement();
            callerPrincipal = caller.getName();
        }
    }
    catch(NamingException e)
    {
        throw new ServletException("Failed to find Subject", e);
    }
    return callerPrincipal;
}
}

```

The logic of the `getCallerPrincipal` method is based on the `java:comp/env/security` context that JBoss creates for every J2EE component deployed under a security domain. The security context contains the following bindings:

- **securityMgr** : this is the `org.jboss.security.AuthenticationManager` interface view of the security manager.
- **realmMapping** : this is the `org.jboss.security.RealmMapping` interface view of the security manager.
- **subject** : this is the `javax.security.auth.Subject` for the authenticated caller. This is a thread local binding that provides access to the `Subject` instance as populated by the JAAS login modules configured for the security domain. The exact information contained in the `Subject` will depend on the configured login modules, but at a minimum it will contain the information documented in Chapter 8 in the section on the JBossSX login module `Subject` usage patterns.

The `MailAccountServlet` is using only the subject binding from the security context. The `getCallerPrincipal` code looks first for the subject `CallerPrincipal Group` in order to obtain an explicit mapping of the operation environment user to the application domain. If no such mapping exists, the authenticated `Principal` is used instead. The name of the located `Principal` is used as application domain username.

Note that this is a pure JBoss construct and is not part of any J2EE specification. This level of programming is really not appropriate at the J2EE component level as it is not portable across application servers. The correct way to integrate this logic is through some reusable security framework that abstracts the application server specific details out of the component.

Building the secured mail forwarding application

Now we need to build the application packages, create an updated JBoss server configuration file set, create the user security database, and create a sample `MailAccount` bean for a prototype ASP environment user. To build and install the application EAR and MBean, run the following Ant command from the book examples directory:

```
examples 2084>ant -Dchap=11s build-chap
...
secure-mailer-war:
    [war] Building war: ...ples\build\chap11s\secure-mailer.war

secure-mailer-ear:
    [ear] Building ear: ...ples\build\chap11s\secure-mailer.ear
    [delete] Deleting: ...4.4_Tomcat-4.0.3\jboss\deploy\mailer.ear
    [copy] Copying 1 ...G:\JBoss-2.4.5_Tomcat-4.0.3\jboss\deploy

BUILD SUCCESSFUL

Total time: 12 seconds
```

This installs the `mailer-service.jar` into the JBoss `lib/ext` directory and the `secure-mailer.ear` into the JBoss `deploy` directory. Next, create the `chap11s` configuration file set by running the following Ant command from the book examples directory:

```
examples 2086>ant -Dchap=11s config
Buildfile: build.xml

config:

config:
    [copy] Copying 15 files to ...Tomcat-4.0.3\jboss\conf\chap11s
    [copy] Copying 2 files to ..._Tomcat-4.0.3\jboss\conf\chap11s
    [copy] Copying 2 files to ...oss-2.4.5_Tomcat-4.0.3\jboss\bin
```

```
BUILD SUCCESSFUL
```

```
Total time: 2 seconds
```

The creates a custom configuration file set named chap11s that contains the same customized jboss.jcml configuration used in the unsecured example, along with a new auth.conf file that defines a secure-mailer login configuration based on the DatabaseServerLoginModule. For convenience, two new starts scripts called run_secure_mailer.bat and run_secure_mailer.sh are also copied to the JBoss server bin directory. These run the JBoss/Tomcat bundle using the chap11s configuration. The final setup step to perform is the creation of the user security database and a sample account. This is done using the org.jboss.chap11s.SetupAccount class. The JBoss server must be running to use this program, so start the JBoss server using the run_security_mailer script, and once it is up, run the following Ant command:

```
examples 2088>ant -Dchap=11s -Dex=1 run-example
Buildfile: build.xml

...
init:
    [echo] Using jboss.dist=G:/JBoss-2.4.5_Tomcat-4.0.3/jboss

compile:

run-example:

run-example1:
    [java] Created Principals table, result=false
    [java] Created username=duke, password=javaman
    [java] Created username=admin, password=admin
    [java] Created Roles table, result=false
    [java] Assigned duke the role MailServiceUser
    [java] Assigned admin the role MailServiceAdmin
    [java] Assigned duke the CallerPrincipal jduke
    [java] Created LoginContext
    [java] Logged in as admin
    [java] Created account for username=jduke
    [java] Logged out
```

The SetupAccount class creates a Principals and a Roles table in the default Hypersonic database and populates the Principals with two logins, one for username "duke", and one for username "admin". These users are then assigned the roles "MailServiceUser" and "MailServiceAdmin" respectively. The "duke" user is also assigned a caller principal name of "jduke". The SetupAccount class lastly does a login as the "admin" user and creates a MailAccount bean with a primary key of "jduke". At the conclusion of the SetupAccount

program run, we have enabled the mail forwarding application for one user "duke" with a password of "javaman". Let's test access by this user.

Testing the secured mail forwarding application

To test the secured version of the mail forwarding application, direct your web browser to <http://localhost:8080/secure-mailer/>. You will see a web page similar to the unsecured version. Access to this page does not require a password. Now, click on the "Edit your mail forwarding account" link and a login dialog will be displayed. The login dialog and web page you see at this point should be similar to that shown in Figure 11-8.

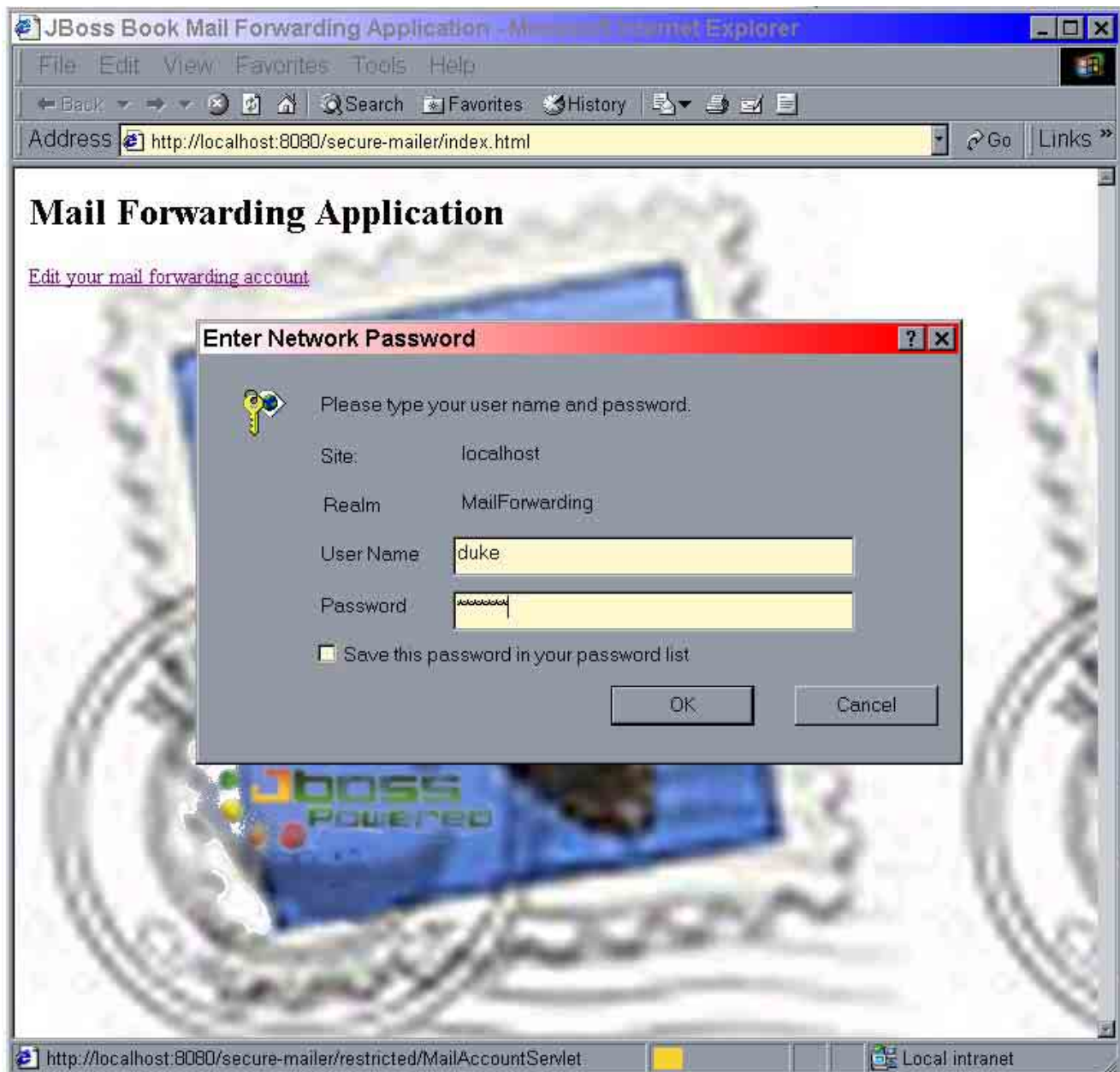


Figure 11-8, The secured mail forwarding application home page and login dialog box that is displayed after clicking on the edit account link.

Enter the username "duke" with a password of "javaman" to gain access to the mail forwarding account information setup by the SetupAccount program. You will then see the dummy account information as shown in Figure 11-9. No other user login information will gain you access to the account information page.

Mail Forwarding Setup(Edit Account Mode)

MailServer Hostname/IP: nohost.nowhere.com ?

MailServer Type: ☐ POP3 ☒ IMAP ?

MailServer Password: **** ?

Mail Check Frequency: Every 1 Minutes ?

Email Address: nouser@nohost.nowhere.com ?

SMTPServer Hostname/IP: ?

Forwarding Address: ?

Figure 11-9, The mail forwarding account information page for the mail forwarding service user "duke".

Migrating the Java Pet Store 1.1.2 Application to JBoss

In the previous section we went through the creation and JBoss configuration of a J2EE application from scratch. In this section we will look at porting an existing J2EE application that runs on the J2EE reference implementation to JBoss. The application that we will port is the J2EE Blueprints Java Pet Store.

The Java Pet Store (JPS) is a sample application from the J2EE Blueprints program. The self-description from the JPS demo states that it demonstrates how to use the capabilities of the J2EE platform to develop flexible, scalable, cross-platform e-business applications. The Java Pet Store comes with full source code and documentation, so you can experiment with J2EE technology and learn how to use it to build your own enterprise solutions. The JPS build is designed for use with the J2EE reference implementation and its Cloudscape database. In this section we will describe how to port version 1.1.2 of the JPS to the JBoss/Tomcat bundle using the default Hypersonic database. We will highlight the changes that need to be made and show the `jboss.xml` and `jboss-web.xml` that are needed, as well as the `jboss.jcml` configuration additions for the JPS resources. The complete patch is available with the book CD so you don't have to perform the work yourself. The following sections highlight the changes that were made so that you can understand the contents of the patch and gain familiarity with the general steps required to port a J2EE application to JBoss.

Patching the JPS distribution

The first step is to obtain the JPS download from the JPS 1.1.2 release home page here: http://developer.java.sun.com/developer/sampsource/petstore/petstore1_1_2.html. The JPS distribution is also available on the book CD. Once you have downloaded the bundle, unarchive it to create the directory structure shown in Figure 11-10.

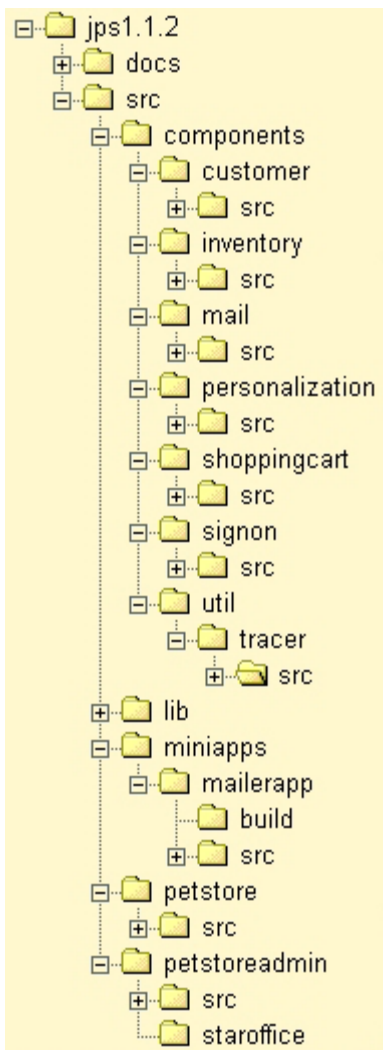


Figure 11-10, The 1.1.2 Java Pet Store application distribution directory structure.

The folders of the JPS directory structure have been expanded to show the various module `src` directories. The `src` directories contain the Ant `build.xml` scripts we will have to update to rebuild the JPS EAR using the JBoss server libraries. We will also have to create `jboss.xml` and `jboss-web.xml` deployment descriptors for the various EJB jars and web application WARs the JPS build process creates. We will place the JBoss deployment descriptors into the `src` directories along side of the JPS standard J2EE descriptors. A list of the `build.xml` files from the JPS distribution and the types of changes required to them is:

- `jps1.1.2/src/components/build.xml`, no changes required.
- `jps1.1.2/src/components/customer/src/build.xml`, update the `customer.classpath` property, add `jboss.xml` to the `customerejbjar` target.

- `jps1.1.2/src/components/inventory/src/build.xml`, update the `inventory.classpath` property.
- `jps1.1.2/src/components/mail/src/build.xml`, update the `mail.classpath` property.
- `jps1.1.2/src/components/personalization/src/build.xml`, update the `personalization.classpath` property.
- `jps1.1.2/src/components/shoppingcart/src/build.xml`, update the `shoppingcart.classpath` property.
- `jps1.1.2/src/components/signon/src/build.xml`, update the `signon.classpath` property.
- `jps1.1.2/src/components/util/tracer/src/build.xml`, update the `tracer.classpath` property.
- `jps1.1.2/src/miniapps/mailerapp/src/build.xml`, update the `j2ee.classpath`, change the ear target to work with JBoss.
- `jps1.1.2/src/petstore/src/build.xml`, copy to `jps1.1.2/src/petstore/src`, update relative paths, update the `j2ee.classpath` property, remove the `jaxp.jar` and `parser.jar` from the app, update jar and war targets to include the `jboss.xml` and `jboss-web.xml` descriptors, change the ear target to work with JBoss, remove the depends on the runtime target and change the deploy target to work with JBoss.

Creating the `jboss.xml` and `jboss-web.xml` descriptors

After updating the build files so that the JPS code base can be built without the J2EE reference implementation, we need to create the JBoss specific deployment descriptors to map the J2EE component resource references to the corresponding deployment environment JNDI binding. Listing 11-13 through Listing 11-19 give the various JBoss deployment descriptors along with a brief description of why the descriptor is required.

Listing 11-13, the `jps1.1.2/src/components/customer/src/jboss.xml` file for the `customerEjb.jar`. The `jboss.xml` descriptor is required to map the JDBC `javax.jdbc.DataSource` references to the deployment environment resource factory location.

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>TheAccount</ejb-name>
      <resource-ref>
        <res-ref-name>jdbc/EstoreDataSource</res-ref-name>
        <jndi-name>java:/EstoreDB</jndi-name>
      </resource-ref>
    </entity>
```

```

<entity>
  <ejb-name>TheOrder</ejb-name>
  <resource-ref>
    <res-ref-name>jdbc/EstoreDataSource</res-ref-name>
    <jndi-name>java:/EstoreDB</jndi-name>
  </resource-ref>
</entity>
</enterprise-beans>
</jboss>

```

Listing 11-14, the `jps1.1.2/src/components/inventory/src/jboss.xml` file for the `inventoryEjb.jar`. The `jboss.xml` descriptor is required to map the JDBC `javax.jdbc.DataSource` reference to the deployment environment resource factory location.

```

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>TheInventory</ejb-name>
      <resource-ref>
        <res-ref-name>jdbc/InventoryDataSource</res-ref-name>
        <jndi-name>java:/InventoryDB</jndi-name>
      </resource-ref>
    </entity>
  </enterprise-beans>
</jboss>

```

Listing 11-15, the `jps1.1.2/src/components/mail/src/jboss.xml` file for the `mailerEjb.jar`. The `jboss.xml` descriptor is required to map the JavaMail `javax.mail.Session` reference to the deployment environment resource factory location.

```

<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>TheMailer</ejb-name>
      <resource-ref>
        <res-ref-name>mail/MailSession</res-ref-name>
        <jndi-name>java:/MailSession</jndi-name>
      </resource-ref>
    </session>
  </enterprise-beans>
</jboss>

```

Listing 11-16, the `jps1.1.2/src/components/personalization/src/jboss.xml` file for the `personalizationEjb.jar`. The `jboss.xml` descriptor is required to map the JDBC `javax.jdbc.DataSource` reference to the deployment environment resource factory location.

```

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>TheProfileMgr</ejb-name>

```

```

    <resource-ref>
      <res-ref-name>jdbc/EstoreDataSource</res-ref-name>
      <jndi-name>java:/EstoreDB</jndi-name>
    </resource-ref>
  </entity>
</enterprise-beans>
</jboss>

```

Listing 11-17, the `jps1.1.2/src/components/shoppingcart/src/jboss.xml` file for the `shoppingcartEjb.jar`. The `jboss.xml` descriptor is required to map the JDBC `javax.jdbc.DataSource` reference to the deployment environment resource factory location.

```

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>TheCatalog</ejb-name>
      <resource-ref>
        <res-ref-name>jdbc/EstoreDataSource</res-ref-name>
        <jndi-name>java:/EstoreDB</jndi-name>
      </resource-ref>
    </entity>
  </enterprise-beans>
</jboss>

```

Listing 11-18, the `jboss.xml` file for the `jps1.1.2/src/components/signon/src/signonEjb.jar`. The `jboss.xml` descriptor is required to map the JDBC `javax.jdbc.DataSource` reference to the deployment environment resource factory location.

```

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>TheSignOn</ejb-name>
      <resource-ref>
        <res-ref-name>jdbc/SignOnDataSource</res-ref-name>
        <jndi-name>java:/EstoreDB</jndi-name>
      </resource-ref>
    </entity>
  </enterprise-beans>
</jboss>

```

Listing 11-19, the `jps1.1.2/src/petstore/src/docroot/WEB-INF/jboss-web.xml` file for the `petstore.war`. The `jboss-web.xml` descriptor is required to map the JDBC `javax.jdbc.DataSource` reference to the deployment environment resource factory location, as well as setting the locations of the EJB reference homes. Note that the EJB references could have been handled in the `web.xml` descriptor using `ejb-link` elements.

```

<jboss-web>
  <resource-ref>
    <res-ref-name>jdbc/EstoreDataSource</res-ref-name>
    <jndi-name>java:/EstoreDB</jndi-name>
  </resource-ref>
  <ejb-ref>

```

```

    <ejb-ref-name>ejb/catalog/Catalog</ejb-ref-name>
    <jndi-name>TheCatalog</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>ejb/cart/Cart</ejb-ref-name>
    <jndi-name>TheCart</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>ejb/customer/Customer</ejb-ref-name>
    <jndi-name>TheCustomer</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>ejb/profilemgr/ProfileMgr</ejb-ref-name>
    <jndi-name>TheProfileMgr</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>ejb/scc/Scs</ejb-ref-name>
    <jndi-name>TheShoppingClientController</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>ejb/inventory/Inventory</ejb-ref-name>
    <jndi-name>TheInventory</jndi-name>
  </ejb-ref>
</jboss-web>

```

Configure the Hypersonic database

The next step is the configuration of the Hypersonic database. This entailed configuring the DataSource resource factory for Hypersonic, porting the SQL database initialization script, and making code changes to the database setup servlet. Two additional source code changes were also required to the JPS BMP use of SQL constructs that were not supported by the Hypersonic database. The fact that such changes were required to change the JDBC database highlights one of the weaknesses of choosing BMP as the entity bean persistence model; namely, SQL is not completely portable. The JPS could have been made more portable by externalizing the JDBC statements from the source code to mitigate this, but the 1.1.2 version does not. Porting and testing the JPS BMP code and associated scripts was by far the largest task in the porting effort.

The first task is the creation of the DataSource resource factories for the two references used in the JPS code. If you look through the JBoss deployment descriptors you will see there are two unique JDBC DataSources: java:/EstoreDB and java:/InventoryDB. We simply created one database pool for use by both references since the tables associated with each database have no name conflicts. The database pool was given the java:/EstoreDB binding and java:/InventoryDB was created as a link to java:/EstoreDB. The jboss.jcml configuration fragment to do this is given in Listing 11-20. Note that you don't have to have to add this fragment to your jboss.jcml file as the JPS patch includes a version suitable for use with the JBoss-2.4.5/Tomcat-4.0.3 bundle. The build and deploy process we'll go through in the next

section will create a custom pestore configuration file set that includes the Listing 11-20 settings.

Listing 11-20, the jboss.jcml configuration fragment for setting up the JPS DataSource factories.

```
<mbean code="org.jboss.jdbc.XADataSourceLoader"
  name="DefaultDomain:service=XADataSource,name=EstoreDB">
  <attribute name="PoolName">EstoreDB</attribute>
  <attribute name="DataSourceClass">
    org.jboss.pool.jdbc.xa.wrapper.XADataSourceImpl
  </attribute>
  <attribute name="URL">
    jdbc:hsqldb:hsqldb://localhost:1476
  </attribute>
  <attribute name="GCMinIdleTime">1200000</attribute>
  <attribute name="JDBCUser">sa</attribute>
  <attribute name="Password" />
  <attribute name="MaxSize">10</attribute>
  <attribute name="GCEnabled">false</attribute>
  <attribute name="InvalidateOnError">false</attribute>
  <attribute name="TimestampUsed">false</attribute>
  <attribute name="Blocking">true</attribute>
  <attribute name="GCInterval">120000</attribute>
  <attribute name="IdleTimeout">1800000</attribute>
  <attribute name="IdleTimeoutEnabled">false</attribute>
  <attribute name="LoggingEnabled">false</attribute>
  <attribute name="MaxIdleTimeoutPercent">1.0</attribute>
  <attribute name="MinSize">0</attribute>
</mbean>

<mbean code="org.jboss.naming.NamingAlias"
  name="DefaultDomain:service=NamingAlias,fromName=InventoryDB">
  <attribute name="ToName">java:/EstoreDB</attribute>
  <attribute name="FromName">java:/InventoryDB</attribute>
</mbean>
```

The next task is to create the database SQL initialization script. This can be put together with a minor variation of the Oracle and Cloudscape scripts, and this is included in the JPS patch bundle as the petstore/src/docroot/WEB-INF/sql/Hypersonic.sql file.

The next task is to update the database population servlets. When you first enter the JPS store main page there is a check for the existence of the required database tables. If the tables are not found you are redirected to a database initialization page. Figure 11-11 presents the browser view you will see on following the "Enter Store" link when the JPS database tables are found to be missing.

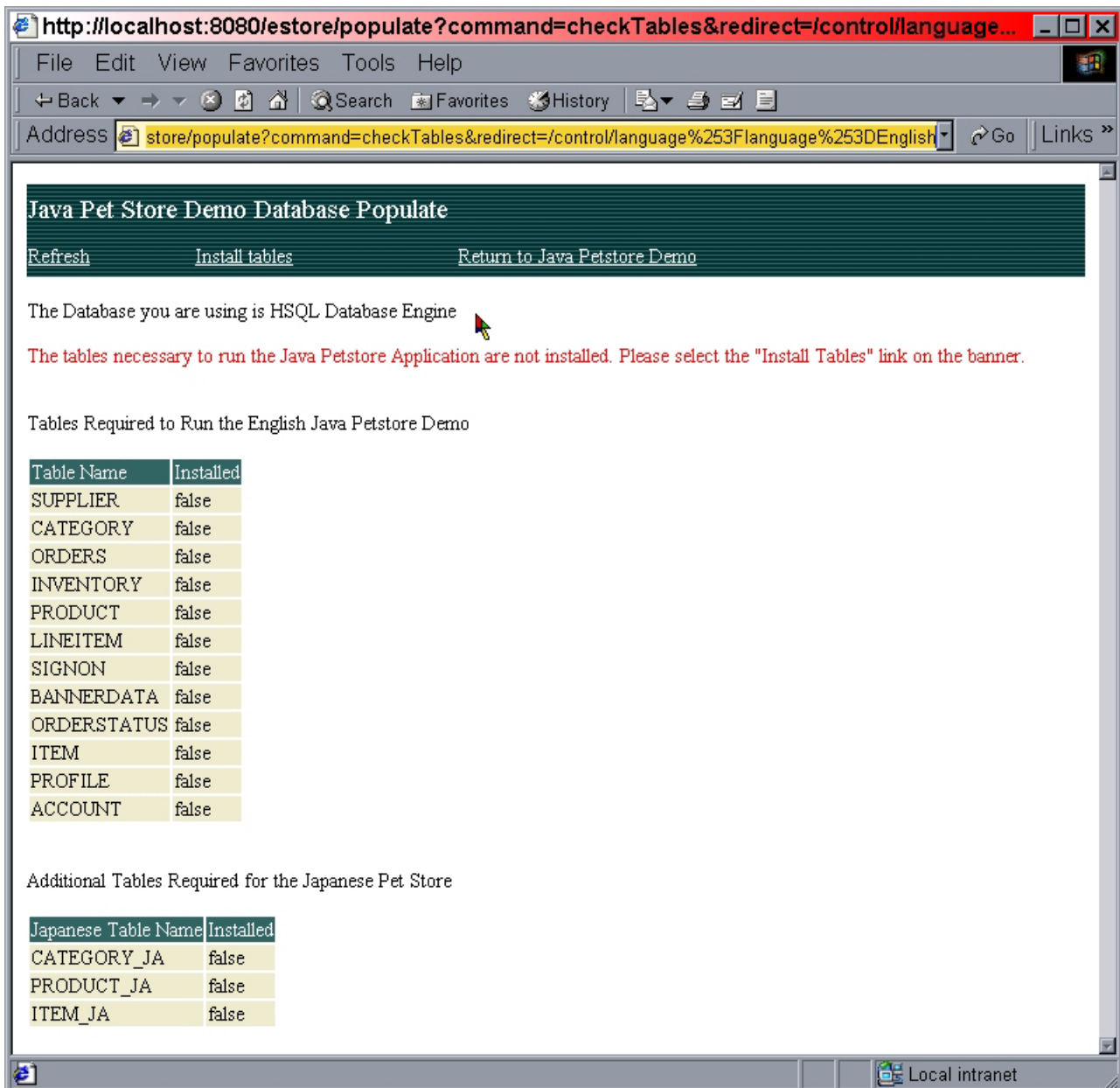


Figure 11-11, The Java Pet Store Demo Database Populate browser view you will see when the JPS database tables are found to be missing.

The page shows the database type to be "HSQL Database Engine", and lists the required tables. All are seen to be uninstalled. To install the tables click on the "Install tables" link in the page banner. This will present a page like that shown in Figure 11-12.

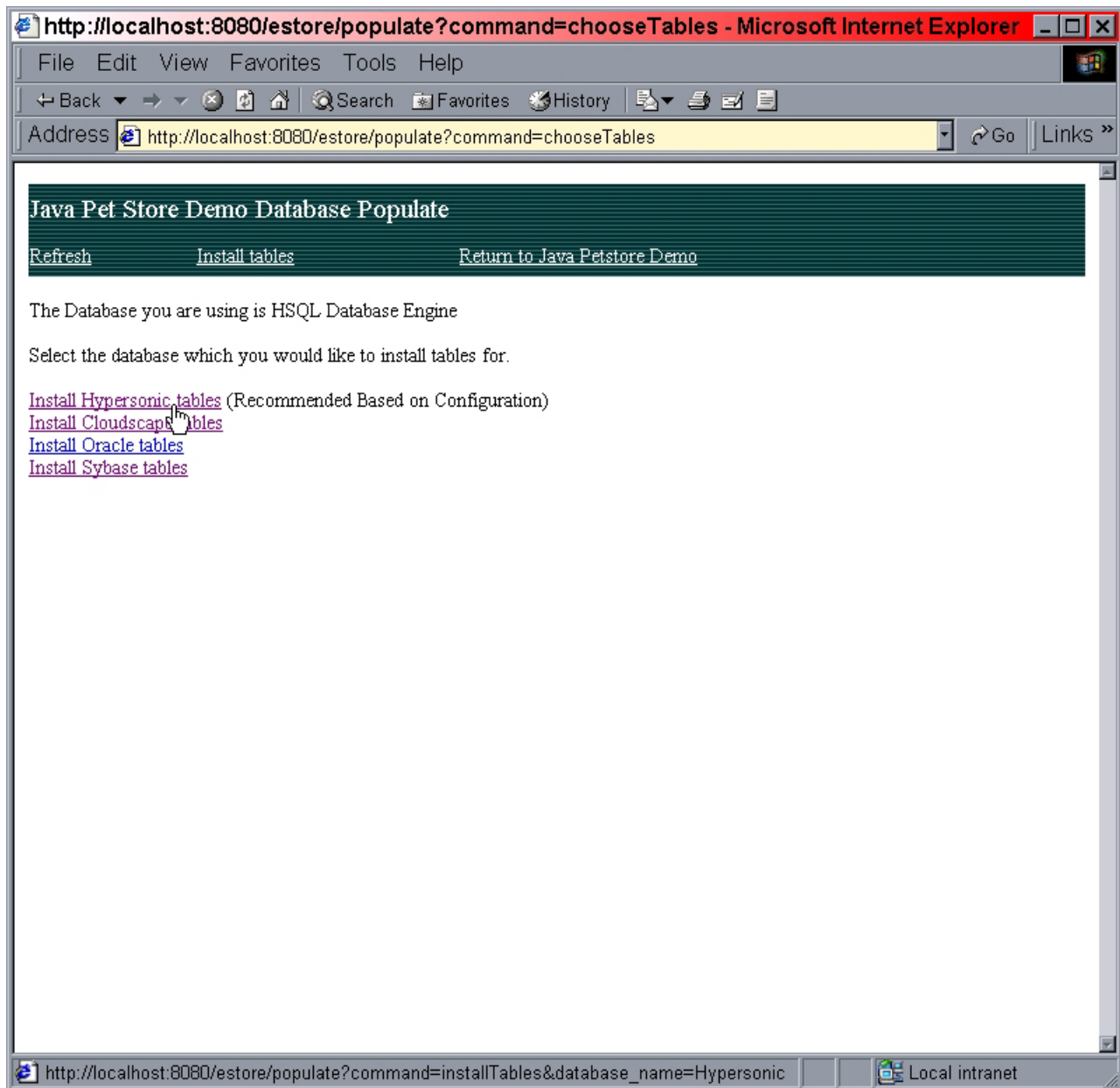


Figure 11-12, The Java Pet Store Demo Database Populate installation page view.

The recommended installation option is the "Install Hypersonic tables" link since this is what matches the configured Hypersonic database pool driver name. Clicking on this link will create and populate the JPS tables using the Hypersonic.sql script. The Figure 11-11 view is then redisplayed with the current table installation status. If any table failed to be created the page will continue to indicate that installation is required. If all tables have been installed the page will indicate that you may return to the petstore start page. Figure 11-13 shows the page view you should see after successful installation of the JPS tables into the

Hypersonic database. Return to these database initialization steps after you have built and deployed the JPS as described in the following section.

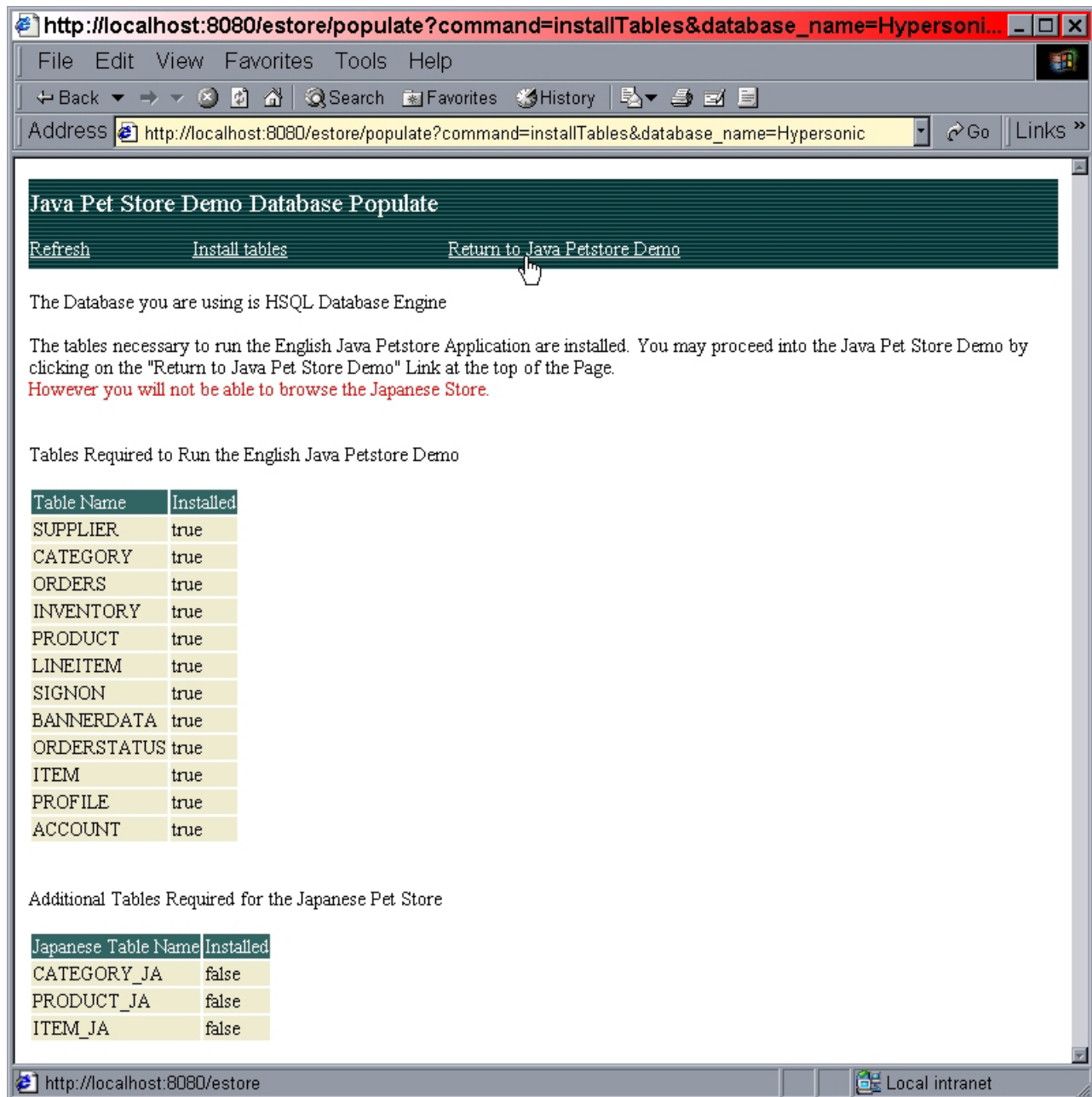


Figure 11-13, The Java Pet Store Demo Database Populate installation page view after successful installation of the JPS tables into the Hypersonic database.

Building and deploying the JPS EAR

In this section we'll walk you through the steps to apply the JPS patches and build the JPS petstore.ear. The steps for the process are:

- Start by unpacking JPS distribution to create the jps1.1.2 directory. You can use either the JDK jar tool or any platform unzip tool you may have.
- While in the parent directory of the jps1.1.2 directory, unjar or unzip the jboss-jps-patch.zip archive to overwrite the jps1.1.2 contents with the patched files.
- Go to the jps1.1.2/src directory where you will find an Ant build.xml file with a project name of "JPS Build Master". Edit the value of the jboss.bundle property to point to the location of your JBoss-2.4.5_Tomcat-4.0.3 bundle directory.
- Run the ant command without any arguments from the jps1.1.2/src directory to build the default core target. This will build all JPS component jars, wars and assemble them into the petstore.ear file in the jps1.1.2/src/petstore/build directory. Note that there will be many warnings using the 1.4.1 version of Ant that is on the book CD. Only minimal changes were made to the various JPS build.xml files to get it to build with JPS. All of the deprecated Ant constructs used by the JPS build scripts were not corrected.
- Run the ant command with "deploy" as the sole argument from the jps1.1.2/src directory to build the deploy target. This creates a petstore configuration file set in the JBoss-2.4.5_Tomcat-4.0.3/jboss/conf directory and places the petstore.ear into the 2.4.5_Tomcat-4.0.3/jboss/deploy directory.
- You are now ready to run the JPS application. Go to your 2.4.5_Tomcat-4.0.3/jboss/bin directory and you will find run_pestore.bat and run_petstore.sh start scripts. In addition to starting JBoss with the petstore configuration file set, these scripts add the JDK lib/tools.jar file to the startup classpath so that the javac compiler is available for compiling the JSP files used by JPS. This is done using the value of the JAVA_HOME environment variable. You must either set JAVA_HOME to the location of your JDK root directory or edit the start script to setup the classpath to include a javac compiler. Run the start script that is appropriate for your operating system to start the JPS application inside of JBoss.
- Enter the JPS start page by browsing to <http://localhost:8080/estore/>. When you follow the "Enter store" link for the first time you will be taken to the database initialization page. Follow the directions given in the Hypersonic configuration section to create the required JPS database tables.

Congratulations, you should now have a JPS application working under JBoss.

Using the JBossTest unit 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. Originally the unit tests were maintained as independent programs with their own scripts. The remains of this procedure can still be seen in the dist/bin directory produced by the build. However, this process is no longer maintained and it is likely the scripts no longer work.

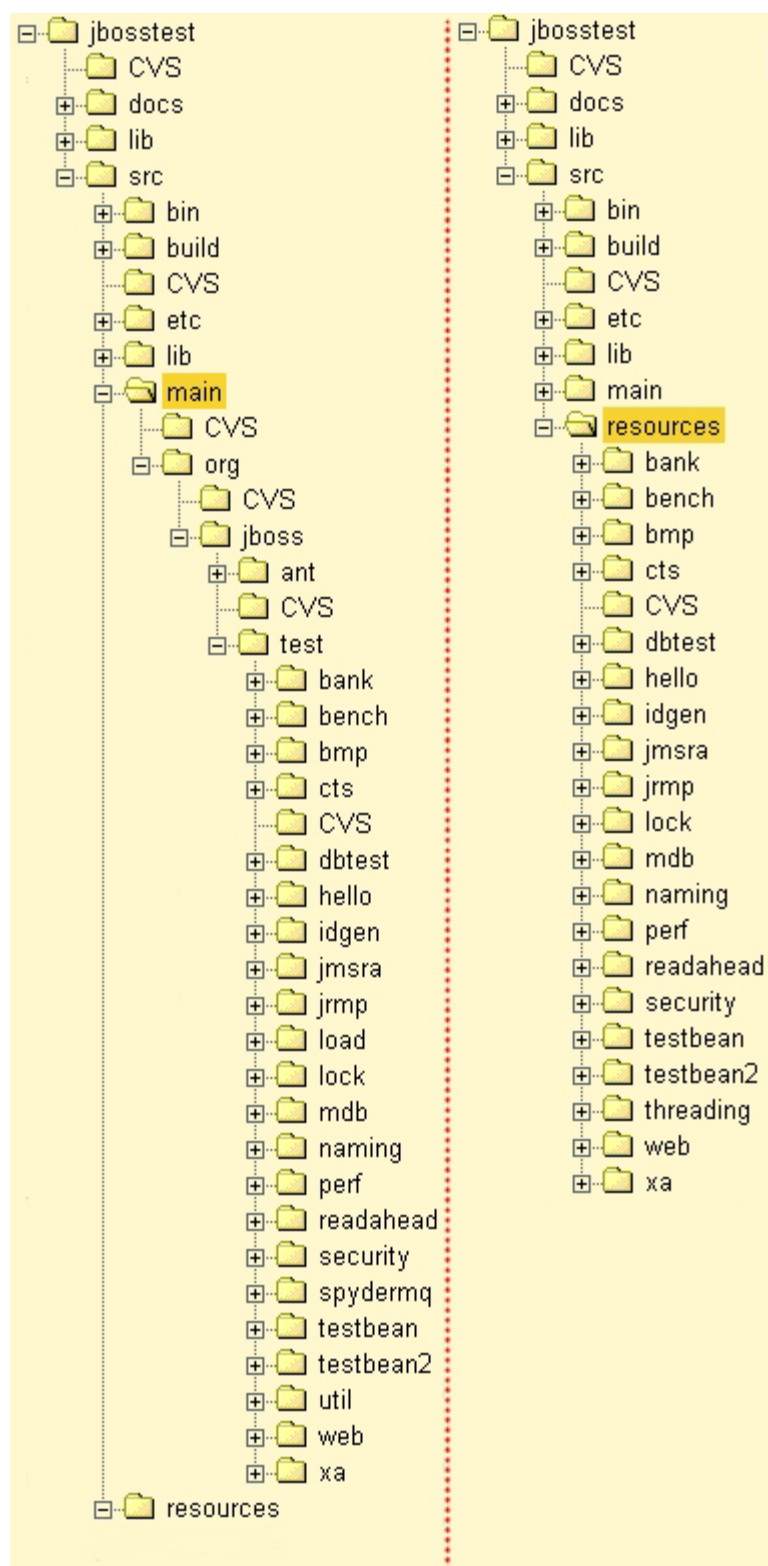


Figure 11-14, The JBossTest CVS module directory structure.

The structure of the jbosstest CVS module is illustrated in Figure 11-14. The two main branches of the source tree are shown expanded side by side. 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 as shown in the left hand side of Figure 11-14. 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. Currently there are three conventions for unit class file names: Main.java, AllJUnitTests.java, and TestXXX.java, where XXX is either the class being tests or the name of the functionality being tested. New unit tests added to the code base should use the TestXXX.java naming convention.

To run the unit tests use Ant with the src/build/run_tests.xml build file. The src/build/build.xml file is the legacy build file that creates the unit test jars. The key targets in the run_test.xml file include:

- **build** : this target builds all unit test jars using the original build.xml file. The run-tests target depends on the build target.
- **run-tests** : this is the default target of the run_tests.xml file. It runs the standard-tests, basic-security-tests and client-tests targets.
- **client-tests** : this series of tests requires that the tests be loaded from a controlled classpath rather than the build/classes directory. The reason for this is that these tests focus on issues like dynamic class loading of RMI stubs that should not come from a local classpath.
- **standard-tests** : this target consists of nearly all unit tests other than the security unit tests. Any class matching one the of patterns `**/test/*/test/Test*.java`, `**/test/*/test/AllJUnitTests.java` or `**/test/*/test/Main.java` is considered a JUnit test case.
- **basic-security-tests** : this series of tests include those tests under the org.jboss.test.security package that do not require special setup of the JBoss server.
- **run-testcase** : this target allows one to run all tests within a particular package. To run this target you need to specify a testcase property package value using - `Dtestcase=package` on the ant program 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, one would use:
ant -buildfile run_tests.xml -Dtestcase=naming run-testcase

- **test-and-report** : this target runs the run-tests target and then collates the unit text XML output files into a text summary and a nicely formatted html summary.
- **test-and-report-and-mail** : this target run the test-and-report target and then mails the text summary to the build file mail address. This is used to run the nightly unit tests and mail the results to the JBoss development mailing list.

On completion of the test-and-report target, parent directory of the jbosstest CVS working directory will contain server xml files that represent the individual junit test runs. These are collated into an html report located in the html subdirectory along with a text report located in the text subdirectory. Figure 11-15 shows an example of the html report for a run of the test suite against the JBoss 2.4.5 release on a RedHat 7.2 Linux system. The one failure listed is due to the xa unit test not having two databases available. This is a configuration issue that has not been corrected since we stopped bundling InstantDB with JBoss.

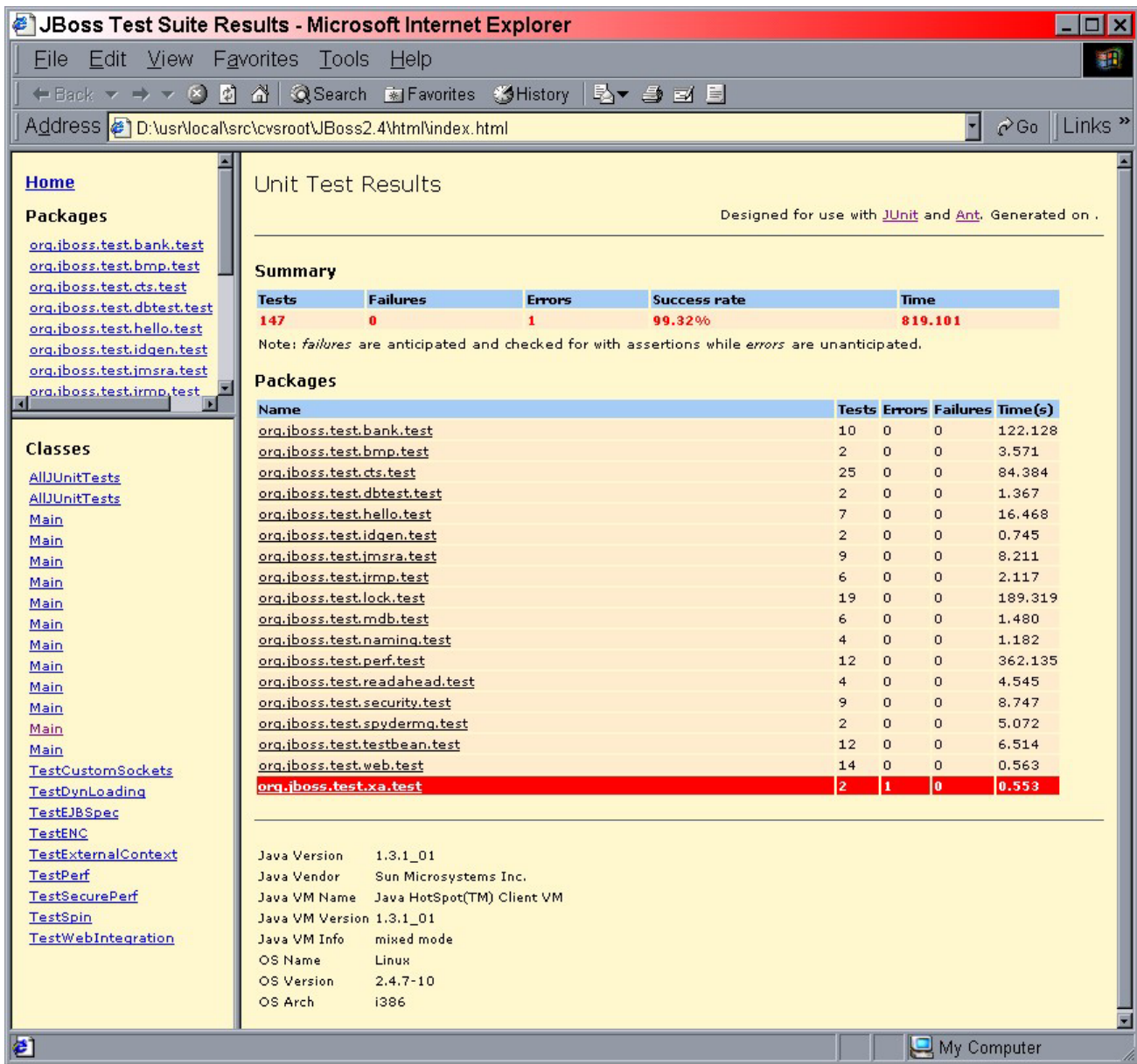


Figure 11-15, An example JBossTest test suite run report status html view as generated by the test-and-report target.



12. Appendix A

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/jbossgroup/services.jsp>.

The GNU lesser general public license (LGPL) and X license

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 with the exception of code under the `org.jboss.pool.*` package namespace, which is licensed under the X license (see <http://www.x.org/terms.htm>). Listing 12-1 gives the complete text of the LGPL license. Listing 12-2 gives the complete text of the X license.

Listing 12-1, 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 free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU 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

APPENDIX A

former contains code derived from the library, whereas the later 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,

APPENDIX A

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
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
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!

Listing 12-2, the X license text

COPYRIGHT AND PERMISSION NOTICE

```
Copyright (c) 1999,2000,2001 Compaq Computer Corporation
Copyright (c) 1999,2000,2001 Hewlett-Packard Company
Copyright (c) 1999,2000,2001 IBM Corporation
```


APPENDIX A

Copyright (c) 1999,2000,2001 Hummingbird Communications Ltd.

Copyright (c) 1999,2000,2001 Silicon Graphics, Inc.

Copyright (c) 1999,2000,2001 Sun Microsystems, Inc.

Copyright (c) 1999,2000,2001 The Open Group

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

X Window System is a trademark of The Open Group.



13. Appendix B

JBoss descriptor schema reference

In this appendix we present the DTDs for the various JBoss server deployment descriptor and configuration file schemas.

The JBoss server jboss.xml descriptor DTD

The jboss.xml descriptor provides the JBoss server specific deployment environment settings for the standard ejb-jar.xml descriptor. Listing 13-1 gives the jboss_2_4.dtd file.

Listing 13-1, the jboss_2_4.dtd file

```
<?xml version='1.0' encoding='UTF-8' ?>

<!--
This is the XML DTD for the JBoss 2.4 EJB deployment descriptor.
The DOCTYPE is:
  <!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 2.4//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_2_4.dtd">

$Id: jboss_2_4.dtd,v 1.1.2.8 2001/11/09 10:43:45 starksm Exp $
$Revision: 1.1.2.8 $
```

Overview of the architecture of jboss.xml

```
<jboss>

  <enforce-ejb-restrictions />
  <security-domain />
  <unauthenticated-principal />

  <enterprise-beans>

    <entity>
      <ejb-name />
      <jndi-name />
      <resource-ref>
        <res-ref-name />
        <resource-name />
```

```

        </resource-ref>
    </entity>

    <session>
        <ejb-name />
        <jndi-name />
        <resource-ref>
            <res-ref-name />
            <resource-name />
        </resource-ref>
    </session>

</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 />
        <instance-pool />
        <instance-cache />
        <persistence-manager />
        <transaction-manager />
        <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 AuthenticationManager 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 privileges 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? , configuration-name? ,
security-proxy? , ejb-ref* , resource-ref* , resource-env-ref*)>

<!--
    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? , configuration-name? ,
security-proxy? , ejb-ref* , resource-ref* , resource-env-ref*)>

<!--
    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 ,
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 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 none is provided, jboss
will automatically use the right standard configuration, see
container-configurations.

    Used in: entity, session, and message-driven
-->
<!ELEMENT configuration-name (#PCDATA)>

<!ELEMENT destination-jndi-name (#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 implementating 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. See
resource-managers for the actual

    Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-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 5 standard configurations with the following container-names:

- Standard CMP EntityBean
- Standard BMP EntityBean

- Standard Stateless SessionBean
- Standard Stateful SessionBean
- Standard Message Driven Bean

These standard configurations will automatically be used if no custom configuration is specified. 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? , container-interceptors? , instance-pool? ,
instance-cache? , persistence-manager? , transaction-manager? ,
locking-policy? , container-invoker-conf? , container-cache-conf?,
container-pool-conf? , commit-option? , optiond-refresh-rate? ,
(security-domain |
(role-mapping-manager , authentication-module)))>
```

<!-- The configuration-class attribute is used to indicate the implementation class that will be loaded for this configuration. This usually indicates what type of bean the configuration applies to.

-->

```
<!ATTLIST container-configuration configuration-class 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.ejb.plugins.jrmp13.server.JRMPContainerInvoker.
```

```
    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 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.EntityInstanceCache for entity beans
- org.jboss.ejb.plugins.StatefulSessionInstanceCache 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 is no longer used as the TM is assumed to be
located at java:/TransactionManager

    Used in: container-configuration
    -->
<!ELEMENT transaction-manager (#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 invoker, `JRMPContainerInvoker`.

```

    Used in: container-configuration
-->
<!ELEMENT container-invoker-conf (JMSPProviderAdapterJNDI? ,
ServerSessionPoolFactoryJNDI? , MaximumSize? , MaxMessages? ,
RMIOObjectPort? , Optimized , RMIClientSocketFactory? ,
RMIServerSocketFactory? , RMIServerSocketAddr? , ssl-domain?)>

<!-- This element is only valid if the container invoker is
JRMPContainerInvoker.
```

The `Optimized` element tells if the container invoker to bypass RMI layers when the client is local (same VM as the server). This optimizes RMI calls. Its value must be true or false.

```

    Used in: container-invoker-conf for JRMPContainerInvoker
-->
<!ELEMENT Optimized (#PCDATA)>

<!-- The RMIOObjectPort element indicates what port the RMI objects
created by this container should listen on. Any number of objects
in the same VM can use the same port. However, objects in
different VMs cannot use the same port. You may set this value
to 0 to use anyonmous ports (that is, each object just picks a
free port to use). If you want to run jBoss more than once on
the same machine, you must either create separate configurations
with separate ports, or set all the configurations to use
anonymous port. The standard jBoss setting is "4444".
```

Its value must an integer (0, or a valid port number). Note that normal user on a UNIX system cannot access privileged ports (<1024)

```

    Used in: container-invoker-conf for JRMPContainerInvoker
-->
<!ELEMENT RMIOObjectPort (#PCDATA)>

<!-- The RMIClientSocketFactory element indicates the use of a
custom socket factory that should be used by RMI objects created
by this container. The combination of socket factory type and port
must be unique but more than one container can use the same socket
factory, port combination.
```

Its value must be the fully qualified name of the class that implements the `java.rmi.server.RMIClientSocketFactory` interface, and the class must be available to the JBoss class loader. If this element is not specified the default VM client socket factory will be used.

Used in: container-invoker-conf for JRMPCContainerInvoker
-->

```
<!ELEMENT RMIClientSocketFactory (#PCDATA)>
```

<!-- The RMIServerSocketFactory element indicates the use of a custom socket factory that should be used by RMI objects created by this container. The combination of socket factory type and port must be unique but more than one container can use the same socket factory, port combination.

Its value must be the fully qualified name of the class that implements the java.rmi.server.RMIServerSocketFactory interface, and the class must be available to the JBoss class loader. If this element is not specified the default VM server socket factory will be used.

Used in: container-invoker-conf for JRMPCContainerInvoker
-->

```
<!ELEMENT RMIServerSocketFactory (#PCDATA)>
```

<!-- The RMIServerSocketAddr element specifies the address on which the RMI objects should be bound.

Its value is the interface address as a dot decimal IP address or hostname.

Used in: container-invoker-conf for JRMPCContainerInvoker
-->

```
<!ELEMENT RMIServerSocketAddr (#PCDATA)>
```

<!-- The ssl-domain element specifies the JNDI name of a org.jboss.security.SecurityDomain implementation. It is used by the custom SSL socket factory implementations.

Used in: container-invoker-conf for JRMPCContainerInvoker
-->

```
<!ELEMENT ssl-domain (#PCDATA)>
```

```
<!ELEMENT JMSProviderAdapterJNDI (#PCDATA)>
```

```
<!ELEMENT ServerSessionPoolFactoryJNDI (#PCDATA)>
```

```
<!ELEMENT MaxMessages (#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 `MaximumSize` configuration.

```
Used in: container-configuration
```

```
-->
```

```
<!ELEMENT container-pool-conf ((MaximumSize , MinimumSize) |
Synchronized)>
```

```
<!-- This element is only valid if the instance pool is a subclass
of AbstractInstancePool.
```

The `MaximumSize` element gives the maximum number of instance to keep in the pool. Its value must be an integer.

```
Used in: container-pool-conf for AbstractInstancePool subclasses
```

```
-->
```

```
<!ELEMENT MaximumSize (#PCDATA)>
```

```
<!-- This element is only valid if the instance pool is a subclass
of AbstractInstancePool.
```

The `MinimumSize` element gives the minimum number of instance to keep in the pool. Its value must be an integer.

```
Used in: container-pool-conf for AbstractInstancePool subclasses
```

```
-->
```

```
<!ELEMENT MinimumSize (#PCDATA)>
```

```
<!-- This element is only valid if the instance pool is
StatelessSessionInstancePool.
```

The `Synchronized` element instructs the the pool to synchronize

calls to the Session bean. Its value must be true or false.

```

    Used in: container-pool-conf for StatelessSessionInstancePool
-->
<!--ELEMENT Synchronized (#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 role-mapping-manager element specifies the JNDI name of
the org.jboss.security.RealmMapping implementation that is to be
used by the container SecurityInterceptor. Its use is deprecated
in favor of the security-domain element.
```

```

    Used in: container-configuration
-->
<!--ELEMENT role-mapping-manager (#PCDATA)>
```

```

<!-- The authentication-module element specifies the JNDI name of
the org.jboss.security.AuthenticationManager implementation that
is to be used by the container SecurityInterceptor. Its use is
deprecated in favor of the security-domain element.
```

```

    Used in: container-configuration
-->
<!--ELEMENT authentication-module (#PCDATA)>
```


The JBoss server jaws.xml descriptor DTD

The jaws.xml descriptor provides customization for the JBossCMP persistence engine. Listing 13-2 gives the jaws_2_4.dtd file.

Listing 13-2, the jaws_2_4.dtd file

```
<?xml version='1.0' encoding='UTF-8' ?>

<!--
This is the XML DTD for the JAWS deployment descriptor.
  <!DOCTYPE jaws PUBLIC
    "-//JBoss//DTD JAWS 2.4//EN"
    "http://www.jboss.org/j2ee/dtd/jaws_2_4.dtd">
-->
<!-- The jaws element is always the root (document) node of the
jaws.xml deployment descriptor or the standardjaws.xml defaults
document. All elements are declared as optional - if not given
in jaws.xml, defaults will be read from standardjaws.xml -->
<!ELEMENT jaws (datasource? , type-mapping? , debug? ,
default-entity? , enterprise-beans? , type-mappings?)>

<!-- the datasource element is used to indicate to JAWS which
datasource should be used for persistence of the CMP entities in
this ejb-jar. It should be the datasource named as it appears in
jboss' global naming context. The default is java:/DefaultDS

Beans are also allowed to specify datasources at bean level and
will override this datasource if specified.

Used In: jaws, entity
-->
<!ELEMENT datasource (#PCDATA)>

<!-- the type-mapping element is used to indicate to JAWS which
set of mappings from java types to jdbc and SQL types to be used
for CMP beans in this jar. type-mappings are defined within the
type-mappings element with a type-mapping element that carries a
separate meaning: This DTD will not parse! -->
<!ELEMENT type-mapping (#PCDATA)>

<!ELEMENT debug (#PCDATA)>

<!ELEMENT default-entity (create-table , remove-table ,
  tuned-updates , read-only , pk-constraint? , select-for-update?,
  time-out)>

<!ELEMENT create-table (#PCDATA)>

<!ELEMENT remove-table (#PCDATA)>

<!ELEMENT tuned-updates (#PCDATA)>
```

```

<!--ELEMENT read-only (#PCDATA)>

<!--ELEMENT pk-constraint (#PCDATA)>

<!--ELEMENT select-for-update (#PCDATA)>

<!--ELEMENT time-out (#PCDATA)>

<!-- the enterprise-beans tag contains overridden attribute
mappings for any CMP bean in this ejb-jar that requires
non-default column mapping behavior -->
<!--ELEMENT enterprise-beans (entity*)>

<!-- the entity element defines a non-default column mapping for
a CMP entity bean in this ejb-jar. This includes query
specifications for any finders that either do not correspond to a
single cmp-field or that require a specific ordering. it must
contain an ejb-name element, can contain 0 or more cmp-field
elements and may contain 0 or more finder elements.
Other options include:
- read-ahead: When a finder is called, load all data for all
entities.
- read-only: Do not persist any changes to the bean's state.
- table-name: Name of the corresponding table.
- tuned-updates: emit 'update' SQL statements that update only
changed fields.
- create-table: On deploy, create the table if it doesn't exist.
- remove-table: On undeploy, drop the table from the database
(with all_data_!!!)
- select-for-update: On loading the bean, use the
'select ... for update' syntax, locking the row.
- pk-constraint: If create-table is on, create it with a primary
key.
- time-out: For read-only only, re-load entity after time-out
-->
<!--ELEMENT entity (ejb-name , datasource? , cmp-field* , finder* ,
read-ahead? , read-only? , table-name? , tuned-updates? ,
create-table? , remove-table? , select-for-update? , time-out? ,
pk-constraint?)>

<!-- ejb-name within an entity element must contain the ejb-name
as specified in ejb-jar.xml. -->
<!--ELEMENT ejb-name (#PCDATA)>

<!--ELEMENT cmp-field (field-name , column-name ,
(jdbc-type , sql-type)?)>

<!--ELEMENT field-name (#PCDATA)>

<!--ELEMENT column-name (#PCDATA)>

<!-- the finder element overrides JAWS default behavior for a
finder, or specifies JAWS behavior for finders requiring
multi-column where clauses or a specific ordering. it must contain

```

name and query elements and my contain 1 order element.
 After JBoss version 2.3, it may contain a read-ahead element indicating whether or not all data for the entities selected should be loaded immediately. Note that JAWS/JBoss cannot guarantee serializable transactions with the read-ahead option!-->
 <!ELEMENT finder (name , query , order? , read-ahead?)>

<!-- the name within a finder element must contain the name of the finder method from the bean's home interface -->
 <!ELEMENT name (#PCDATA)>

<!-- the query element must contain the where clause that will select the proper rows to be returned by the finder. If this query begins with an inner join clause, it may specify multiple tables. -->
 <!ELEMENT query (#PCDATA)>

<!-- the order element should contain a SQL order by clause (without the initial 'order by' verb!) that should be used to order the results of the query for the finder -->
 <!ELEMENT order (#PCDATA)>

<!ELEMENT read-ahead (#PCDATA)>

<!ELEMENT table-name (#PCDATA)>

<!ELEMENT type-mappings (type-mapping-definition*)>

<!ELEMENT type-mapping-definition (name , mapping*)>

<!ELEMENT mapping (java-type , jdbc-type , sql-type)>

<!-- The java-type element specifies the fully qualified name of a Java class. This is the Java type of an entity bean cmp-field. -->
 <!ELEMENT java-type (#PCDATA)>

<!-- The jdbc-type element specifies Java class name to JDBC type mapping. The value of the jdbc-type element is the string name of the java.sql.Types constant to which the Java class should map. This is used to determine what JDBC type to use when encoding an entity bean field value into a JDBC java.sql.PreparedStatement. -->
 <!ELEMENT jdbc-type (#PCDATA)>

<!-- The sql-type element specifies the database SQL declaration for the jdbc-type. This is used when JAWS creates a table for an entity bean. -->
 <!ELEMENT sql-type (#PCDATA)>

The JBoss server jboss-web.xml descriptor DTD

The jboss-web.xml descriptor provides the JBoss server specific deployment environment settings for the standard web.xml descriptor. Listing 13-3 gives the jboss_2_4.dtd file.

Listing 13-3, the jboss-web.dtd file

```
<?xml version='1.0' encoding='UTF-8' ?>

<!-- The JBoss specific elements used to integrate the servlet
web.xml elements into a JBoss deployment.

DOCTYPE jboss-web
    PUBLIC "-//JBoss//DTD Web Application 2.2//EN"
    "http://www.jboss.org/j2ee/dtds/jboss-web.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 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 AuthenticationManager and
RealmMapping for the domain.
-->
<!ELEMENT security-domain (#PCDATA)>

<!-- The context-root element allows one to specify the servlet context path
for a stand-alone war. This serves the same function as the J2EE application.xml
context-root element used within the web module. To specify that a war should be
deployed under the path '/myweb', you would specify a context-root as follows:
    <context-root>myweb</context-root>
and the web application would be accessed using http://somehost/myweb/.
To specify that a war should be deployed as the root web application use a
context-root of
    <context-root></context-root>
and the web application would be accessed using http://somehost/
-->
<!ELEMENT context-root (#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 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-app.xml ejb-ref-name element.
-->
<!ELEMENT ejb-ref-name (#PCDATA)>

<!-- The jndi-name element specifies the JNDI name of the deployed
EJB home interface to which the servlet ENC binding will link to.
-->
<!ELEMENT jndi-name (#PCDATA)>

<!-- The resource-ref maps from the servlet ENC relative name to
the deployed JNDI name of the resource factory.
Example:
    <resource-ref>
      <res-ref-name>jms/QCF</res-ref-name>
      <jndi-name>QueueConnectionFactory</jndi-name>
    </resource-ref>
-->
<!ELEMENT resource-ref (res-ref-name , jndi-name)>

<!ELEMENT res-ref-name (#PCDATA)>

<!-- The resource-env-ref maps from the servlet ENC relative
name to the deployed JNDI name of the env resource.
Example:
<resource-env-ref>
  <resource-env-ref-name>ldap/Groups</res-ref-name>
  <jndi-name>ldap://somehost:389/ou=Group,o=somedot.com</jndi-name>
</resource-env-ref>
-->
<!ELEMENT resource-env-ref (resource-env-ref-name , jndi-name)>

<!ELEMENT resource-env-ref-name (#PCDATA)>

```

The JBoss server jboss.jcml configuration file DTD

The **jboss.jcml** configuration file is an XML document that defines the MBean services that are to be loaded into the JBoss server. The content model is defined by the **jboss_jcml.dtd**, and Listing 13-4 gives the **jboss.jcml** file DTD.

Listing 13-4, the jboss.jcml file DTD

```

<?xml version='1.0' encoding='UTF-8' ?>

<!-- The server element is the root element of the jboss.jcml

```

configuration document. It may contain one or more mbean configuration elements.

-->

<!ELEMENT server (mbean+)>

<!-- The mbean element defines a configuration for an MBean that is to be instantiated in the JBoss server. The mbean class files must be available via the JBoss thread context class loader.

-->

<!ELEMENT mbean (constructor? , attribute*, config?)>

<!-- The mbean element attributes include:

- code: the fully qualified class name of the mbean implementation.
- name: the JMX ObjectName to assign to the MBean.
- serviceFactory: the class name which implements the org.jboss.util.ServiceFactory interface. This is used to obtain an org.jboss.util.Service interface wrapper for the MBean.

-->

```
<!ATTLIST mbean  code          CDATA  #REQUIRED
                  name          CDATA  #REQUIRED
                  serviceFactory CDATA  #IMPLIED >
```

<!-- The constructor element specifies the mbean constructor signature. The child arg elements give the constructor argument type and values.

-->

<!ELEMENT constructor (arg*)>

<!-- The arg element is used to specify one constructor argument type and value.

-->

<!ELEMENT arg EMPTY>

<!-- The arg element attributes include:

- type: the fully qualified class name of the argument type as defined by the ctor signature. This defaults to java.lang.String
- value: the string representation of the argument value. If the type is not String, a string to value converter is located using the java.beans.PropertyEditorManager class.

-->

```
<!ATTLIST arg  type  CDATA  #IMPLIED
              value CDATA  #IMPLIED >
```

<!-- The attribute element specifies an attribute name and value for an mbean attribute. The content of the attribute element is the string representation of the attribute value.

-->

<!ELEMENT attribute (#PCDATA)>

<!-- The name attribute of the attribute element gives the name of the enclosing mbean attribute to set.

-->

```
<!ATTLIST attribute  name CDATA  #REQUIRED >
```

```

<!-- The config element is an optional placeholder element for
arbitrary configuration information. If the mbean supports a
importXml(org.w3c.dom.Element) method it can be supplied arbitrary
configuration data by including a config element.
-->
<!ELEMENT config ANY>

```

The JBoss server jbossmq-state.xml configuration file DTD

The jbossmq-state.xml configuration file is an XML document that is used by the StateManager MBean for simple user to password and user to durable subscription mapping. The content model is defined by the jbossmq-state.dtd, and Listing 13-5 gives the jbossmq-state.xml file DTD.

Listing 13-5, the jbossmq-state.xml file DTD

```

<?xml version='1.0' encoding='UTF-8' ?>

<!-- The StateManager element is the root element of the
jbossmq-state.xml document.
-->
<!ELEMENT StateManager (User*)>

<!-- The User element defines a JBossMQ user.
-->
<!ELEMENT User (Name , Password , Id , DurableSubscription*)>

<!-- The Name element gives the username that corresponds to the
Connection.createConnection(username, password) method, as well
as the value passed as the name parameter to the
TopicSession.createDurableSubscriber(Topic, name) method when
used in the DurableSubscription element.
-->
<!ELEMENT Name (#PCDATA)>

<!-- The Password element gives the password that corresponds to
the Connection.createConnection(username, password) method.
-->
<!ELEMENT Password (#PCDATA)>

<!-- The Id element gives the clientID that will be associated
with the connection for the username.
-->
<!ELEMENT Id (#PCDATA)>

<!-- The DurableSubscription: element is a listing of the durable
subscriptions associated with the username.
-->
<!ELEMENT DurableSubscription (Name , TopicName)>

<!-- The TopicName element gives the name of the Topic currently
associated with the durable subscription.

```

APPENDIX B

```
-->  
<!ELEMENT TopicName (#PCDATA)>
```




14. Appendix C

The Book CD Contents

In this appendix we overview the contents of the CD included with the book. The CD contents consist of the book examples source and Ant build scripts, the Ant distribution, the JBoss source and binary distributions, the Java Pet Store application and patch, and the Apache servlet container distributions. The individual files are:

- **Apache/LICENSE.txt** : The Apache Software License text.
- **Apache/jakarta-ant-1.4.1-bin.tar.gz** : The Ant 1.4.1 binary distribution as a gzipped tar archive. It is available from <http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/jakarta-ant-1.4.1-bin.tar.gz>
- **Apache/jakarta-ant-1.4.1-bin.zip** : The Ant 1.4.1 binary distribution as a zip archive. It is available from <http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/jakarta-ant-1.4.1-bin.zip>
- **Apache/jakarta-Tomcat-4.0.3.tar.gz**: The Tomcat 3.2.4 servlet container as a gzipped tar archive. It is available from <http://jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.4/bin/jakarta-Tomcat-4.0.3.tar.gz>
- **Apache/jakarta-Tomcat-4.0.3.zip** : The Tomcat 3.2.4 servlet container as a zip archive. Its available from <http://jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.4/bin/jakarta-Tomcat-4.0.3.zip>
- **Apache/jakarta-tomcat-4.0.1.tar.gz**: The Tomcat 4.0.1 servlet container as a gzipped tar archive. It is available from <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.1/bin/jakarta-tomcat-4.0.1.tar.gz>
- **Apache/jakarta-tomcat-4.0.1.zip** : The Tomcat 4.0.1 servlet container as a zip archive. Its available from <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.1/bin/jakarta-tomcat-4.0.1.zip>
- **JavaPetStore/jboss-jps-patch.zip** : The patch described in Chapter 11 that allows one to run the 1.1.2 version of the Java Pet Store application with JBoss.

- **JavaPetStore/jps-1_1_2_license.txt** : The redistribution and use license for the 1.1.2 version of the Java Pet Store.
- **JavaPetStore/jps-1_1_2.zip** : The original Java Pet Store application bundle as distributed by Sun. It is available from http://developer.java.sun.com/developer/sampsource/petstore/petstore1_1_2.html. Note that this requires a Java Developer Connection login.
- **JBoss/JBoss-2.4.4.zip** : The JBoss application server. This does not include a servlet container. Its available from <http://prdownloads.sourceforge.net/jboss/JBoss-2.4.4.zip>.
- **JBoss/JBoss-2.4.4-src.tgz**: The JBoss components source distribution. This includes the JBossServer, JBossCX, JBossMQ, JBossNS, JBossPool, JBossSX, JBossTest, contrib/tomcat, and contrib/catalina module source. It is available from <http://prdownloads.sourceforge.net/jboss/JBoss-2.4.4-src.tgz>.
- **JBoss/JBoss-2.4.4_Tomcat-4.0.3.zip** : The JBoss components source distribution. This is the integrated JBoss/Tomcat-4.0.1 servlet container bundle. Tomcat-4.0.3 is a Servlet 2.2/JSP 1.1 specification compliant servlet container. It is available from http://prdownloads.sourceforge.net/jboss/JBoss-2.4.4_Tomcat-4.0.3.zip.
- **JBoss/JBoss-2.4.4_Tomcat-4.0.1.zip** : The JBoss components source distribution. This is the integrated JBoss/Tomcat-4.0.1 servlet container bundle. Tomcat-4.0.1 is a Servlet 2.3/JSP 1.2 specification compliant servlet container. It is available from http://prdownloads.sourceforge.net/jboss/JBoss-2.4.4_Tomcat-4.0.1.zip.
- **JBoss/JBoss-2.4.4_Jetty-3.1.3.zip**: The JBoss components source distribution. This is the integrated JBoss/Jetty-3.1.3servlet container bundle. Jetty-3.1.3 is a Servlet 2.2/JSP 1.1 specification compliant servlet container. It is available from http://prdownloads.sourceforge.net/jboss/JBoss-2.4.4_Jetty-3.1.3.zip.
- **Example/examples.zip** : The book example source code bundle. It contains the chapter source code and Ant build scripts. This is the example code and structure described in the book.



15. Appendix D, Tools and Examples

Two tools used throughout the book examples and by JBoss developers are Ant and Log4j. This appendix provides an introduction to their usage. This appendix also takes you through the installation of the book example source code, and gives an overview of each example.

Using Ant

Ant is a Java- and XML-based build tool that is now the standard, used pervasively throughout the Java community. Ant is used extensively in JBoss for building the server and its modules, running the unit tests, and building and running examples. Throughout the book, you use Ant for building and running the book examples. This section provides a quick introduction to help you get Ant installed and comfortable with its basic operation.

Ant version 1.4.1 is used in this book, as this is the version bundled with JBoss 2.4.4. It's the version available on the book CD, and you can always download the latest version of Ant from its home page on the Apache/Jakarta Web site at <http://jakarta.apache.org/ant/>. The Ant home page also includes links to numerous Ant related resources, such as documentation and articles on Ant.

The first step to using Ant is to install it. A step-by-step installation of Ant and its optional jars help to make this as easy as possible. The installation procedure is as follows:

- 1. Unarchive the jakarta-ant-1.4.1-bin.zip or jakarta-ant-1.4.1-bin.tar.gz distribution bundle into a directory of your choice. This will create the jakarta-ant-1.4.1 directory, ANT_HOME.**
- 2. Add the ANT_HOME bin directory to your operating system path. This enables you to run Ant from a command shell by typing ant.... The bin directory contains a number of shell script wrappers for Unix, Win32, Perl, and Python.**

3. You can optionally set the ANT_HOME environment variable to the full path to the jakarta-ant-1.4.1 directory. On some operating systems the ant wrapper scripts in the bin directory can guess ANT_HOME (Unix dialects and Windows NT/2000), so you can try using Ant without doing this if you are using one of these platforms.
4. You must also have a JAXP-compliant XML parser installed and available on your classpath. The binary distribution of Ant includes the latest version of the Apache Crimson XML parser in the ANT_HOME lib subdirectory. If you want to use a different JAXP-compliant parser, you should remove jaxp.jar and crimson.jar from Ant's lib directory. You can then put the jars from your preferred parser into Ant's lib directory, or put the jars on the system classpath.
5. Set the JAVA_HOME environment variable to the directory where your JDK is installed. When you need JDK functionality (such as for the javac task or the rmic task), the tools.jar must be added to the Ant classpath. The scripts supplied with Ant in the bin directory will add the required JDK classes automatically if the JAVA_HOME environment variable is set.
6. Install the Ant optional tasks jar named jakarta-ant-1.4.1-optional.jar into the ANT_HOME lib subdirectory. The jakarta-ant-1.4.1-optional.jar is available from the Ant binaries distribution page.
7. Optionally install the JUnit jar into the ANT_HOME lib subdirectory. The JUnit jar is required to run the JBossTest unit tests. If you are not interested in running the JBossTest unit tests, you can skip this step.
8. Optionally install the Xalan XSLT processor jar into the ANT_HOME lib subdirectory. An XSLT processor is required for the Ant style task that is used by the JBossTest report generation step. If you are not interested in running the JBossTest unit tests, you can skip this step.

The Xalan processor can be obtained from the book CD or the Xalan home page at <http://xml.apache.org/xalan-j/index.html>.

Now test the Ant installation by opening a shell or command prompt and then create a build.xml file in the current directory that contains the contents given in Listing 15-1.

Listing 15-1, An Ant build.xml script for testing the Ant installation.

```

<!-- Simple Ant build script to test an Ant installation -->
<project name="TestInstall" default="run" basedir=".">

    <property name="hello.class" value="ASimpleHelloObject" />

    <target name="init">
        <available file="${hello.class}.java"
            property="hello.src.exists"/>
    </target>

    <target name="ASimpleHelloObject" unless="hello.src.exists"
        depends="init">
        <echo file="${hello.class}.java">
public class ${hello.class}
{
    public static void main(String[] args)
    {
        System.out.println("${hello.class}.main was called");
    }
}
        </echo>
        <echo message="Wrote ${hello.class}.java" />
    </target>

    <target name="compile" depends="ASimpleHelloObject">
        <javac destdir="." srcdir="." debug="on" classpath=".">
            <include name="ASimpleHelloObject.java"/>
        </javac>
    </target>

    <target name="run" depends="compile">
        <java classname="${hello.class}" classpath="." />
        <echo message="Ant appears to be correctly installed" />
    </target>

</project>

```

When you run Ant on a linux system with the build.xml file in the /tmp directory you should see output like:

```

bash-2.04$ ant
Buildfile: build.xml

init:

ASimpleHelloObject:
    [echo] Wrote ASimpleHelloObject.java

compile:
    [javac] Compiling 1 source file to /tmp

run:
ASimpleHelloObject.main was called

```

```
[echo] Ant appears to be successfully installed
BUILD SUCCESSFUL
Total time: 2 seconds
```

On a Win32 system with the build.xml file in D:/temp you should see output like:

```
D:\temp>ant
Buildfile: build.xml

init:

ASimpleHelloObject:

compile:
[javac] Compiling 1 source file to D:\temp

run:
ASimpleHelloObject.main was called
[echo] Ant appears to be successfully installed

BUILD SUCCESSFUL

Total time: 2 seconds
```

At this point you have Ant installed. This is really all you need for running the book examples, the JBossTest code and building the JBoss server. The next step is getting to the point that you can read an Ant build file and follow the gist what it is trying to do.

An Ant build file is an XML document that consists of seven types of elements: project, properties, targets, tasks, pattern sets, file sets, and path structures. Most of these are illustrated in Listing 15-1. The following numbered lines extracted from Listing 15-1 are annotated to highlight the key Ant element types.

```
2:<project name="TestInstall" default="run" basedir=".">
```

Line 2 is the Ant project element. The root element of every Ant build file is a project element. A project element simply encloses the other types of Ant elements and associates a name (name attribute), the default target name to execute (default attribute), and the base directory (basedir attribute). The base directory is used to resolve non-absolute paths that occur elsewhere in the build file.

```
4: <property name="hello.class" value="ASimpleHelloObject" />
```

Line 4 is an example of an Ant property element. A property element assigns a property variable a string value. The scope of the property value is global from the point of its definition. A property element may be nested inside of a target element so that the property will only be defined if the containing target is executed. Note that all Java system properties are also automatically available to Ant. To obtain the value of a property you enclose its name in “\${}”. For example, several places in Listing 15-1 demonstrate accessing the `hello.class` property value using “\${hello.class}” and three examples are on lines 7, 13, and 22:

```
7: <available file="${hello.class}.java"
13: <echo file="${hello.class}.java">
22: <echo message="Wrote ${hello.class}.java" />
```

A property value may also be specified on the Ant command line using the same syntax one uses to pass system properties to the java program; that is, `-Dproperty=value`.

```
6: <target name="init">
7: <available file="${hello.class}.java"
8: property="hello.src.exists"/>
9: </target>
```

Lines 6 through 9 give an example of an Ant target element. A target element is a container of all other Ant elements except for the project element. It allows you to group tasks together and assign the grouping a name.

```
11: <target name="ASimpleHelloObject" unless="hello.src.exists"
12: depends="init">
13: <echo file="${hello.class}.java">
14: public class ${hello.class}
15: {
16:     public static void main(String[] args)
17:     {
18:         System.out.println("${hello.class}.main was called");
19:     }
20: }
21: </echo>
22: <echo message="Wrote ${hello.class}.java" />
23: </target>
```

Lines 11 through 23 give another example of a target element that illustrates that targets can be conditional on the existence of a property and that targets can depend on other targets. The `unless="hello.src.exists"` attribute specifies that the target named “ASimpleHelloObject” will be executed only if the `hello.src.exists` property has been set. The `depends="init"` attribute indicates that the “init” target should be executed prior to the “ASimpleHelloObject” target. A target may depend on multiple targets by specifying the names of the prerequisite targets in the attribute value. The names must be separated by commas.

```

26: <javac destdir="." srcdir="." debug="on" classpath=".">
27: <include name="ASimpleHelloObject.java"/>
28: </javac>

```

Lines 26 through 28 give an example of a task named `javac`. The `javac` task executes the Java compiler on a set of source files. Generally, a task is simply a piece of Java code that is to be executed. Ant comes with numerous standard tasks, many more optional tasks, and has a simple mechanism that allows you to create and include custom tasks. Tasks can be passed any number of attributes. Ant validates that a task does, in fact, support the setting of the specified attributes so you must know which attributes a task supports by consulting the task documentation.

The `include` element on line 27 is an example of a file set specification. Here an explicit file is given. In general a simple regular expression facility is supported that allows for wildcards such as `*.java` to specify all java source files.

Of course, to really understand an Ant build file, you need to know the range of Ant tasks and their syntax. You have to obtain this by reading the task documentation as well as by usage experience. Take a look at the ant command syntax. Ant itself will tell you this by passing the `-help` argument to the following ant command:

```

bin 1461>ant -help
ant [options] [target [target2 [target3] ...]]
Options:
  -help                print this message
  -projecthelp          print project help information
  -version              print the version information and exit
  -quiet                be extra quiet
  -verbose              be extra verbose
  -debug                print debugging information
  -emacs                produce logging information without adornments
  -logfile <file>      use given file for log
  -logger <classname>  the class which is to perform logging
  -listener <classname> add an instance of class as a project listener
  -buildfile <file>    use given buildfile
  -D<property>=<value> use value for given property
  -find <file>         search for buildfile towards the root of the
                      filesystem and use it

```

The key options include `buildfile`, `projecthelp`, and `D`. By default the ant command will look for a `build.xml` file in the current directory, but you can use any XML document. You specify the name of an alternate build file to ant using the `buildfile` argument. The `projecthelp` argument displays a summary of all targets in a build file along with any description attributes for the targets. The `D` option is used to specify the value of properties at runtime to Ant.

Using the log4j framework in JBoss

Logging of messages is a common requirement in all applications. In a server environment it is a critical feature due to the distributed multi-user interaction that is characteristic of a server. Many users interact simultaneously with an application server and some degree of logging of the interactions is essential for support. A unique aspect of an application server is that many different developers may have contributed code to the applications that comprise the active components. The logging requirement could vary significantly between the various components or applications. What is needed is a flexible logging API that supports these use cases. The JBoss server has standardized on log4j as its logging API. The switch to log4j has been a gradual one, and as of the 2.4.4 release, log4j is the only logging API used internally by JBoss.

Although there are many logging APIs, including the JSR47 logging framework that is bundled with the current JDK 1.4 release, the log4j API appears to be the most commonly used of all available. It is designed to be fast, flexible, and simple. These are probably the most important criteria for an application server logging framework. So what is the log4j API?

Log4j has four fundamental objects: categories, priorities, appenders and layouts. Of these, API users directly use only categories and maybe priorities. Together these components allow developers to log messages according to message type and priority, and to control at runtime how these messages are formatted and where they are reported. We will cover the basics of log4j to allow you to understand the JBoss log4j configuration and help get you started using log4j in your components. For additional documentation refer to the log4j home page, which is located here: <http://jakarta.apache.org/log4j/>.

The org.apache.log4j.Category class

The central component in the log4j API is the org.apache.log4j.Category class. A category is a named entity and its name is a case-sensitive, hierarchical construct whose naming hierarchy adheres to the following rule, which is taken from the log4j manual:

A category is said to be an ancestor of another category if its name followed by a dot is a prefix of the descendant category name. A category is said to be a parent of a child category if there are no ancestors between itself and the descendant category.

This is the same convention as the Java package namespace. There exists a special root category that simply is, but has no name. It is accessed via a static method of the Category class. The Category class itself contains a large number of methods, but only the factory, logging and priority state methods are of general interest. A summary of the Category class restricted to these methods is summarized in Listing 15-2.

Listing 15-2, a summary of the key methods in the log4j Category class.

```
public class Category
{
    public static Category getRoot()
    public static Category getInstance(Class clazz)
    public static Category getInstance(String name)
    ...
    public void debug(Object msg)
    public void debug(Object msg, Throwable t)
    public boolean isDebugEnabled()
    public void info(Object msg)
    public void info(Object msg, Throwable t)
    public boolean isInfoEnabled()
    ...
    public boolean isEnabledFor(Priority priority)
    public void log(Priority priority, Object msg)
    public void log(Priority priority, Object msg, Throwable t)
}
```

Before going through the methods we need to define the Priority class that shows up here. The org.apache.log4j.Priority object represents the importance or level of a message. Whenever you log a message it has a Priority associated with it. There are a small number of Priorities defined by default and are know by the names: FATAL, ERROR, WARN, INFO and DEBUG. You can extend the set of known priorities by providing subclasses of the Priority class. The utility of assigning a priority to a message is that it allows one to filter messages based on their priority or importance. Further, you can test to see if a given priority has been enabled for a Category to avoid generating log messages that would have no affect due to the current priority filters. This is important for high frequency debugging messages whose volume can adversely impact the server. Priority objects have both a string name and an integer value. The name is simply a mnemonic label for the priority. The integer value defines a relative order amongst priorities. This allows one to enable or disable all priorities below a given threshold.

The getRoot method is an accessor for the anonymous root of the default category hierarchy. The getInstance method is a factory method which returns the unique Category instance associated with the given name. If the category does not exist it will be created. The version that accepts a Class simply calls getInstance(clazz.getName()).

The debug, isDebugEnabled, info, and isInfoEnabled methods are convenience methods that invoke the corresponding log or isEnabledFor method with the Priority that corresponds the priority associated with the convenience method. For example, debug(Object) simply invokes log(Priority.DEBUG, Object).

The isEnabledFor(Priority) method checks to see if the Category will accept a message of the indicated Priority. The log(Priority, Object) and log(Priority, Object, Throwable) pass the

message onto the appenders associated with the Category provided that the messages pass the current Priority filter.

The JBoss `org.jboss.log.Logger` wrapper

The JBoss server framework actually uses a simple wrapper around the `log4j Category`. This wrapper adds support for a custom TRACE level priority and removes the unused Category methods. This does not interfere with the `log4j Category` usage in any way. The `Logger` class simply provides a collection of explicit log priority convenience methods as well as a factory method as show in Listing 15-3.

Listing 15-3, The JBoss Logger class summary.

```
package org.jboss.logging;

import org.apache.log4j.Category;
import org.apache.log4j.Priority;

public class Logger
{
    private Category log;

    public static Logger getLogger(String name)
    public static Logger getLogger(Class clazz)

    public Category getCategory()

    public boolean isTraceEnabled()
    public void trace(Object message)
    public void trace(Object message, Throwable t)

    public boolean isDebugEnabled()
    public void debug(Object message)
    public void debug(Object message, Throwable t)

    public boolean isInfoEnabled()
    public void info(Object message)
    public void info(Object message, Throwable t)

    public void warn(Object message)
    public void warn(Object message, Throwable t)

    public void error(Object message)
    public void error(Object message, Throwable t)

    public void fatal(Object message)
    public void fatal(Object message, Throwable t)

    public void log(Priority p, Object message)
    public void log(Priority p, Object message, Throwable t)
}
```

Not only does this provide direct support for the TRACE level priority used internally by the JBoss server for high-frequency messages that should not normally be displayed, it also avoids the problem of introducing a custom Category factory. In previous versions of JBoss, support for the TRACE priority was done using a custom subclass of Category that added the trace support methods. The problem with the custom subclass is that it tended to result in integration problems like ClassCastException errors with custom user services.

You are free to use the JBoss Logger class if you want to take advantage of the TRACE level priority feature. If you are writing custom MBeans or other services that extend from JBoss classes it is likely that you inherit a Logger instance for use. If you are writing applications that should remain independent of the JBoss classes, then use of the JBoss Logger class should be avoided in place of the standard log4j Category.

The org.apache.log4j.Appender interface

The ability to selectively enable or disable logging requests based on their category is only a request to log a message. The appenders associated with the category that receives the log message handle the actual rendering of the log message. An appender is a logical message destination. An appender delegates the task of rendering log messages into strings to the layout instance assigned to the appender. There can be multiple appenders attached to a category, which means that a given message can be sent to multiple destinations. All appenders must implement the org.apache.log4j.Appender interface. This interface imposes the notions of layouts as well as filters and error handlers. A number of appenders are bundled with the log4j framework, including appenders for consoles, files, GUI components, remote socket servers, JMS, Windows event loggers, and remote UNIX syslog daemons. Appenders also exist which allow the rendering of messages to occur asynchronously.

The org.apache.log4j.Layout class

The rendering of a log message into a string representation is delegated to instances of the org.apache.log4j.Layout class. A Layout is a formatter that transforms an org.apache.log4j.spi.LoggingEvent object into a string representation. A Layout can also specify the content type of the string as well as header and footer strings.

Configuring log4j using org.apache.log4j.PropertyConfigurator

That is really all you need to know to use the log4j API to perform logging from components. One large detail missing so far is how to configure log4j. This entails setting the category priorities as well as configuration of the appenders associated with categories. The log4j framework provides support for programmatic configuration as well as configuration using XML and Java properties files. We'll discuss the Java properties file configuration method as this is what JBoss uses in its standard configuration.

The Java properties file based configuration of log4j is handled by the org.apache.log4j.PropertyConfigurator class. The PropertyConfigurator class reads the configuration information for category priority thresholds, appender definitions, and category to appender mappings from a Java properties file. The properties file can be changed at runtime to modify the active log4j configuration. We'll learn the basic syntax of the PropertyConfigurator properties file by discussing the standard JBoss log4j.properties file given in Listing 15-4.

Listing 15-4, the standard JBoss log4j.properties configuration file.

```
# A default log4j properties file suitable for JBoss

### Appender Settings ###
### The server.log file appender
log4j.appender.Default=org.apache.log4j.FileAppender
log4j.appender.Default.File=../log/server.log
log4j.appender.Default.layout=org.apache.log4j.PatternLayout
# Use the default JBoss format
log4j.appender.Default.layout.ConversionPattern=[%c{1}] %m%n
# Truncate if it already exists.
log4j.appender.Default.Append=false

### The console appender
log4j.appender.Console=org.jboss.logging.log4j.ConsoleAppender
log4j.appender.Console.Threshold=INFO
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=[%c{1}] %m%n

### Category Settings ###
log4j.rootCategory=DEBUG, Default, Console

# Example of only showing INFO msgs for any categories under
# org.jboss.util
#log4j.category.org.jboss.util=INFO

# An example of enabling the custom TRACE level priority that is
# used by the JBoss internals to diagnose low level details. This
# example turns on TRACE level msgs for the org.jboss.ejb.plugins
# package and its subpackages. This will produce A LOT of logging
# output.
#log4j.category.org.jboss.ejb.plugins=TRACE#org.jboss.logging.TracePriority
```

The first thing to note is that property names in the file are compound names whose components are separated by periods. This is a common pattern used in property files to group properties together. There are really only two classes of properties being defined in Listing 15-4, appenders (prefix = log4j.appender) and categories (prefix = log4j.category, log4j.rootCategory is a special case for the default root category).

The first section of the file (### Appender Settings ###) defines the log4j appender configuration. Property names that begin with the "log4j.appender" prefix specify properties that apply to Appender instances. The first component in the property name after the log4j.appender prefix is the name of the appender. Thus, the first appender configuration is for the appender named "Default". The log4j.appender.Default property defines the type of appender implementation to use. In this case, the org.apache.log4j.FileAppender is specified. The FileAppender implementation represents a file destination. All properties with the log4j.appender.Default prefix define properties on the Default appender instance. The set of properties one can specify for a given appender depend on the appender type. For the FileAppender the name of the log file, the Layout instance to use, and whether existing log files should be appended to are allowed properties. The log4j.appender.Default.layout.ConversionPattern property is setting the ConversionPattern property value for the log4j.appender.Default.layout property of the FileAppender. The type of the Layout instance was specified to be org.apache.log4j.PatternLayout by the log4j.appender.Default.layout property. Refer to the log4j javadocs for the complete syntax of the format string the PatternLayout class supports.

The log4j.appender.Console properties configure a second appender named Console. This appender sends its output to the System.out and System.error streams of the console in which JBoss is run. One feature common to most appenders, and illustrated by the Console appender configuration, is the ability to filter out log events whose priority is below some threshold. The log4j.appender.Console.Threshold=INFO setting says that only events with priorities greater than or equal to INFO should be handled by an appender. All other messages should simply be ignored.

The second section of the file (###Category Settings ###) defines the appender to category mappings as well as the category priority thresholds. The root category specification of threshold priority and associated appenders is a special case of the log4j.category grouping of properties, which has the following syntax:

```
log4j.rootCategory=[priority] [(, appenderName)*]
```

So, the log4j.rootCategory entry in Listing 11.4 states that the root category priority threshold is set to DEBUG, and its appenders are Default and Console. The general syntax for the category setting is:

```
log4j.category.category_name=[priority] [(, appenderName)*]
```

There are two commented out examples of the general form. The first states that the org.jboss.util category and its subcategories should filter all messages below the INFO priority level. The second, states that the org.jboss.ejb.plugins category should filter all messages below the custom TRACE#org.jboss.logging.TracePriority priority level. Since

log4j does not know which class provides the custom priority implementation, the class must be specified using the "#classname" suffix added to the name of the priority.

The XML based org.apache.log4j.xml.DOMConfigurator configuration class offers more flexibility and the benefits, and drawbacks, of an XML based configuration. As we'll describe in the Log4jService MBean configuration section, JBoss supports both the properties file and XML version of the log4j configuration files. For reference, the DTD for the configuration documents supported by the DOMConfigurator is given in Figure 15-1.

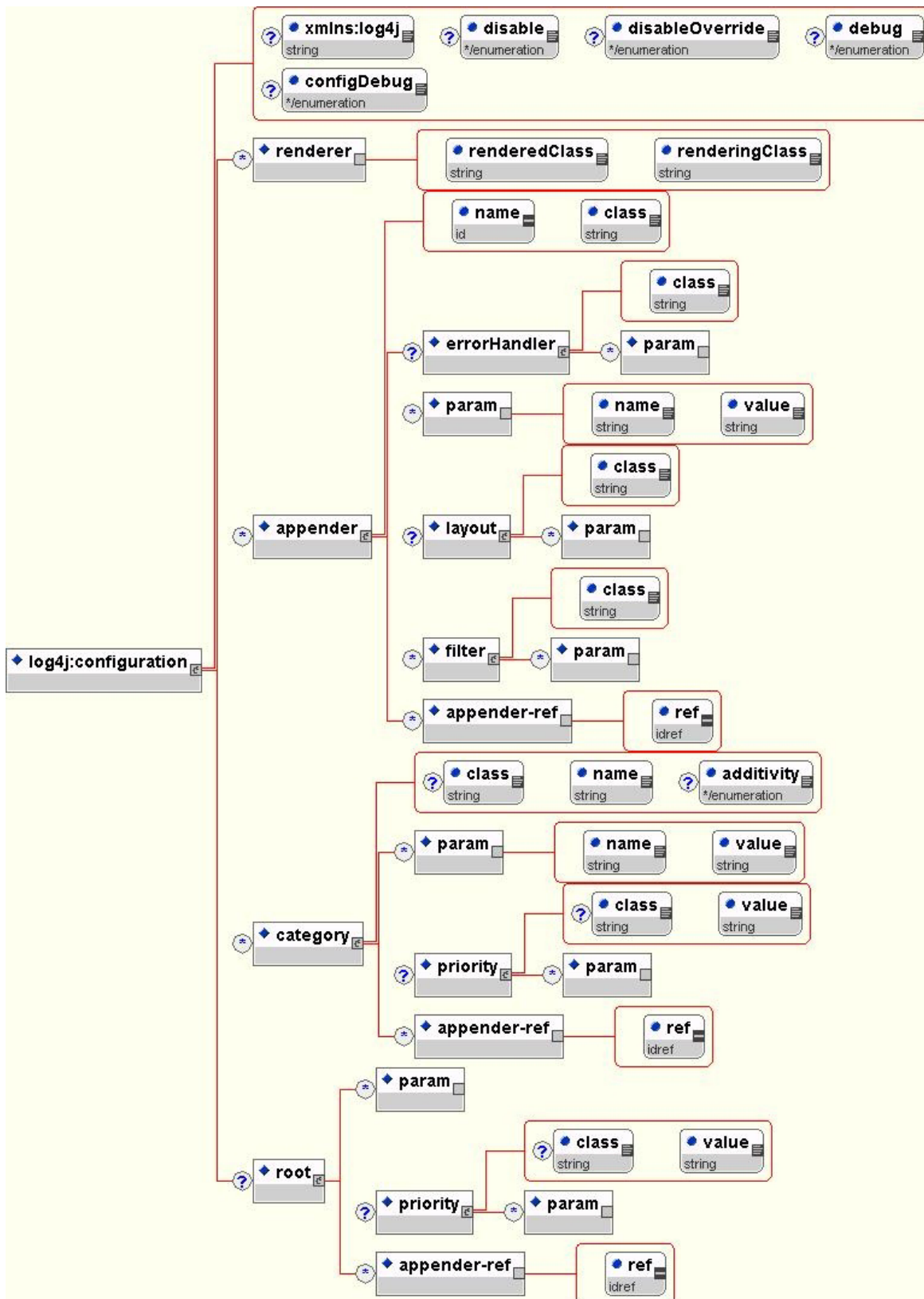


Figure 15-1, The DTD for the configuration documents supported by the log4j version 1.1.3 DOMConfigurator.

Log4j usage patterns

The two biggest usage questions regarding log4j from a developer's perspective are what category names to use, and what message priorities should be used. The pattern used in JBoss is based on the class name of the component performing the logging. In many cases this is the category name used. If there are multiple instances of a component, and it is associated with another meaningful name, this name will be added as a subcategory to the component class name. For example, the `org.jboss.security.plugins.JaasSecurityManager` class uses a base category name equal to its class name. There can be multiple `JaasSecurityManager` instances, and each is associated with a security domain name. Therefore, the complete log4j category name used by the `JaasSecurityManager` is "org.jboss.security.plugins.JaasSecurityManager.securityDomain", where the securityDomain value is the name of the associated security domain.

For message logging priorities, the JBoss usage policy is the following:

- **TRACE**, use the TRACE level priority for log messages that are directly associated with activity that corresponds requests. Further, such messages should not be submitted to a `Logger` unless the `Logger` category priority threshold indicates that the message will be rendered. Use the `Logger.isTraceEnabled()` method to determine if the category priority threshold is enabled. The point of the TRACE priority is to allow for deep probing of the JBoss server behavior when necessary. When the TRACE level priority is enabled, you can expect the number of messages in the JBoss server log to grow at in proportion to N, where N is the number of requests received by the server. The server log may well grow as power of N depending on the request-handling layer being traced.
- **DEBUG**, use the DEBUG level priority for log messages that convey extra information regarding service life-cycle events. Developer or in depth information required for support is the basis for this priority. The important point is that when the DEBUG level priority is enabled, the JBoss server log should not grow proportionally with the number of server requests. Looking at the DEBUG and INFO messages for a given service category should tell you exactly what state the service is in, as well as what server resources it is using: ports, interfaces, log files, etc.
- **INFO**, use the INFO level priority for service life-cycle events and other crucial related information. Looking at the INFO messages for a given service category should tell you exactly what state the service is in.

- **WARN**, use the **WARN** level priority for events that may indicate a non-critical service error. Resumable errors, or minor breaches in request expectations fall into this category. The distinction between **WARN** and **ERROR** may be hard to discern and so its up to the developer to judge. The simplest criterion is would this failure result in a user support call. If it would use **ERROR**. If it would not use **WARN**.
- **ERROR**, use the **ERROR** level priority for events that indicate a disruption in a request or the ability to service a request. A service should have some capacity to continue to service requests in the presence of **ERRORs**.
- **FATAL**, use the **FATAL** level priority for events that indicate a critical failure of a service. If a service issues a **FATAL** error it is completely unable to service requests of any kind.

This usage policy may indirectly affect your choice of priorities if you log events to the JBoss server appenders. If you do, then you would want to adhere to the above usage policy or you would lose your ability to effectively filter messages in a consistent manner across categories. If you introduce your own appenders for your own category namespace, you are free to choose any priority policy you want as filtering can be done independent from the JBoss categories.

The Log4jService MBean revisited

Recall from Chapter 2 that the Log4jService MBean configures the Apache log4j system, which JBoss uses as its internal logging API. The Log4jService can use either a Java properties style configuration file, or an XML configuration file. The choice between the two is determined solely on the basis of the configuration file name. If the configuration file ends in ".xml", the XML configuration is assumed and the org.apache.log4j.xml.DOMConfigurator class is used. If this is not the case, the configuration file is assumed to be in the Java properties format and the org.apache.log4j.PropertyConfigurator class is used.

Because the Log4jService is loaded as a bootstrap MBean using the standard jboss.conf MLET configuration file, one must specify the service attributes using the MLET constructor syntax. The format used in the default jboss.conf file is:

```
<MLET CODE = "org.jboss.logging.Log4jService"
    ARCHIVE="jboss.jar,log4j.jar"
    CODEBASE=".../lib/ext/">
</MLET>
```

This form uses default values for the log4j configuration file and refresh period. The default configuration file is named "log4j.properties". The default refresh period is 60 seconds. The log4j configuration layer will look to see if the configuration file has changed after each

refresh period, and if it has, it will be reloaded. To specify an alternate classpath resource name for the log4j configuration file use:

```
<MLET CODE = "org.jboss.logging.Log4jService"
  ARCHIVE="jboss.jar,log4j.jar"
  CODEBASE="../../lib/ext/">
  <ARG TYPE="java.lang.String" VALUE="log-config.xml">
</MLET>
```

To specify both a classpath resource name for the log4j configuration file and the refresh period use:

```
<MLET CODE = "org.jboss.logging.Log4jService"
  ARCHIVE="jboss.jar,log4j.jar"
  CODEBASE="../../lib/ext/">
  <ARG TYPE="java.lang.String" VALUE="log-config.xml">
  <ARG TYPE="int" VALUE="180">
</MLET>
```

If you need to modify the log4j setup to add your category configuration, you need to modify the JBoss server log4j configuration file to add this information. Log4j does not currently support multiple instances of a configuration class, unless you arrange to load your configuration in an isolated class loader, and so you must augment the configuration file used by the JBoss server.

Installing and Using the Book Examples

The book archive contains an `examples-2.4.4` and `examples-2.4.5` directories. The `examples-2.4.4` work with the JBoss-2.4.4 release while the `examples 2.4.5` have been updated to work with the JBoss-2.4.5 release. Within each directory is an Ant `build.xml` file and `src` subdirectory that includes Java source code and associated files for the examples presented in every chapter. To install the book examples, copy the examples directory to any location you choose on your computer's hard drive. This location is referred to whenever a chapter example is referenced in the book. For example, if you copied the contents of Examples directory to `D:/JBossBook/examples` on a Win32 platform, a reference to the book CD examples directory would be to your `D:/JBossBook/examples` location.

All of the examples are built and run using the Ant `build.xml` file in the examples directory. The top of the `build.xml` file looks similar to the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- An Ant build file for the JBoss Book: JBoss Administration and
Development examples
-->
<project name="JBossBook 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="dist.root" value="G:/JBoss-2.4.5_Tomcat-4.0.3" />
<property name="jboss.dist" value="${dist.root}/jboss"/>
<property name="jboss.deploy.dir" value="${jboss.dist}/deploy"/>
<!-- Change if your not using tomcat -->
<property name="servlet.jar" value="${dist.root}/tomcat/lib/servlet.jar"/>

```

Note the **bolded** line in the previous build.xml file. This is a reference to the location of a JBoss distribution. You need to update the dist.root this to point to the location you have installed the JBoss-2.4.5_Tomcat-4.0.3 distribution. If you do not do this, any attempt to build the examples will fail because the required Java jar files will not be available for compilation. You can change the path to the JBoss-2.4.5_Tomcat-4.0.3 by either editing the build.xml file, or you can create an .ant.properties file in the directory containing the build.xml file. This is a standard Java properties file must contain a definition for the dist.root property. The following example illustrates how to change the dist.root property.

Suppose that you have copied the JBoss-2.4.5_Tomcat-4.0.3 distribution to your D:/JBossBook directory. To edit the examples build.xml file, you would change the dist.root property definition to the following:

```

<property name="dist.root" value="D:/JBossBook/JBoss-2.4.5_Tomcat-4.0.3" />

```

If you want to create a corresponding .ant.properties file, its contents would consist of the following line:

```

dist.root= D:/JBossBook/JBoss-2.4.5_Tomcat-4.0.3

```

Building and Running An Example

Ant is used to build and run the examples. All Ant commands must be run from your examples installation directory (for example, D:/JBossBook/Examples). The steps to build and run an example are as follows:

1. Compile the corresponding chapter source. Each chapter package has a build.xml file that compiles and jars the examples for the chapter. You use the build-chap Ant target to compile a chapter's source. The chapter number to compile is specified using a chap property. For example, to compile the Chapter 4 examples, use the following command:
Examples 972>ant -Dchap=4 build-chap
2. Optionally configure the JBoss server for the chapter examples. Chapters 7, 8, and 11 require modifications to the default JBoss server configuration to run. For these chapters, you need to create the custom JBoss server configuration. You use the config Ant target to create the custom configuration, and specific the chapter number using a chap property. For example, to create the custom Chapter 7

configuration, use the following command:

```
Examples 972>ant -Dchap=7 config
```

3. **Run the chapter example.** You use the run-example Ant target to run a chapterexample. Both the chapter number and example number need to be specified using the chap and ex properties, respectively. For example, to run the first example from Chapter 4, use the following command:

```
Examples 981>ant -Dchap=4 -Dex=1 run-example
```

Each chapter that presents an example goes through these steps.



16. Appendix E, Change Notes

Notes for the changes made after the 2.4.4 release.

2.4.5 Changes

Changes between JBoss 2_4_5_RC1 and JBoss 2_4_4

Rel_2_4_5_1

- Fix org/jboss/ejb/plugins/StatefulSessionInstanceInterceptor.java The security context must be established before the cache lookup because the SecurityInterceptor is after the instance interceptor and handles of passivated sessions expect that they are restored with the correct security context since the handles not serialize the principal and credential information. See Bug #511280
- Fixes for org/jboss/util/Scheduler.java
 - Change the behaviour when the Start Date is in the past. Now the Scheduler will behave as the Schedule is never stopped and find the next available time to start with respect to the settings. Therefore you can restart JBoss without adjust your Schedule every time. BUT you will still loose the calls during the Schedule was down.
 - Added parsing capabilities to setInitialStartDate. Now NOW: current time, and a string in a format the SimpleDateFormat understand in your environment (US: m/d/yy h:m a) but of course the time in ms since 1/1/1970.
 - Some fixes like the stopping a Schedule even if it already stopped etc.
 - Added the ability to call another MBean instead of an instance of the given Schedulable class. Use setSchedulableMBean() to specify the JMX Object Name pointing to the given MBean. Then if the MBean does not contain the same method as the Schedulable instance you have to specify the method with setSchedulableMBeanMethod(). There you can use some constants.
 - Fixed a bug not indicating that no MBean is used when setSchedulableClass() is called.
 - Fixed a bug therefore that when NOW is set as initial start date a restart of the Schedule will reset the start date to the current date
 - Fixed a bug because the start date was update during recalculation of the start date when the original start date is in the past (see above). With this you could restart the schedule even after all hits were fired when the schedule was not started immediately after the initial start date was set.

Rel_2_4_5_2

- CacheKey now uses the underlying key equals and hashCode method if they are implemented by the id class(not a superclass). See Bug #510397.
- org/jboss/ejb/plugins/BMPPersistenceManager.java Fix problem with wrapping application exceptions in UndeclaredThrowableException. See Bug #512465
- Added application deadlock detection to org/jboss/ejb/plugins/lock/BeanLockSupport.java, org/jboss/ejb/plugins/lock/QueuedPessimisticEJBLOCK.java

Rel_2_4_5_3

- Fix org/jboss/ejb/plugins/AbstractInstanceCache.java Passivation of an Entity Bean does not call removeRef() on the BeanLockManager which results in an object leak. If 'remove()' on the Entity is called, everything is OK and the entry is removed from the cache, but passivation fails to do this. See Bug #504895.

Rel_2_4_5_4

- Fix NullPointerException in JaasSecurityMgr indicated by patch #503590
- Fix problem with incorrect role assignment in UsersRolesLoginModule for users that have a common username prefix. See Bug #513245

Rel_2_4_5_5

- Fix org/jboss/ejb/plugins/jaws/jdbc/JDBCInitCommand.java 'Can't create table' log should be at error level, not debug! This can result from a problem with the bean itself, and should be made hard to ignore in order to avoid mistery problems.
- org/jboss/ejb/plugins/jaws/metadata/CMPFieldMetaData.java Make 'does not have a get method' messages more informative

Rel_2_4_5_6

- org/jboss/pool/{ObjectPool.java,PoolObjectFactory.java}, apply patch #515411 and remove legacy logging w/ PrintWriter, in favor of log4j
- org/jboss/pool/PoolParameters.java, Fix speling mistake in GCIntervalMillis attribute name so that it now works.
- org/jboss/pool/connector/jboss/{MinervaNoTransCMFactory.java,MinervaSharedLocalCMFactory.java,MinervaXACMFactory.java}, enable parameter TimestampUsed in jboss.jcml

Rel_2_4_5_9

- Fix org/jboss/ejb/plugins/{BMPPersistenceManager.java,CMPPersistenceManager.java} See Bug #529956 BMP cache zombies. Now bypass check of the cache on findByPrimaryKey with commit option B or C.
- Fix org/jboss/util/TimedCachePolicy.java threadSafe flag usage. The check was inverting the threadSafe and only creating a thread-safe collection when threadSafe was false.

Rel_2_4_5_10

- Fix org/jboss/util/Shutdown.java bug. The System.exit() call is now performed in a seperate thread. See Bug #536633

Rel_2_4_5_11

- org/jboss/metadata/WebMetaData.java Added support for specifying the context-root value in stand-alone wars via a jboss-web/context-root element in the jboss-web.xml descriptor.
- Added support for specifying the virtual host of a web application via a jboss-web/virtual-host element in the jboss-web.xml descriptor.
- org/jboss/web/AbstractWebContainer.java now performs the parsing of the web.xml and jboss-web.xml descriptors.
 - Changed the void parseWebAppDescriptors(ClassLoader loader, Element webApp, Element jbossWeb) throws Exception; method signature to void parseWebAppDescriptors(ClassLoader loader, WebMetaData metaData) throws Exception;

- o Changed the protected abstract `WebApplication` `performDeploy(String ctxPath, String warUrl, protected abstract void performDeploy(WebApplication webApp, String warUrl, WebDescriptorParser webAppParser) throws Exception;` method signature to protected abstract `void performDeploy(WebApplication webApp, String warUrl, WebDescriptorParser webAppParser) throws Exception;`. The context path is available via `webApp.getMetaData().getContextRoot()` and the virtual host is available via `webApp.getMetaData().getVirtualHost()`.
- `org/jboss/web/WebApplication.java` Drop the `web.xml` and `jboss-web.xml` DOM elements and add the `WebMetaData` which is populated by the `AbstractWebContainer`.
- Catalina emedded service. Drop the default engine and connector setup via JMX attributes and simply configure using the `XmlMapper` and a subset of the `server.xml` DTD as an embedded configuration within the `EmbeddedCatalinaServiceSX` config. The outline of the supported `server.xml` elements is:
- `<mbean code="org.jboss.web.catalina.EmbeddedCatalinaServiceSX" name="DefaultDomain:service=EmbeddedCatalinaSX">`
- `<attribute name="Config">`
- `<Service name="JBoss-Tomcat">`
- `<Connector className="some.connector" port="8080" />`
- `<Connector className="another.connector">`
- `<Factory className="another.factory" />`
- `</Connector>`
- `<Engine defaultHost="somehost" className="the.engine.class">`
- `<Logger className = "org.jboss.web.catalina.Log4jLogger"`
- `verbosityLevel = "trace" category =`
- `"org.jboss.web.somehost.Engine"/>`
- `<Listener className="dot.com.LifecycleListener" />`
- `<Host name="somehost">`
- `<Alias>www.somehost.net</Alias>`
- `<Valve className =`
- `"org.apache.catalina.valves.AccessLogValve"`
- `prefix = "somehost_access" suffix = ".log"`
- `pattern = "common"`
- `directory = "../jboss/log" />`
- `<DefaultContext cookies = "false" crossContext = "true"`
- `override = "true">`
- `<WrapperLifecycle />`
- `<InstanceListener />`
- `<WrapperListener />`
- `</DefaultContext>`
- `</Host>`
- `</Engine>`
- `</Service>`
- `<Service name="Service2">`
- `...`
- `</Service>`
- `</attribute>`
- `</mbean>`

Rel_2_4_5_12

- org/jboss/pool/jdbc/xa/wrapper/XAClientConnection.java, Apply fix for scrollable prepared statements. See Patch #506673.
- org/jboss/pool/PoolGCThread.java, org/jboss/pool/jdbc/xa/{XAPoolDataSource.java,XAConnectionFactory.java XAPoolDriver.java}, org/jboss/pool/jdbc/xa/wrapper/{XAConnectionImpl.java, XADataSourceImpl.java}, Apply fix for Bug #525840 to allow resource factory getConnection(username, password) and reduce logging threshold to trace for per connection messages
- org/jboss/pool/jdbc/{JDBCConnectionFactory.java,JDBCPoolDataSource}, Change to use log4j as the logging API.

Rel_2_4_5_13

- org/jboss/ejb/Container.java Use the local-jndi-name when binding a local home into JNDI rather than creating a dynamic key based on a timestamp.
- org/jboss/ejb/plugins/local/BaseLocalContainerInvoker.java Was not binding the local home proxy into JNDI when the container was started.
- org/jboss/web/AbstractWebContainer.java Added support for ejb-local-ref values to web apps.
- org/jboss/ejb/plugins/StatefulSessionInstanceInterceptor.java. Initial attempt to handle SessionSynchronization method failures as per section 18.3.3 of the EJB 2.0 spec. See Patches #537220 and #537246.
- org/jboss/metadata/XmlFileLoader.java Added local entity resolution for "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN" == "application_1_3.dtd" and "-//JBoss//DTD Web Application 2.2//EN" == "jboss-web.dtd"

Rel_2_4_5_14

- org/jboss/security/srp/SRPPParameters.java, Add hashAlgorithm, cipherAlgorithm, and cipherIV for user session encryption info.
- org/jboss/security/srp/SRPServerInterface.java, Add a close method to close a user SRP session.
- org/jboss/security/srp/jaas/SRPLoginModule.java, login adds the session key and SRPPParameters to the Subject private credentials. logout closes the user SRP session with the server.
- org/jboss/security/srp/jaas/SRPCacheLoginModule.java, login adds the session key and SRPPParameters to the Subject private credentials.
- org/jboss/security/auth/spi/UsersRolesLoginModule.java, Add new login module options: usersProperties: The name of the properties resource containing user/passwords. The default is "users.properties" rolesProperties: The name of the properties resource containing user/roles The default is "roles.properties".

Rel_2_4_5_15

Rel_2_4_5_16

- org/jboss/security/SecurityAssociation.java, added the following permissions: /** The permission required to access getPrincipal, getCredential, getSubject */ private static final RuntimePermission getPrincipalInfoPermission = new RuntimePermission("org.jboss.security.SecurityAssociation.getPrincipalInfo"); /** The permission required to access setPrincipal, setCredential, setSubject */ private static final RuntimePermission setPrincipalInfoPermission = new RuntimePermission("org.jboss.security.SecurityAssociation.setPrincipalInfo"); /** The permission required to access setServer */ private static final RuntimePermission setServerPermission = new RuntimePermission("org.jboss.security.SecurityAssociation.setServer");
- Added public static Subject getSubject()
- Integrated a patch to JBossMQ that adds heartbeat pings on the invocation connections.
- org/jboss/mq/Connection.java

- org/jboss/mq/Subscription.java, org/jboss/mq/selectors/Selector.java. Treat empty selectors as null selectors and include the selector in the InvalidSelectorException. Fixes Bug #537564.
- org/jboss/web/catalina/EmbeddedCatalinaServiceSX.java, Fix problem with not treating '/' as the default root context
- org/jboss/web/catalina/ConfigHandler.java now supports multiple Services within a Server tag. The correct Config attribute outline is now:
- `<mbean code="org.jboss.web.catalina.EmbeddedCatalinaServiceSX"`
- `name="DefaultDomain:service=EmbeddedCatalinaSX">`
- `<attribute name="Config">`
- `<Server>`
- `<Service name="JBoss-Tomcat">`
- `<Connector className="some.connector" port="8080" />`
- `<Connector className="another.connector">`
- `<Factory className="anothers.factory" />`
- `</Connector>`
- `<Engine defaultHost="somehost"`
- `className="the.engine.class">`
- `<Logger className =`
- `"org.jboss.web.catalina.Log4jLogger"`
- `verbosityLevel = "trace" category =`
- `"org.jboss.web.somehost.Engine"/>`
- `<Listener className="dot.com.LifecycleListener" />`
- `<Host name="somehost">`
- `<Alias>www.somehost.net</Alias>`
- `<Valve className =`
- `"org.apache.catalina.valves.AccessLogValve"`
- `prefix = "somehost_access" suffix =`
- `".log" pattern = "common"`
- `directory = "../jboss/log" />`
- `<DefaultContext cookies = "false" crossContext =`
- `"true" override = "true">`
- `<WrapperLifecycle />`
- `<InstanceListener />`
- `<WrapperListener />`
- `</DefaultContext>`
- `</Host>`
- `</Engine>`
- `</Service>`
- `<Service name="Service2">`
- `...`
- `</Service>`
- `</Server>`
- `</attribute>`
- `</mbean>`

Changes between JBoss_2_4_5_RC2 and JBoss_2_4_5_RC1

Rel_2_4_5_17

- `conf/default/jboss.jcml`, Remove obsolete services and update the timer service configuration in the default `jboss.jcml` file
- `org/jboss/ejb/plugins/jaws/jdbc/JDBCInitCommand.java`, 'Can't create table' log should be at error level, not debug! This can result from a problem with the bean itself, and should be made hard to ignore in order to avoid mistery problems.
- `org/jboss/metadata/XmlFileLoader.java`, Added local entity resolvers for the servlet web-app_2_2.dtd and web-app_2_3.dtd and added these DTDs to the `jboss.jar`
- `org/jboss/ejb/plugins/jaws/jdbc/JDBCLoadEntityCommand.java`, Resultsets are now read in column order. This related to some JDBC drivers (notably the free Microsoft SQL Server driver by Merant) requiring that columns be read in the order they are specified in the query. See Patch #520200 and Bug #517062.

Rel_2_4_5_18

- `conf/default/jboss.jcml`, Updated the local transaction connection factory `ConnectionURL` to correspond to the latest HypersonicSQL jdbc url
- `org/jboss/ejb/ContainerFactory.java`, Return a copy of the deployed applications to avoid concurrent access problems.
- `org/jboss/ejb` and `org/jboss/ejb/plugins` packages. Cleanup ejb container leaks that occur during deploy/undeploy cycles. Also removed JMS monitoring of the default instance pool and cache as this is not being maintained.

Rel_2_4_5_19

- `org/jboss/ejb` and `org/jboss/ejb/plugins` packages. More cleanup of ejb container deploy/undeploy memory leaks.

Rel_2_4_5_20

- `org/jboss/pool/connector/{BaseConnectionManager.java,SharedLocalConnectionManager.java}` Fix transaction and connection listener object leaks seen in the `SharedLocalConnectionManager`

Rel_2_4_5_21

- `JBossSX jars`, Add the `org.jboss.crypto.JBossSXProvider` to the `jboss-jaas.jar` and `jboss-sx-client.jar` to avoid `ClassNotFoundExceptions`
- `org/jboss/security/plugins/JaasSecurityDomain.java`, Restore the registration of the security domain with the `java:/jaas` namespace through JMX and the security manager service
- `org/jboss/security/srp/SRPVerifierStoreService.java`, Create intermediate subcontexts when binding the `SRPVerifierStore` interface into jndi
- `org/jboss/security/srp/SRPParameters.java`, `org/jboss/security/srp/jaas/SRPLoginModule.java`, Don't set a cipher algorithm by default and remove the static reference to the `javax.crypto.spec.SecretKeySpec` from `SRPLoginModule` so that clients do not need JCE unless they setup the `SRPParameters` to include a cipher algorithm.
- `org/jboss/web/catalina/EmbeddedCatalinaServiceSXMBean.java`, Allow the setting of the `catalina.base` property independently of the `catalina.home` property using the `CatalinaBase` attribute.

Changes between JBoss 2_4_5_RC3 and JBoss 2_4_5_RC2

Rel_2_4_5_22

- org/jboss/security/plugins/{JaasSecurityManagerServiceMBean.java, JaasSecurityManagerService.java}, Add support for chaining JAAS login configuration implementations and wrap the default auth.conf based implementation in a dynamic mbean that is installed as the default.
- org/jboss/security/plugins/JaasSecurityDomain.java, Dynamically install the com.sun.net.ssl.internal.ssl.Provider if one is not found to allow the JSSE jars to be bundled with the JBoss server.
- org/jboss/web/catalina/{EmbeddedCatalinaServiceSXMBean.java, EmbeddedCatalinaServiceSX.java}, Add a Java2ClassLoadingCompliance attribute to allow one to override the servlet 2.3 class loading model in favor of the Java2 parent delegation model and set this to true by default since this was causing too many problems with war deployments that included classes duplicated elsewhere.
- org/jboss/ejb/plugins/jaws/metadata/FinderMetaData.java, honor the fact that the jaws_2_4.dtd states that the order tag is optional. See Bug #546374.
- org/jboss/ejb/plugins/jaws/bmp/CustomFindByEntitiesCommand.java, log the cause of the FinderException before throwing the exception. See Bug #512649

Changes between JBoss 2_4_5 and JBoss 2_4_5_RC3

Rel_2_4_5_23

- Use the standard log4j org.apache.log4j.ConsoleAppender rather than the custom JBoss appender so that either log4j1.1.3 or log4j1.2 may be used.

Rel_2_4_5_24

- RCS file: /cvsroot/jboss/jboss/src/main/org/jboss/deployment/{LegacyInstaller.java, LocalDirInstaller.java}, org/jboss/web/AbstractWebContainer.java, Fix problem with using the temporary war filename for standalone wars that do not explicitly specify their context-root. See Bug #545160.
- org/jboss/ejb/plugins/{LRUEnterpriseContextCachePolicy.java, LRUStatefulContextCachePolicy.java}, The passivation timers were only running one time because of not passing the repeat period to the timer scheduler. See Patch #551618.
- org/jboss/ejb/plugins/StatefulSessionInstanceInterceptor.java, Backport the setting of the caller principal security information before obtaining the session from the cache so that secured references are properly restored. See Bug #552157.
- EJB container config override simplified Simplify support for overriding standard container configurations. The container-configuration now supports an extends attribute through which you can specify which standard container-configuration/container-name you want to extend or override. This allows for simple modifications of container configurations without having to completely redefine the configuration. For example, to set the commit option to A on a given entity bean deployment:
 - `<container-configurations>`
 - `<container-configuration extends="Standard CMP EntityBean">`
 - `<container-name>Cached EntityBean</container-name>`
 - `<commit-option>B</commit-option>`
 - `</container-configuration>`
 - `</container-configurations>`
- org/jboss/naming/NamingService.java, org/jnp/interfaces/{NamingContext.java, TimedSocketFactory.java}, org/jnp/server/{Main.java, MainMBean.java} Added support for customizing the bootstrap sockets and establishing a connection timeout.
 - jnp.socketFactory, The jnp protocol socket factory class which defaults to org.jnp.interfaces.TimedSocketFactory

- jnp.timeout, TimedSocketFactory connection timeout in milliseconds(0 == blocking)
- jnp.sotimeout, The TimedSocketFactory read timeout in milliseconds(0 == blocking)

Rel_2_4_5_25

- org/jboss/pool/jdbc/PreparedStatementInPool.java, Integrate Patch #532376 to close the internal PreparedStatement when the pool statement is closed.
- org/jboss/tm/usertx/server/UserTransactionSessionImpl.java, When committing a tx resume any current tx afterwards. See Patch #541946.
- Drop the bin/admin directory and the as its contents is unsupported.
- JBossMQ, Backporting many memory leak fixes. Fixed SpyMessage leak due to selector usage on topics. Fixed Subscription object that occurred on on both the server and client side. See Bug #554502.

Changes between JBoss_2_4_6 and JBoss_2_4_5

- org/jboss/pool/jdbc/PreparedStatementInPool.java, Back out the change to close the db connection as this is causing too many problems not seen in testing

17. Index

A

AbstractWebContainer	317
Subclassing	323
Apache	
and AJP connector	338
and Tomcat	337
auth.conf	15

C

Catalina	<i>See</i> Tomcat-4.x
CD Contents	431
Classpath	
build path	351
ClientLoginModule	274
CMP	<i>See</i> Container managed persistence
Container Managed Persistence	161
Custom example	166
JAWS	182
JBossCMP	162
CVS	
Anonymous access	19
Clients	19
JBoss repositories	<i>See</i> SourceForge

D

DatabaseServerLoginModule	272
Dynamic MBeans	33

E

EJB	
container architecture	66
Container cache configuration	313
Container commit option configuration	315
Container configuration	307
Container interceptor configuration	311
Container locking policy configuration	315
Container persistence configuration	314
Container plugin framework	80
Dynamic proxies	67
Instance pool configuration	312
local references	<i>See</i> ejb-local-ref
Message driven beans	137
method permissions	237
references	<i>See</i> ejb-ref
Tracing the call through container	90
ejb-jar.xml	
ENC elements	100
Security elements	232
ejb-local-ref	110
ejb-ref	107
and JBoss descriptors	109
<u>EmbeddedCatalinaServiceSX</u>	325
EmbeddedTomcatServiceSX	339

ENC	97. <i>See Also</i> JNDI Application Component Environment and UserTransactions
env-entry	105

I

IdentityLoginModule	265
---------------------------	-----

J

J2EE	
About	2
APIs	2
declarative security overview	231
JAAS	
Authentication	244
Introduction to	242
Login code	245
LoginModule	246
Principal	243
Subject	243
Java Pet Store	
Migrating to JBoss	376
JAWS	<i>See</i> JBossCMP
jaws.xml	183
DTD reference	423
JBoss	
and JMX	38
Building from source	18
Building the JBoss-2.4.5 distribution	21
Building the JBoss-2.4.5/Tomcat-4.0.3 bundle	26
Configuration files	14
Directory structure	12
Enabling declarative security	242
Installing the binary	12
JMX bootstrap	39
Security model	248
Service interface	42
JBoss Group	
About	393
jboss.conf	15, 51
Syntax	36
jboss.jcml	16
default	54
DTD	40
DTD reference	427
jboss.xml	
Container configuration	307
DTD	303
DTD reference	408
ENC elements	104
MDB elements	141
Security elements	250
JBossCMP	
Architecture	162
Custom finder methods	191
Customizing	183
type mappings	187
JBossCX	
Architecture	214

MBeans	215
Sample adaptor	218
JBossMQ	
Application Server Facilities	151
Architecture	147
Configuring	153
jbossmq-state.xml	
DTD reference	429
JBossNS	
Architecture	116
InitialContext Factory	119
jndi.properties settings	119
MBeans	121
JBossSX	256
Architecture	256
Custom security proxy	252
Login modules	265
MBeans	262
Subject usage pattern	277
Writing custom login modules	275
JBossTX	
Adapting a Transaction Manager	205
Internals	204
jboss-web.xml	
context-root	318
DTD Graphic	317
DTD reference	426
ENC elements	104
Security elements	250
virtual-host	318
JCA	209
Common Client Interface	209
Overview	209
JDBC	
Configuring	192
Jetty	339
JettyService	339
JMS	
and J2EE	137
API	131
Introduction to	128
PTP Example	134
JMX	28
Introduction to	29
MBeans	32
MLet	36
MLet configuration	36
JNDI	
Application Component Environment	97
ENC	<i>See</i> JNDI Application Component Environment
ENC conventions	99
Overview	94
Viewing	124
jndi.properties	16
JSSE	
Installing	296
JBoss and SSL	296

K

keystore	297
----------------	-----

L

LdapLoginModule	268
LGPL	393
Log4j	
Appender	442
Category	439
DOMConfigurator DTD	445
JBoss Logger	441
Layout	442
MBeans	448
PropertyConfigurator	442
Usage patterns	447
Using	439
log4j.properties	17, 443
Login modules	<i>See</i> JAAS

M

mail.properties	17
MBean	
com.sun.jdmk.comm.HtmlAdaptorServer	65
org.jboss.configuration.ConfigurationService	41
org.jboss.deployment.J2eeDeployer	63
org.jboss.ejb.AutoDeployer	64
org.jboss.ejb.ContainerFactory	62
org.jboss.jdbc.HypersonicDatabase	197
org.jboss.jdbc.JdbcProvider	193
org.jboss.jdbc.XADataSourceLoader	194
org.jboss.jetty.JettyService	339
org.jboss.jms.asf.ServerSessionPoolLoader	159
org.jboss.jms.jndi.JMSProviderLoader	159
org.jboss.jmx.server.RMICConnectorService	64
org.jboss.logging.Log4jService	52
org.jboss.mail.MailService	65
org.jboss.mq.il.jvm.JVMServerILService	157
org.jboss.mq.il.oil.OILServerILService	156
org.jboss.mq.il.oil.UILServerILService	156
org.jboss.mq.il.rmi.RMIServerILService	158
org.jboss.mq.pm.file.PersistanceManager	155
org.jboss.mq.pm.jdbc.PersistanceManager	155
org.jboss.mq.pm.rollinglogged.PersistanceManager	155
org.jboss.mq.server.JBossMQService	153
org.jboss.mq.server.QueueManager	158
org.jboss.mq.server.StateManager	154
org.jboss.mq.server.TopicManager	158
org.jboss.naming.ExternalContext	121
org.jboss.naming.JNDIView	124
org.jboss.naming.NamingService	117
org.jboss.resource.ConnectionFactoryLoader	217
org.jboss.resource.ConnectionManagerFactoryLoader	215
org.jboss.resource.RARDeployer	218
org.jboss.security.plugins.JaasSecurityDomain	264
org.jboss.security.plugins.JaasSecurityManagerService	263
org.jboss.security.srp.SRPVerifierStoreService	285
org.jboss.security.srp.SRPService	284
org.jboss.tm.TransactionManagerService	206
org.jboss.tm.usertx.server.ClientUserTransactionService	207
org.jboss.tomcat.EmbeddedTomcatServiceSX	339
org.jboss.util.ClassPathExtension	53
org.jboss.util.Info	53
org.jboss.util.Scheduler	66
org.jboss.util.ServiceControl	43
org.jboss.web.catalina.EmbeddedCatalinaServiceSX	325

org.jboss.web.WebService	62
Writing JBoss services	41
MDB	<i>See</i> Message driven beans
Message driven beans	137
example	138
Messaging	<i>See</i> JBossMQ
method permission	237
Model MBeans	33

O

Open MBeans	33
Open Source	
definition	1

P

Passivation	
timeout setting	313
Priorities	
Logging	447
ProxyLoginModule	274

R

Resource adapters	<i>See</i> JBossCX
resource-env-ref	115
and JBoss descriptors	116
resource-ref	112
and JBoss descriptors	114
RFC2945	<i>See</i> SRP
RMI	
Configuring for EJBs	310
Over SSL	296

S

Security	
Enabling in JBoss	242, 250
Security Manager	
Running with	293
security-constraint	240
security-identity	235
security-role	236
security-role-ref	234
server.log	443
Servlet Containers	
Integrating	317
SourceForge	18
JBoss project	19

Understanding the JBoss CVS modules	19
SRP	281
Algorithm	286
Example	289
JBossSX features	282
JBossSX implementation	282
login modules	283
Sample login config	284
SRPLoginModule options	283
SRPLoginModule	283
SSL	
and EJBs	296
and Tomcat-3.2.x	339
and Tomcat-4.x	331
Installing JSSE	296
Standard MBeans	33
standardjaws.xml	17, 183
standardjboss.xml	17
as default for jboss.xml	303

T

Tomcat-3.2.3	339
Tomcat-4.x	325
and virtual hosts	335
Configuring	326
Setting up SSL	331
Transaction	
Overview	199
Transaction Manager	199
Tyrex Transaction Manager	206

U

UsersRolesLoginModule	266
UserTransaction	
Support	207
Using JBoss	341
Building and assembling apps	349
Mail forwarding app	342

W

web.xml	
ENC elements	101
Security elements	232

X

X License	406
-----------------	-----