



JBoss Enterprise Application Platform 4.3

Server Configuration Guide

for Use with JBoss Enterprise Application Platform 4.3

Edition 4.3.10

JBoss Enterprise Application Platform 4.3 Server Configuration Guide

for Use with JBoss Enterprise Application Platform 4.3
Edition 4.3.10

Red Hat Documentation Group

Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This book is a guide to configuring JBoss Enterprise Application Platform 4.3 and its patch releases.

Table of Contents

PREFACE	12
1. WHAT THIS BOOK COVERS	12
PART I. JAVA EE 5 APPLICATION CONFIGURATION	13
CHAPTER 1. ENTERPRISE APPLICATIONS WITH EJB3 SERVICES	14
1.1. SESSION BEANS	14
1.2. ENTITY BEANS (A.K.A. JAVA PERSISTENCE API)	16
1.2.1. The persistence.xml file	18
1.2.2. Use Alternative Databases	19
1.2.3. Default Hibernate options	20
1.3. MESSAGE DRIVEN BEANS	20
1.4. PACKAGE AND DEPLOY EJB3 SERVICES	21
1.4.1. Deploy the EJB3 JAR	22
1.4.2. Deploy EAR with EJB3 JAR	22
CHAPTER 2. NETWORK	25
2.1. ENABLING IPV6	25
CHAPTER 3. DATASOURCE CONFIGURATION	27
3.1. TYPES OF DATASOURCES	27
3.2. DATASOURCE PARAMETERS	28
3.3. DATASOURCE EXAMPLES	32
3.3.1. Generic Datasource Example	32
3.3.2. Configuring a DataSource for Remote Usage	34
3.3.3. Configuring a Datasource to Use Login Modules	35
CHAPTER 4. DEPLOYMENT	37
4.1. DEPLOYABLE APPLICATION TYPES	37
4.2. STANDARD SERVER CONFIGURATIONS	37
4.2.1. The production Configuration	38
4.2.2. Further Tuning from the production Configuration	38
PART II. JBOSS AS INFRASTRUCTURE	40
CHAPTER 5. THE JBOSS JMX MICROKERNEL	41
5.1. AN INTRODUCTION TO JMX	41
5.1.1. Instrumentation Level	42
5.1.2. Agent Level	43
5.1.3. Distributed Services Level	43
5.1.4. JMX Component Overview	43
5.1.4.1. Managed Beans or MBeans	44
5.1.4.2. Notification Model	44
5.1.4.3. MBean Metadata Classes	44
5.1.4.4. MBean Server	45
5.1.4.5. Agent Services	46
5.2. JBOSS JMX IMPLEMENTATION ARCHITECTURE	46
5.2.1. The JBoss ClassLoader Architecture	46
5.2.2. Class Loading and Types in Java	46
5.2.2.1. ClassCastExceptions - I'm Not Your Type	47
5.2.2.2. IllegalAccessException - Doing what you should not	51
5.2.2.3. LinkageErrors - Making Sure You Are Who You Say You Are	54
5.2.2.3.1. Debugging Class Loading Issues	58

5.2.2.4. Inside the JBoss Class Loading Architecture	60
5.2.2.4.1. Viewing Classes in the Loader Repository	61
5.2.2.4.2. Scoping Classes	63
5.2.2.4.3. The Complete Class Loading Model	63
5.2.3. JBoss XMBeans	65
5.2.3.1. Descriptors	66
5.2.3.2. The Management Class	69
5.2.3.3. The Constructors	69
5.2.3.4. The Attributes	70
5.2.3.5. The Operations	71
5.2.3.6. Notifications	71
5.3. CONNECTING TO THE JMX SERVER	72
5.3.1. Inspecting the Server - the JMX Console Web Application	72
5.3.1.1. Securing the JMX Console	74
5.3.2. Connecting to JMX Using RMI	76
5.3.3. Command Line Access to JMX	79
5.3.3.1. Connecting twiddle to a Remote Server	80
5.3.3.2. Sample twiddle Command Usage	80
5.3.4. Connecting to JMX Using Any Protocol	84
5.4. USING JMX AS A MICROKERNEL	84
5.4.1. The Startup Process	84
5.4.2. JBoss MBean Services	86
5.4.2.1. The SARDeployer MBean	86
5.4.2.2. The Service Life Cycle Interface	90
5.4.2.3. The ServiceController MBean	90
5.4.2.3.1. The create(ObjectName) method	91
5.4.2.3.2. The start(ObjectName) method	91
5.4.2.3.3. The stop(ObjectName) method	91
5.4.2.3.4. The destroy(ObjectName) method	92
5.4.2.4. Specifying Service Dependencies	92
5.4.2.5. Identifying Unsatisfied Dependencies	94
5.4.2.6. Hot Deployment of Components, the URLDeploymentScanner	94
5.4.3. Writing JBoss MBean Services	95
5.4.3.1. A Standard MBean Example	96
5.4.3.2. XMBean Examples	100
5.4.3.2.1. Version 1, The Annotated JNDIMap XMBean	100
5.4.3.2.2. Version 2, Adding Persistence to the JNDIMap XMBean	104
5.4.4. Deployment Ordering and Dependencies	109
5.5. JBOSS DEPLOYER ARCHITECTURE	120
5.5.1. Deployers and ClassLoaders	121
5.6. REMOTE ACCESS TO SERVICES, DETACHED INVOKERS	122
5.6.1. A Detached Invoker Example, the MBeanServer Invoker Adaptor Service	125
5.6.2. Detached Invoker Reference	130
5.6.2.1. The JRMPInvoker - RMI/JRMP Transport	130
5.6.2.2. The PooledInvoker - RMI/Socket Transport	131
5.6.2.3. The IIOPInvoker - RMI/IIOP Transport	131
5.6.2.4. The JRMPProxyFactory Service - Building Dynamic JRMP Proxies	132
5.6.2.5. The HttpInvoker - RMI/HTTP Transport	132
5.6.2.6. The HA JRMPInvoker - Clustered RMI/JRMP Transport	133
5.6.2.7. The HA HttpInvoker - Clustered RMI/HTTP Transport	133
5.6.2.8. HttpProxyFactory - Building Dynamic HTTP Proxies	133
5.6.2.9. Steps to Expose Any RMI Interface via HTTP	134

CHAPTER 6. NAMING ON JBOSS	136
The JNDI Naming Service	136
6.1. AN OVERVIEW OF JNDI	136
6.1.1. Names	136
6.1.2. Contexts	137
6.1.2.1. Obtaining a Context using InitialContext	137
6.2. THE JBOSSNS ARCHITECTURE	138
6.3. THE NAMING INITIALCONTEXT FACTORIES	140
6.3.1. The standard naming context factory	141
6.3.2. The org.jboss.naming.NamingContextFactory	142
6.3.3. Naming Discovery in Clustered Environments	142
6.3.4. The HTTP InitialContext Factory Implementation	143
6.3.5. The Login InitialContext Factory Implementation	144
6.3.6. The ORBInitialContextFactory	144
6.4. JNDI OVER HTTP	145
6.4.1. Accessing JNDI over HTTP	145
6.4.2. Accessing JNDI over HTTPS	147
6.4.3. Securing Access to JNDI over HTTP	150
6.4.4. Securing Access to JNDI with a Read-Only Unsecured Context	151
6.5. ADDITIONAL NAMING MBEANS	154
6.5.1. JNDI Binding Manager	154
6.5.2. The org.jboss.naming.NamingAlias MBean	155
6.5.3. org.jboss.naming.ExternalContext MBean	156
6.5.4. The org.jboss.naming.JNDIView MBean	158
6.6. J2EE AND JNDI - THE APPLICATION COMPONENT ENVIRONMENT	160
6.6.1. ENC Usage Conventions	161
6.6.1.1. Environment Entries	162
6.6.1.2. EJB References	163
6.6.1.3. EJB References with jboss.xml and jboss-web.xml	165
6.6.1.4. EJB Local References	166
6.6.1.5. Resource Manager Connection Factory References	167
6.6.1.6. Resource Manager Connection Factory References with jboss.xml and jboss-web.xml	169
6.6.1.7. Resource Environment References	170
6.6.1.8. Resource Environment References and jboss.xml, jboss-web.xml	171
CHAPTER 7. CONNECTORS ON JBOSS	172
The JCA Configuration and Architecture	172
7.1. JCA OVERVIEW	172
7.2. AN OVERVIEW OF THE JBOSSCX ARCHITECTURE	174
7.2.1. BaseConnectionManager2 MBean	175
7.2.2. RARDeployment MBean	175
7.2.3. JBossManagedConnectionPool MBean	176
7.2.4. CachedConnectionManager MBean	177
7.2.5. A Sample Skeleton JCA Resource Adaptor	177
7.3. CONFIGURING JDBC DATASOURCES	183
7.4. CONFIGURING GENERIC JCA ADAPTORS	193
CHAPTER 8. TRANSACTIONS ON JBOSS	198
The JTA Transaction Service	198
8.1. TRANSACTION/JTA OVERVIEW	198
8.1.1. Pessimistic and optimistic locking	199
8.1.2. The components of a distributed transaction	199
8.1.3. The two-phase XA protocol	200

8.1.4. Heuristic exceptions	200
8.1.5. Transaction IDs and branches	201
8.2. JTS SUPPORT	201
8.3. WEB SERVICES TRANSACTIONS	201
8.4. CONFIGURING JBOSS TRANSACTIONS	202
8.5. LOCAL VERSUS DISTRIBUTED TRANSACTIONS	202
CHAPTER 9. MESSAGING ON JBOSS	203
9.1. JBOSS MESSAGING OVERVIEW	203
CHAPTER 10. SECURITY ON JBOSS	204
J2EE Security Configuration and Architecture	204
10.1. J2EE DECLARATIVE SECURITY OVERVIEW	204
10.1.1. Security References	204
10.1.2. Security Identity	205
10.1.3. Security roles	207
10.1.4. EJB method permissions	209
10.1.5. Web Content Security Constraints	212
10.1.6. Enabling Declarative Security in JBoss	214
10.2. AN INTRODUCTION TO JAAS	214
10.2.1. What is JAAS?	214
10.2.1.1. The JAAS Core Classes	214
10.2.1.1.1. The Subject and Principal Classes	215
10.2.1.1.2. Authentication of a Subject	215
10.3. THE JBOSS SECURITY MODEL	218
10.3.1. Enabling Declarative Security in JBoss Revisited	220
10.4. THE JBOSS SECURITY EXTENSION ARCHITECTURE	224
10.4.1. How the JaasSecurityManager Uses JAAS	227
10.4.2. The JaasSecurityManagerService MBean	230
10.4.3. The JaasSecurityDomain MBean	232
10.5. DEFINING SECURITY DOMAINS	233
10.5.1. Loading Security Domains	235
10.5.2. The DynamicLoginConfig service	236
10.5.3. Using JBoss Login Modules	237
10.5.3.1. Password Stacking	237
10.5.3.2. Password Hashing	238
10.5.3.3. Unauthenticated Identity	239
10.5.3.4. UsersRolesLoginModule	240
10.5.3.5. LdapLoginModule	240
10.5.3.6. LdapExtLoginModule	244
10.5.3.7. DatabaseServerLoginModule	254
10.5.3.8. BaseCertLoginModule	255
10.5.3.9. IdentityLoginModule	257
10.5.3.10. RunAsLoginModule	258
10.5.3.11. ClientLoginModule	258
10.5.4. Writing Custom Login Modules	259
10.5.4.1. Support for the Subject Usage Pattern	260
10.5.4.2. A Custom LoginModule Example	265
10.6. THE SECURE REMOTE PASSWORD (SRP) PROTOCOL	268
10.6.1. Providing Password Information for SRP	271
10.6.2. Inside of the SRP algorithm	273
10.6.2.1. An SRP example	275
10.7. RUNNING JBOSS WITH A JAVA 2 SECURITY MANAGER	278

10.8. USING SSL WITH JBOSS	281
10.8.1. Adding SSL to EJB3	281
10.8.1.1. Generating the keystore and truststore	281
10.8.1.2. Setting up the SSL transport	282
10.8.1.3. Configuring your beans to use the SSL transport	283
10.8.1.4. Setting up the client to use the truststore	284
10.8.2. Adding SSL to EJB 2.1 calls	284
10.8.2.1. Generating the keystore and truststore	284
10.8.2.2. Setting up the SSL transport	284
10.8.2.3. Configuring your beans to use the SSL transport	285
10.8.2.4. Setting up the client to use the truststore	288
10.9. CONFIGURING JBOSS FOR USE BEHIND A FIREWALL	288
10.10. HOW TO SECURE THE JBOSS SERVER	291
10.10.1. The JMX Console	291
10.10.2. The Web Console	292
10.10.3. The HTTP Invokers	292
10.10.4. The JMX Invoker	292
CHAPTER 11. WEB SERVICES	293
11.1. DOCUMENT/LITERAL	293
11.2. DOCUMENT/LITERAL (BARE)	293
11.3. DOCUMENT/LITERAL (WRAPPED)	294
11.4. RPC/LITERAL	295
11.5. RPC/ENCODED	296
11.6. WEB SERVICE ENDPOINTS	296
11.7. PLAIN OLD JAVA OBJECT (POJO)	296
11.8. THE ENDPOINT AS A WEB APPLICATION	296
11.9. PACKAGING THE ENDPOINT	297
11.10. ACCESSING THE GENERATED WSDL	297
11.11. EJB3 STATELESS SESSION BEAN (SLSB)	297
11.12. ENDPOINT PROVIDER	298
11.13. POJO ENDPOINT AUTHENTICATION AND AUTHORIZATION	299
11.14. WEBSERVICECONTEXT	300
11.15. WEB SERVICE CLIENTS	301
11.15.1. Service	301
11.15.1.1. Service Usage	301
11.15.1.2. Handler Resolver	302
11.15.1.3. Executor	302
11.15.2. Dynamic Proxy	302
11.15.3. WebServiceRef	303
11.15.4. Dispatch	305
11.15.5. Asynchronous Invocations	305
11.15.6. Oneway Invocations	306
11.16. COMMON API	307
11.16.1. Handler Framework	307
11.16.1.1. Logical Handler	307
11.16.1.2. Protocol Handler	307
11.16.1.3. Service endpoint handlers	307
11.16.1.4. Service client handlers	307
11.16.2. Message Context	308
11.16.2.1. Logical Message Context	308
11.16.2.2. SOAP Message Context	308
11.16.3. Fault Handling	308

11.17. DATABINDING	309
11.17.1. Using JAXB with non annotated classes	309
11.18. ATTACHMENTS	309
11.18.1. MTOM/XOP	309
11.18.1.1. Supported MTOM parameter types	310
11.18.1.2. Enabling MTOM per endpoint	310
11.18.2. SwaRef	311
11.18.2.1. Using SwaRef with JAX-WS endpoints	311
11.18.2.2. Starting from WSDL	312
11.19. TOOLS	313
11.19.1. Bottom-Up (Using wsprovide)	313
11.19.2. Top-Down (Using wsconsume)	315
11.19.3. Client Side	317
11.19.4. Command-line & Ant Task Reference	320
11.19.5. JAX-WS binding customization	320
11.20. WEB SERVICE EXTENSIONS	320
11.20.1. WS-Addressing	320
11.20.1.1. Specifications	320
11.20.1.2. Addressing Endpoint	321
11.20.1.3. Addressing Client	322
11.20.2. WS-BPEL	323
11.20.3. WS-Security	324
11.20.3.1. Endpoint configuration	324
11.20.3.2. Server side WSSE declaration (jboss-wsse-server.xml)	324
11.20.3.3. Client side WSSE declaration (jboss-wsse-client.xml)	325
11.20.3.3.1. Client side key store configuration	326
11.20.3.4. Installing the BouncyCastle JCE provider (JDK 1.4)	327
11.20.3.5. Keystore, truststore - What?	327
11.20.4. WS-Transaction	327
11.20.5. XML Registries	327
11.20.5.1. Apache jUDDI Configuration	328
11.20.5.2. JBoss JAXR Configuration	329
11.20.5.3. JAXR Sample Code	329
11.20.5.4. Troubleshooting	332
11.20.5.5. Resources	333
11.21. JBOSSWS EXTENSIONS	333
11.21.1. Proprietary Annotations	333
11.21.1.1. EndpointConfig	333
11.21.1.2. WebContext	334
11.21.1.3. SecurityDomain	335
CHAPTER 12. ADDITIONAL SERVICES	337
12.1. MEMORY AND THREAD MONITORING	337
12.2. THE LOG4J SERVICE	337
12.3. SYSTEM PROPERTIES MANAGEMENT	338
12.4. PROPERTY EDITOR MANAGEMENT	339
12.5. SERVICES BINDING MANAGEMENT	339
12.5.1. AttributeMappingDelegate	341
12.5.2. XSLTConfigDelegate	341
12.5.3. XSLTFileDelegate	342
12.5.4. The Sample Bindings File	344
12.6. RMI DYNAMIC CLASS LOADING	345
12.7. SCHEDULING TASKS	345

12.7.1. org.jboss.varia.scheduler.Scheduler	345
12.8. THE TIMER SERVICE	348
12.9. THE BARRIERCONTROLLER SERVICE	350
12.10. EXPOSING MBEAN EVENTS VIA SNMP	353
PART III. CLUSTERING CONFIGURATION	355
CHAPTER 13. CLUSTERING	356
High Availability Enterprise Services via JBoss Clusters	356
13.1. INTRODUCTION	356
13.2. CLUSTER DEFINITION	356
13.3. SEPARATING CLUSTERS	357
13.4. HAPARTITION	359
13.5. JBOSS CACHE CHANNELS	360
13.5.1. Service Architectures	360
13.5.1.1. Client-side interceptor architecture	361
13.5.1.2. Load balancer	362
13.5.2. Load-Balancing Policies	362
13.5.2.1. Client-side interceptor architecture	363
13.5.2.2. External load balancer architecture	363
13.5.3. Farming Deployment	363
13.5.4. Distributed state replication services	365
CHAPTER 14. CLUSTERED JNDI SERVICES	366
14.1. HOW IT WORKS	366
14.2. CLIENT CONFIGURATION	368
14.2.1. For clients running inside the application server	368
14.2.2. For clients running outside the application server	369
14.2.3. JBoss configuration	370
CHAPTER 15. CLUSTERED SESSION EJBS	374
15.1. STATELESS SESSION BEAN IN EJB 2.X	374
15.2. STATEFUL SESSION BEAN IN EJB 2.X	375
15.2.1. The EJB application configuration	375
15.2.2. Optimize state replication	376
15.2.3. The HASessionState service configuration	376
15.2.4. Handling Cluster Restart	376
15.2.5. JNDI Lookup Process	378
15.2.6. SingleRetryInterceptor	378
15.3. STATELESS SESSION BEAN IN EJB 3.0	379
15.4. STATEFUL SESSION BEANS IN EJB 3.0	379
CHAPTER 16. CLUSTERED ENTITY EJBS	384
16.1. ENTITY BEAN IN EJB 2.X	384
16.2. ENTITY BEAN IN EJB 3.0	385
16.2.1. Configure the distributed cache	385
16.2.2. Configure the entity beans for cache	386
16.2.3. Query result caching	388
CHAPTER 17. HTTP SERVICES	391
17.1. CONFIGURING LOAD BALANCING USING APACHE AND MOD_JK	391
17.2. DOWNLOAD THE SOFTWARE	391
17.3. CONFIGURE APACHE TO LOAD MOD_JK	392
17.4. CONFIGURE WORKER NODES IN MOD_JK	393
17.5. CONFIGURING JBOSS TO WORK WITH MOD_JK	394

17.6. CONFIGURING HTTP SESSION STATE REPLICATION	395
17.7. ENABLING SESSION REPLICATION IN YOUR APPLICATION	397
17.8. USING FIELD LEVEL REPLICATION	399
17.9. MONITORING SESSION REPLICATION	401
17.10. USING CLUSTERED SINGLE SIGN ON	401
17.11. CLUSTERED SESSION NOTIFICATION POLICY	402
CHAPTER 18. CLUSTERED SINGLETON SERVICES	403
18.1. HASINGLETONDEPLOYER SERVICE	403
18.2. MBEAN DEPLOYMENTS USING HASINGLETONCONTROLLER	404
18.3. HASINGLETON DEPLOYMENTS USING A BARRIER	406
18.4. DETERMINING THE MASTER NODE	406
CHAPTER 19. JBOSSCACHE AND JGROUPS SERVICES	408
19.1. JGROUPS CONFIGURATION	408
19.2. COMMON CONFIGURATION PROPERTIES	410
19.3. TRANSPORT PROTOCOLS	410
19.3.1. UDP configuration	410
19.3.2. TCP configuration	412
19.3.3. TUNNEL configuration	413
19.4. DISCOVERY PROTOCOLS	413
19.4.1. PING	414
19.4.2. TCPGOSSIP	415
19.4.3. TCPPING	415
19.4.4. MPING	416
19.5. FAILURE DETECTION PROTOCOLS	416
19.5.1. FD	416
19.5.2. FD_SOCKET	417
19.5.3. VERIFY_SUSPECT	417
19.5.4. FD versus FD_SOCKET	417
19.6. RELIABLE DELIVERY PROTOCOLS	419
19.6.1. UNICAST	419
19.6.2. NAKACK	419
19.7. OTHER CONFIGURATION OPTIONS	420
19.7.1. Group Membership	420
19.7.2. Flow Control	420
19.7.2.1. Why is FC needed on top of TCP ? TCP has its own flow control !	421
19.7.2.2. So do I always need FC?	421
19.7.3. Fragmentation	422
19.7.4. State Transfer	422
19.7.5. Distributed Garbage Collection	422
19.7.6. Merging	423
19.7.7. Binding JGroups Channels to a particular interface	424
19.7.8. Isolating JGroups Channels	424
19.7.9. Changing the Group Name	425
19.7.10. Changing the multicast address and port	425
19.7.11. JGroups Troubleshooting	426
19.7.12. Causes of missing heartbeats in FD	427
PART IV. LEGACY EJB SUPPORT	428
CHAPTER 20. EJBS ON JBOSS	429
The EJB Container Configuration and Architecture	429
20.1. THE EJB CLIENT SIDE VIEW	429

20.1.1. Specifying the EJB Proxy Configuration	432
20.2. THE EJB SERVER SIDE VIEW	436
20.2.1. Detached Invokers - The Transport Middlemen	436
20.2.2. The HA JRMPInvoker - Clustered RMI/JRMP Transport	440
20.2.3. The HA HttpInvoker - Clustered RMI/HTTP Transport	440
20.3. THE EJB CONTAINER	442
20.3.1. EJBDeployer MBean	442
20.3.1.1. Verifying EJB deployments	443
20.3.1.2. Deploying EJBs Into Containers	443
20.3.1.3. Container configuration information	444
20.3.1.3.1. The container-name element	448
20.3.1.3.2. The call-logging element	449
20.3.1.3.3. The invoker-proxy-binding-name element	449
20.3.1.3.4. The sync-on-commit-only element	449
20.3.1.3.5. insert-after-ejb-post-create	449
20.3.1.3.6. call-ejb-store-on-clean	449
20.3.1.3.7. The container-interceptors Element	449
20.3.1.3.8. The instance-pool element	449
20.3.1.3.9. The container-pool-conf element	449
20.3.1.3.10. The instance-cache element	450
20.3.1.3.11. The container-cache-conf element	450
20.3.1.3.12. The persistence-manager element	452
20.3.1.3.13. The web-class-loader Element	452
20.3.1.3.14. The locking-policy element	453
20.3.1.3.15. The commit-option and optiond-refresh-rate elements	453
20.3.1.3.16. The security-domain element	454
20.3.1.3.17. cluster-config	454
20.3.1.3.18. The depends element	455
20.3.2. Container Plug-in Framework	455
20.3.2.1. org.jboss.ejb.ContainerPlugin	456
20.3.2.2. org.jboss.ejb.Interceptor	456
20.3.2.3. org.jboss.ejb.InstancePool	457
20.3.2.4. org.jboss.ebj.InstanceCache	458
20.3.2.5. org.jboss.ejb.EntityPersistenceManager	459
20.3.2.6. The org.jboss.ejb.EntityPersistenceStore interface	462
20.3.2.7. org.jboss.ejb.StatefulSessionPersistenceManager	466
20.4. ENTITY BEAN LOCKING AND DEADLOCK DETECTION	467
20.4.1. Why JBoss Needs Locking	467
20.4.2. Entity Bean Lifecycle	467
20.4.3. Default Locking Behavior	468
20.4.4. Pluggable Interceptors and Locking Policy	468
20.4.5. Deadlock	469
20.4.5.1. Deadlock Detection	469
20.4.5.2. Catching ApplicationDeadlockException	470
20.4.5.3. Viewing Lock Information	470
20.4.6. Advanced Configurations and Optimizations	471
20.4.6.1. Short-lived Transactions	471
20.4.6.2. Ordered Access	471
20.4.6.3. Read-Only Beans	471
20.4.6.4. Explicitly Defining Read-Only Methods	472
20.4.6.5. Instance Per Transaction Policy	472
20.4.7. Running Within a Cluster	473
20.4.8. Troubleshooting	473

20.4.8.1. Locking Behavior Not Working	473
20.4.8.2. IllegalStateException	474
20.4.8.3. Hangs and Transaction Timeouts	474
20.5. EJB TIMER CONFIGURATION	474
CHAPTER 21. THE CMP ENGINE	477
21.1. EXAMPLE CODE	477
21.1.1. Enabling CMP Debug Logging	478
21.1.2. Running the examples	479
21.2. THE JBOSSCMP-JDBC STRUCTURE	480
21.3. ENTITY BEANS	482
21.3.1. Entity Mapping	484
21.4. CMP FIELDS	488
21.4.1. CMP Field Declaration	489
21.4.2. CMP Field Column Mapping	489
21.4.3. Read-only Fields	491
21.4.4. Auditing Entity Access	492
21.4.5. Dependent Value Classes (DVCs)	493
21.5. CONTAINER MANAGED RELATIONSHIPS	497
21.5.1. CMR-Field Abstract Accessors	497
21.5.2. Relationship Declaration	498
21.5.3. Relationship Mapping	499
21.5.3.1. Relationship Role Mapping	500
21.5.3.2. Foreign Key Mapping	503
21.5.3.3. Relation table Mapping	503
21.6. QUERIES	505
21.6.1. Finder and select Declaration	505
21.6.2. EJB-QL Declaration	506
21.6.3. Overriding the EJB-QL to SQL Mapping	507
21.6.4. JBossQL	508
21.6.5. DynamicQL	510
21.6.6. DeclaredSQL	511
21.6.6.1. Parameters	514
21.6.7. EJBQL 2.1 and SQL92 queries	514
21.6.8. BMP Custom Finders	515
21.7. OPTIMIZED LOADING	516
21.7.1. Loading Scenario	516
21.7.2. Load Groups	517
21.7.3. Read-ahead	518
21.7.3.1. on-find	518
21.7.3.1.1. Left join read ahead	520
21.7.3.1.2. D#findByPrimaryKey	520
21.7.3.1.3. D#findAll	521
21.7.3.1.4. A#findAll	521
21.7.3.1.5. A#findMeParentGrandParent	522
21.7.3.2. on-load	523
21.7.3.3. none	525
21.8. LOADING PROCESS	525
21.8.1. Commit Options	525
21.8.2. Eager-loading Process	526
21.8.3. Lazy loading Process	527
21.8.3.1. Relationships	528
21.8.4. Lazy loading result sets	531

21.9. TRANSACTIONS	531
21.10. OPTIMISTIC LOCKING	534
21.11. ENTITY COMMANDS AND PRIMARY KEY GENERATION	538
21.11.1. Existing Entity Commands	538
21.12. DEFAULTS	541
21.12.1. A sample jbosscmp-jdbc.xml defaults declaration	543
21.13. DATASOURCE CUSTOMIZATION	544
21.13.1. Type Mapping	545
21.13.2. Function Mapping	547
21.13.3. Mapping	548
21.13.4. User Type Mappings	549
APPENDIX A. BOOK EXAMPLE INSTALLATION	551
APPENDIX B. USE ALTERNATIVE DATABASES WITH JBOSS AS	552
B.1. HOW TO USE ALTERNATIVE DATABASES	552
B.2. INSTALL JDBC DRIVERS	552
B.2.1. Special notes on Sybase	553
B.3. CREATING A DATASOURCE FOR THE EXTERNAL DATABASE	554
B.4. CHANGE DATABASE FOR THE JMS SERVICES	555
B.5. SUPPORT FOREIGN KEYS IN CMP SERVICES	555
B.6. SPECIFY DATABASE DIALECT FOR JAVA PERSISTENCE API	556
B.7. CHANGE OTHER JBOSS AS SERVICES TO USE THE EXTERNAL DATABASE	556
B.7.1. The Easy Way	556
B.7.2. The More Flexible Way	557
B.8. A SPECIAL NOTE ABOUT ORACLE DATABASES	558
APPENDIX C. VENDOR-SPECIFIC DATASOURCE DEFINITIONS	560
C.1. DEPLOYER LOCATION AND NAMING	560
C.2. DB2	560
C.3. ORACLE	564
C.3.1. Changes in Oracle 10g JDBC Driver	567
C.3.2. Type Mapping for Oracle 10g	567
C.3.3. Retrieving the Underlying Oracle Connection Object	567
C.4. SYBASE	567
C.5. MICROSOFT SQL SERVER	568
C.5.1. Microsoft JDBC Drivers	570
C.5.2. JSQL Drivers	572
C.5.3. jTDS JDBC Driver	572
C.5.4. "Invalid object name 'JMS_SUBSCRIPTIONS' Exception	574
C.6. MYSQL DATASOURCE	574
C.6.1. Installing the Driver	574
C.6.2. MySQL Local-TX Datasource	575
C.6.3. MySQL Using a Named Pipe	575
C.7. POSTGRESQL	576
C.8. INGRES	577
APPENDIX D. REVISION HISTORY	579

PREFACE

1. WHAT THIS BOOK COVERS

The primary focus of this book is the presentation of the standard JBoss Enterprise Application Platform 4.3 architecture components from both the perspective of their configuration and architecture. As a user of a standard JBoss distribution you will be given an understanding of how to configure the standard components. Note that this book is not an introduction to J2EE or how to use J2EE in applications. It focuses on the internal details of the JBoss server architecture and how our implementation of a given J2EE container can be configured and extended.

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.

PART I. JAVA EE 5 APPLICATION CONFIGURATION

CHAPTER 1. ENTERPRISE APPLICATIONS WITH EJB3 SERVICES

EJB3 (Enterprise JavaBean 3.0) provides the core component model for Java EE 5 applications. An EJB3 bean is a managed component that is automatically wired to take advantage of all services the J2EE server container provides, such as transaction, security, persistence, naming, dependency injection, etc. The managed component allows developers to focus on the business logic, and leave the cross-cutting concerns to the container as configurations. As an application developer, you need not create or destroy the components yourself. You only need to ask for an EJB3 bean from the Java EE container by its name, and then you can call its methods with all configured container services applied. You can get access to an EJB3 bean from either inside or outside of the J2EE container.

JBoss Enterprise Application Platform 4.3 supports EJB3 out of the box.

The details of the EJB3 component programming model is beyond the scope of this guide. Most EJB3 interfaces and annotations are part of the Java EE 5 standard and hence they are the same for all Java EE 5 compliant application servers. Interested readers should refer to the EJB3 specification or numerous EJB3 books to learn more about EJB3 programming.

In this chapter, we only cover EJB3 configuration issues that are specific to the JBoss AS. For instance, we discuss the JNDI naming conventions for EJB3 components inside the JBoss AS, the optional configurations for the Hibernate persistence engine for entity beans, as well as custom options in the JBoss EJB3 deployer.

1.1. SESSION BEANS

Session beans are widely used to provide transactional services for local and remote clients. To write a session bean, you need an interface and an implementation class.

```
@Local
public interface MyBeanInt {
    public String doSomething (String para1, int para2);
}

@Stateless
public class MyBean implements MyBeanInt {

    public String doSomething (String para1, int para2) {
        ... implement the logic ...
    }

}
```

When you invoke a session bean method, the method execution is automatically managed by the transaction manager and the security manager in the server. You can specify the transactional or security properties for each method using annotations on the method. A session bean instance can be reused by many clients. Depending on whether the server maintains the bean's internal state between two clients, the session bean can be stateless or stateful. Depending on whether the bean is available to remote clients (i.e., clients outside of the current JVM for the server), the session bean can be local or remote. All these are configurable via standard annotations on the beans.

After you define a session bean, how does the client get access to it? As we discussed, the client does not create or destroy EJB3 components, it merely asks the server for a reference of an existing instance

managed by the server. That is done via JNDI. In JBoss AS, the default local JNDI name for a session bean is dependent on the deployment packaging of the bean class.

- If the bean is deployed in a standalone JAR file in the **jboss-as/production/deploy** directory, the bean is accessible via local JNDI name **MyBean/local**, where **MyBean** is the implementation class name of the bean as we showed earlier. The "local" JNDI in JBoss AS means that the JNDI name is relative to **java:comp/env/**.
- If the JAR file containing the bean is packaged in an EAR file, the local JNDI name for the bean is **myapp/MyBean/local**, where **myapp** is the root name of the EAR archive file (e.g., **myapp.ear**, see later for the EAR packaging of EJB3 beans).

Of course, you should change **local** to **remote** if the bean interface is annotated with **@Remote** and the bean is accessed from outside of the server it is deployed on. Below is the code snippet to get a reference of the **MyBean** bean in a web application (e.g., in a servlet or a JSF backing bean) packaged in **myapp.ear**, and then invoke a managed method.

```
try {
    InitialContext ctx = new InitialContext();
    MyBeanInt bean = (MyBeanInt) ctx.lookup("myapp/MyBean/local");
} catch (Exception e) {
    e.printStackTrace ();
}

... ..

String result = bean.doSomething("have fun", 1);

... ..
```

What the client gets from the JNDI is essentially a "stub" or "proxy" of the bean instance. When the client invokes a method, the proxy figures out how to route the request to the server and marshal together the response.

If you do not like the default JNDI names, you can always specify your own JNDI binding for any bean via the **@LocalBinding** annotation on the bean implementation class. The JNDI binding is always "local" under the **java:comp/env/** space. For instance, the following bean class definition results in the bean instances available under JNDI name **java:comp/env/MyService/MyOwnName**.

```
@Stateless
@LocalBinding (jndiBinding="MyService/MyOwnName")
public class MyBean implements MyBeanInt {

    public String doSomething (String para1, int para2) {
        ... implement the logic ...
    }

}
```

**NOTE**

Java EE 5 allows you to inject EJB3 bean instances directly into the web application via annotations without explicit JNDI lookup. This behavior is not yet supported in JBoss AS 4.2. However, JBoss Enterprise Application Platform provides an integration framework called JBoss Seam. JBoss Seam brings EJB3 / JSF integration to new heights far beyond what Java EE 5 provides. Please see more details in the JBoss Seam reference guide bundled with the platform.

1.2. ENTITY BEANS (A.K.A. JAVA PERSISTENCE API)

EJB3 session beans allow you to implement data accessing business logic in transactional methods. To actually access the database, you will need EJB3 entity beans and the entity manager API. They are collectively called the Java Persistence API (JPA).

EJB3 Entity Beans are Plain Old Java Objects (POJOs) that map to relational database tables. For instance, the following entity bean class maps to a relational table named customer. The table has three columns: name, age, and signupdate. Each instance of the bean corresponds to a row of data in the table.

```
@Entity
public class Customer {

    String name;

    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    int age;

    public int getAge () {
        return age;
    }

    public void setAge (int age) {
        this.age = age;
    }

    Date signupdate;

    public Date getSignupdate () {
        return signupdate;
    }

    public void setSignupdate (Date signupdate) {
        this.signupdate = signupdate;
    }
}
```

Besides simple data properties, the entity bean can also contain references to other entity beans with relational mapping annotations such as `@OneToOne`, `@OneToMany`, `@ManyToMany` etc. The relationships of those entity objects will be automatically set up in the database as foreign keys. For instance, the following example shows that each record in the Customer table has one corresponding record in the Account table, multiple corresponding records in the Order table, and each record in the Employee table has multiple corresponding records in the Customer table.

```
@Entity
public class Customer {

    ... ..

    Account account;

    @OneToOne
    public Account getAccount () {
        return account;
    }

    public void setAccount (Account account) {
        this.account = account;
    }

    Employee salesRep;

    @ManyToOne
    public Employee getSalesRep () {
        return salesRep;
    }

    public void setSalesRep (Employee salesRep) {
        this.salesRep = salesRep;
    }

    Vector <Order> orders;

    @OneToMany
    public Vector <Order> getOrders () {
        return orders;
    }

    public void setOrders (Vector <Order> orders) {
        this.orders = orders;
    }
}
```

Using the EntityManager API, you can create, update, delete, and query entity objects. The EntityManager transparently updates the underlying database tables in the process. You can obtain an EntityManager object in your EJB3 session bean via the `@PersistenceContext` annotation.

```
@PersistenceContext
EntityManager em;
```

```
Customer customer = new Customer ();
// populate data in customer

// Save the newly created customer object to DB
em.persist (customer);

// Increase age by 1 and auto save to database
customer.setAge (customer.getAge() + 1);

// delete the customer and its related objects from the DB
em.remove (customer);

// Get all customer records with age > 30 from the DB
List <Customer> customers = em.query (
    "select c from Customer where c.age > 30");
```

The detailed use of the EntityManager API is beyond the scope of this book. Interested readers should refer to the JPA documentation or Hibernate EntityManager documentation.

1.2.1. The persistence.xml file

The EntityManager API is great, but how does the server know which database it is supposed to save / update / query the entity objects? How do we configure the underlying object-relational-mapping engine and cache for better performance and trouble shooting? The persistence.xml file gives you complete flexibility to configure the EntityManager.

The persistence.xml file is a standard configuration file in JPA. It has to be included in the META-INF directory inside the JAR file that contains the entity beans. The persistence.xml file must define a persistence-unit with a unique name in the current scoped classloader. The provider attribute specifies the underlying implementation of the JPA EntityManager. In JBoss AS, the default and only supported / recommended JPA provider is Hibernate. The jta-data-source points to the JNDI name of the database this persistence unit maps to. The java:/DefaultDS here points to the HSQL DB embedded in the JBoss AS. Please refer to [Appendix B, Use Alternative Databases with JBoss AS](#) on how to setup alternative databases for JBoss AS.

```
<persistence>
  <persistence-unit name="myapp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      ... ..
    </properties>
  </persistence-unit>
</persistence>
```



NOTE

Since you might have multiple instances of persistence-unit defined in the same application, you typically need to explicitly tell the `@PersistenceContext` annotation which unit you want to inject. For instance, `@PersistenceContext(name="myapp")` injects the `EntityManager` from the persistence-unit named "myapp".

However, if you deploy your EAR application in its own scoped classloader and have only one persistence-unit defined in the whole application, you can omit the "name" on `@PersistenceContext`. See later in this chapter for EAR packaging and deployment.

The `properties` element in the `persistence.xml` can contain any configuration properties for the underlying persistence provider. Since JBoss AS uses Hibernate as the EJB3 persistence provider, you can pass in any Hibernate options here. Please refer to the Hibernate and Hibernate `EntityManager` documentation for more details. Here we will just give an example to set the SQL dialect of the persistence engine to HSQL, and to create tables from the entity beans when the application starts and drop those tables when the application stops.

```
<persistence>
  <persistence-unit name="myapp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

1.2.2. Use Alternative Databases

To use an alternative database other than the built-in HSQL DB to back your entity beans, you need to first define the data source for the database and register it in the JNDI. This is done via the `*-ds.xml` files in the deploy directory. Please see [Section 7.3, "Configuring JDBC DataSources"](#) for more details. Examples of `*-ds.xml` files for various databases are available in `jboss-as/docs/examples/jca` directory in the server.

Then, in the `persistence.xml`, you need to change the `jta-data-source` attribute to point to the new data source in JNDI (e.g., `java:/MySQLDS` if you are using the default `mysql-ds.xml` to setup a MySQL external database).

In most cases, Hibernate tries to automatically detect the database it connects to and then automatically selects an appropriate SQL dialect for the database. However, we have found that this detection does not always work, especially for less used database servers. We recommend you to set the `hibernate.dialect` property explicitly in `persistence.xml`. Here are the Hibernate dialect for database servers officially supported on the JBoss platform.

- Oracle 9i and 10g: `org.hibernate.dialect.Oracle9Dialect`
- Microsoft SQL Server 2005: `org.hibernate.dialect.SQLServerDialect`
- PostgreSQL 8.1: `org.hibernate.dialect.PostgreSQLDialect`

- MySQL 5.0: org.hibernate.dialect.MySQL5Dialect
- DB2 8.0: org.hibernate.dialect.DB2Dialect
- Sybase ASE 15: org.hibernate.dialect.SybaseASE15Dialect



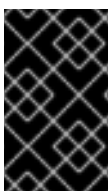
NOTE

The Sybase ASE 12.5 org.hibernate.dialect.SybaseDialect dialect is now considered deprecated; however support is still maintained for users who do not wish to move to the new dialect.

1.2.3. Default Hibernate options

Hibernate has many configuration properties. For the properties that you do not specify in the persistence.xml file, JBoss AS will provide a reasonable set of default values. The default Hibernate property values are specified in the **jboss-as/server/production/deploy/ejb3.deployer/META-INF/persistence.properties** file. Below is the **persistence.properties** file bundled in JBoss Enterprise Application Platform. Notice the options that are commented out. They give you an idea of available properties in your **persistence.xml** file.

```
hibernate.transaction.manager_lookup_class=org.hibernate.transaction.JBoss
TransactionManagerLookup
#hibernate.connection.release_mode=after_statement
#hibernate.transaction.flush_before_completion=false
#hibernate.transaction.auto_close_session=false
#hibernate.query.factory_class=org.hibernate.hql.ast.ASTQueryTranslatorFac
tory
#hibernate.hbm2ddl.auto=create-drop
#hibernate.hbm2ddl.auto=create
hibernate.cache.provider_class=org.hibernate.cache.HashtableCacheProvider
# Clustered cache with TreeCache
#hibernate.cache.provider_class=org.jboss.ejb3.entity.TreeCacheProviderHoo
k
#hibernate.treecache.mbean.object_name=jboss.cache:service=EJB3EntityTreeC
ache
#hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.jndi.java.naming.factory.initial=org.jnp.interfaces.NamingContex
tFactory
hibernate.jndi.java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.inter
faces
hibernate.bytecode.use_reflection_optimizer=false
# I don't think this is honored, but EJB3Deployer uses it
hibernate.bytecode.provider=javassist
```



IMPORTANT

org.hibernate.dialect.HSQLDialect is considered deprecated, replacyed with org.hibernate.dialect.SybaseASE15Dialect. The above example has been left unmodified in order to maintain example cohesion.

1.3. MESSAGE DRIVEN BEANS

Messaging driven beans are specialized EJB3 beans that receive service requests via JMS messages instead of proxy method calls from the "stub". So, a crucial configuration parameter for the message driven bean is to specify which JMS message queue it listens to. When there is an incoming message in the queue, the server invokes the bean's **onMessage()** method, and passes in the message itself for processing. The bean class specifies the JMS queue it listens to in the `@MessageDriven` annotation. The queue is registered under the local JNDI `java:comp/env/` name space.

```
@MessageDriven(activationConfig =
{
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/MyQueue")
})
public class MyJmsBean implements MessageListener {

    public void onMessage (Message msg) {
        // ... do something with the msg ...
    }

    // ... ...
}
```

When a message driven bean is deployed, its incoming message queue is automatically created if it does not exist already. To send a message to the bean, you can use the standard JMS API.

```
try {
    InitialContext ctx = new InitialContext();
    queue = (Queue) ctx.lookup("queue/MyQueue");
    QueueConnectionFactory factory =
        (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    cnn = factory.createQueueConnection();
    sess = cnn.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);

} catch (Exception e) {
    e.printStackTrace ();
}

TextMessage msg = sess.createTextMessage(...);

sender = sess.createSender(queue);
sender.send(msg);
```

Please refer to the JMS specification or books to learn how to program in the JMS API.

1.4. PACKAGE AND DEPLOY EJB3 SERVICES

EJB3 bean classes are packaged in regular JAR files. The standard configuration files, such as `ejb-jar.xml` for session beans, and `persistence.xml` for entity beans, are in the `META-INF` directory inside the JAR. You can deploy EJB3 beans as standalone services in JBoss AS or as part of an enterprise

application (i.e., in an EAR archive). In this section, we discuss those two deployment options.

1.4.1. Deploy the EJB3 JAR

When you drop JAR files into the `jboss-as/server/production/deploy/` directory, it will be automatically picked up and processed by the server. All the EJB3 beans defined in the JAR file will then be available to other applications deployed inside or outside of the server via JNDI names like `MyBean/local`, where `MyBean` is the implementation class name for the session bean. The deployment is done via the JBoss EJB3 deployer in `jboss-as/server/production/ejb3.deployer/`. The `META-INF/persistence.properties` file we discussed earlier to configure the default behavior of EJB3 entity manager is located in the EJB3 deployer.

The EJB3 deployer automatically scans JARs on the classpath to look for EJB3 annotations. When it finds classes with EJB3 annotations, it would deploy them as EJB3 services. However, scanning all JARs on the classpath could be very time-consuming if you have large applications with many JARs deployed. In the `jboss-as/server/production/ejb3.deployer/META-INF/jboss-service.xml` file, you can tell the EJB3 deployer to ignore JARs you know do not contain EJB3 beans. The non-EJB3 JAR files shipped with the JBoss AS are already listed in the `jboss.ejb3:service=JarsIgnoredForScanning` MBean service:

```
... ..
<mbean code="org.jboss.ejb3.JarsIgnoredForScanning"
      name="jboss.ejb3:service=JarsIgnoredForScanning">
  <attribute name="IgnoredJars">
    snmp-adaptor.jar,
    otherimages.jar,
    applet.jar,
    jcommon.jar,
    console-mgr-classes.jar,
    jfreechart.jar,
    juddi-service.jar,
    wsd14j.jar,
    ... ..
    servlets-webdav.jar
  </attribute>
</mbean>
... ..
```

You can add any non-EJB3 JARs from your application to this list so that the server do not have to waste time scanning them. This could significantly improve the application startup time in some cases.

1.4.2. Deploy EAR with EJB3 JAR

Most Java EE applications are deployed as EAR archives. An EAR archive is a JAR file that typically contains a WAR archive for the web pages, servlets, and other web-related components, one or several EJB3 JARs that provide services (e.g., data access and transaction) to the WAR components, and some other support library JARs required by the application. An EAR file also have deployment descriptors such as `application.xml` and `jboss-app.xml`. Below is the basic structure of a typical EAR application.

```
myapp.ear
|+ META-INF
|+ applications.xml and jboss-app.xml
```

```

|+ myapp.war
|+ web pages and JSP /JSF pages
|+ WEB-INF
|+ web.xml, jboss-web.xml, faces-config.xml etc.
|+ lib
|+ tag library JARs
|+ classes
|+ servlets and other classes used by web pages
|+ myapp.jar
|+ EJB3 bean classes
|+ META-INF
|+ ejb-jar.xml and persistence.xml
|+ lib
|+ Library JARs for the EAR

```

Notice that in JBoss AS, unlike in many other application servers, you do not need to declare EJB references in the web.xml file in order for the components in the WAR file to access EJB3 services. You can obtain the references directly via JNDI as we discussed earlier in the chapter.

A typical application.xml file is as follows. It declares the WAR and EJB3 JAR archives in the EAR, and defines the web content root for the application. Of course, you can have multiple EJB3 modules in the same EAR application. The application.xml file could also optionally define a shared classpath for JAR files used in this application. The JAR file location defaults to lib in JBoss AS -- but it might be different in other application servers.

```

<application>
  <display-name>My Application</display-name>

  <module>
    <web>
      <web-uri>myapp.war</web-uri>
      <context-root>/myapp</context-root>
    </web>
  </module>

  <module>
    <ejb>myapp.jar</ejb>
  </module>

  <library-directory>lib</library-directory>

</application>

```

The jboss-app.xml file provides JBoss-specific deployment configuration for the EAR application. For instance, it can specify the deployment order of modules in the EAR, deploy JBoss-specific application modules in the EAR, such as SARs (Service ARchive for MBeans) and HARs (Hibernate ARchive for Hibernate objects), provide security domain and JMX MBeans that can be used with this application, etc. You can learn more about the possible attributes in jboss-app.xml in its DTD: http://www.jboss.org/j2ee/dtd/jboss-app_4_2.dtd.

A common use case for jboss-app.xml is to configure whether this EAR file should be deployed in its own scoped classloader to avoid naming conflicts with other applications. If your EAR application is deployed in its own scoped classloader and it only has one persistence-unit defined in its EJB3 JARs, you will be

able to use `@PersistenceContext EntityManager em` to inject `EntityManager` to session beans without worrying about passing the persistence unit name to the `@PersistenceContext` annotation. The following `jboss-app.xml` specifies a scoped classloader `myapp:archive=myapp.ear` for the EAR application.

```
<jboss-app>
  <loader-repository>
    myapp:archive=myapp.ear
  </loader-repository>
</jboss-app>
```

The EAR deployment is configured by the `jboss-as/server/production/deploy/ear-deploy.xml` file. This file contains three attributes as follows.

```
<server>
  <mbean code="org.jboss.deployment.EARDeployer"
    name="jboss.j2ee:service=EARDeployer">
    <!--
      A flag indicating if ear deployments should
      have their own scoped class loader to isolate
      their classes from other deployments.
    -->
    <attribute name="Isolated">false</attribute>

    <!--
      A flag indicating if the ear components should
      have in VM call optimization disabled.
    -->
    <attribute name="CallByValue">false</attribute>

    <!--
      A flag the enables the default behavior of
      the ee5 library-directory. If true, the lib
      contents of an ear are assumed to be the default
      value for library-directory in the absence of
      an explicit library-directory. If false, there
      must be an explicit library-directory.
    -->
    <attribute name="EnablelibDirectoryByDefault">true</attribute>
  </mbean>
</server>
```

If you set the `Isolated` parameter to `true`, all EAR deployment will have scoped classloaders by default. There will be no need to define the classloader in `jboss-app.xml`. The `CallByValue` attribute specifies whether we should treat all EJB calls as remote calls. Remote calls have a large additional performance penalty compared with local call-by-reference calls, because objects involved in remote calls have to be serialized and de-serialized. For most of our applications, the WAR and EJB3 JARs are deployed on the same server, hence this value should be default to `false` and the server uses local call-by-reference calls to invoke EJB methods in the same JVM. The `EnablelibDirectoryByDefault` attribute specifies whether the `lib` directory in the EAR archive should be the default location for shared library JARs.

CHAPTER 2. NETWORK

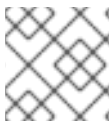
By default, the Enterprise Platform binds to all networking addresses at start-up. You can specify a bind address, as well as a UDP address, at start-up.

Table 2.1. Network-related start-up options

-b [IP-ADDRESS]	Specifies the address the application server binds to. If unspecified, the application server binds to all addresses.
-u [IP-ADDRESS]	UDP multicast address. Optional. If not specified, only TCP is used.
-m [MULTICAST_PORT_ADDRESSES]	UDP multicast port. Only used by JGroups.

2.1. ENABLING IPV6

JBoss Enterprise Application Platform supports IPv4 and IPv6 networking on the application server. The default configuration uses IPv4. To enable IPv6, you need to follow [Procedure 2.1, “Enabling IPv6 networking in Linux / UNIX”](#) or [Procedure 2.2, “Enabling IPv6 networking in Windows”](#) depending on your operating system.



NOTE

If you need to use the Transaction Service, IPv6 networking is not supported.

Procedure 2.1. Enabling IPv6 networking in Linux / UNIX

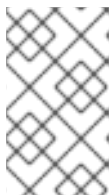
Before you enable IPv6 networking on your application server, enable IPv6 networking for your operating system.

To enable IPv6 networking on your application server, do the following:

1. **Edit `JBOSS_HOME/jboss-as/bin/run.sh`.**

Comment out the `java.net.preferIPv4Stack` parameter. In the default `run.sh` file, this is line 93:

```
92 # Force use of IPv4 stack
93 # JAVA_OPTS="$JAVA_OPTS -Djava.net.preferIPv4Stack=true"
```



NOTE

You cannot pass the `java.net.preferIPv4Stack` variable at the command line because it is a Java parameter, not a parameter to the application server itself. Therefore, you must change it in the `run.sh` file.

2. **Remove JBoss Transactions from your profile.**

Do not deploy the Transaction Service into your Application Server. To prevent the deployment of the Transaction Service, remove the following deployers:

- o `JBOSS_HOME/jboss-as/conf/PROFILE/deploy/transaction-service.xml`

- **`JBOSS_HOME/jboss-as/conf/PROFILE/deploy/transaction-jboss-beans.xml`**
- **`JBOSS_HOME/jboss-as/conf/PROFILE/conf/jbossts-properties.xml`**

3. Bind the application server to your system's IPv6 address.

Specify the IPv6 address the application server should bind to. Enclose the address in square brackets ([]):

```
./run.sh -b [IPv6 address] -u [IPv6 mcast address]
```

Procedure 2.2. Enabling IPv6 networking in Windows

Before you enable IPv6 networking on your application server, enable IPv6 networking in your operating system.

To enable IPv6 networking on your application server, do the following:

1. Edit `JBOSS_HOME/jboss-as/bin/run.bat`.

Comment out the **`java.net.preferIPv4Stack`** parameter. In the default **`run.bat`** file, this is line 109:

```
108 rem Force use of IPv4 stack
109 rem set JAVA_OPTS=%JAVA_OPTS% -Djava.net.preferIPv4Stack=true
```



NOTE

You cannot pass the **`java.net.preferIPv4Stack`** variable at the command line because it is a Java parameter, not a parameter to the application server itself. Therefore, you must change it in the **`run.bat`** file.

2. Remove JBoss Transactions from your profile.

Do not deploy the Transaction Service into your Application Server. To prevent the deployment of the Transaction Service, remove the following deployers:

- **`JBOSS_HOME\jboss-as\conf\PROFILE\deploy\transaction-service.xml`**
- **`JBOSS_HOME\jboss-as\conf\PROFILE\deploy\transaction-jboss-beans.xml`**
- **`JBOSS_HOME\jboss-as\conf\PROFILE\conf\jbossts-properties.xml`**

3. Bind the application server to your system's IPv6 address.

Specify the IPv6 address the application server should bind to. Enclose the address in square brackets ([]):

```
run.bat -b [IPv6 address] -u [IPv6 mcast address]
```

CHAPTER 3. DATASOURCE CONFIGURATION



WARNING

The default persistence configuration works out of the box with Hypersonic (HSQLDB) so that the JBoss Enterprise Platforms are able to run "out of the box". However, *Hypersonic is not supported in production and should not be used in a production environment.*

Known issues with the Hypersonic Database include:

- no transaction isolation
- thread and socket leaks (**connection.close()** does not tidy up resources)
- persistence quality (logs commonly become corrupted after a failure, preventing automatic recovery)
- database corruption
- stability under load (database processes cease when dealing with too much data)
- not viable in clustered environments

Check the "Using Other Databases" chapter of the *Getting Started Guide* for assistance.

Datasources are defined inside a `<datasources>` element. The exact element depends on the type of datasource required.

3.1. TYPES OF DATASOURCES

Datasource Definitions

`<no-tx-datasource>`

Does not take part in JTA transactions. The **java.sql.Driver** is used.

`<local-tx-datasource>`

Does not support two phase commit. The **java.sql.Driver** is used. Suitable for a single database or a non-XA-aware resource.

`<xa-datasource>`

Supports two phase commit. The **javax.sql.XADataSource** driver is used.

3.2. DATASOURCE PARAMETERS

Common Datasource Parameters

<jndi-name>

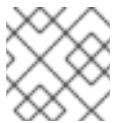
The JNDI name under which the Datasource should be bound.

<use-java-context>

Boolean value indicating whether the jndi-name should be prefixed with *java:*. This prefix causes the Datasource to only be accessible from within the JBoss Enterprise Application Platform virtual machine. Defaults to **TRUE**.

<user-name>

The user name used to create the connection to the datasource.

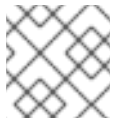


NOTE

Not used when security is configured.

<password>

The password used to create the connection to the datasource.



NOTE

Not used when security is configured.

<transaction-isolation>

The default transaction isolation of the connection. If not specified, the database-provided default is used.

Possible values for <transaction-isolation>

- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE
- TRANSACTION_NONE

<new-connection-sql>

An SQL statement that is executed against each new connection. This can be used to set up the connection schema, for instance.

<check-valid-connection-sql>

An SQL statement that is executed before the connection is checked out from the pool to make sure it is still valid. If the SQL statement fails, the connection is closed and a new one is created.

<valid-connection-checker-class-name>

A class that checks whether a connection is valid using a vendor-specific mechanism.

<exception-sorter-class-name>

A class that parses vendor-specific messages to determine whether SQL errors are fatal, and destroys the connection if so. If empty, no errors are treated as fatal.

<track-statements>

Whether to monitor for unclosed Statements and ResultSets and issue warnings when they haven't been closed. The default value is **NOWARN**.

<prepared-statement-cache-size>

The number of prepared statements per connection to be kept open and reused in subsequent requests. They are stored in a *Least Recently Used (LRU)* cache. The default value is **0**, meaning that no cache is kept.

<share-prepared-statements>

When the <prepared-statement-cache-size> is non-zero, determines whether two requests in the same transaction should return the same statement. Defaults to **FALSE**.

Example 3.1. Using <share-prepared-statements>

The goal is to work around questionable driver behavior, where the driver applies auto-commit semantics to local transactions.

```

false      Connection c = dataSource.getConnection(); // auto-commit ==
            PreparedStatement ps1 = c.prepareStatement(...);
            ResultSet rs1 = ps1.executeQuery();
            PreparedStatement ps2 = c.prepareStatement(...);
            ResultSet rs2 = ps2.executeQuery();

```

This assumes that the prepared statements are the same. For some drivers, **ps2.executeQuery()** automatically closes **rs1**, so you actually need two real prepared statements behind the scenes. This only applies to the auto-commit semantic, where re-running the query starts a new transaction automatically. For drivers that follow the specification, you can set it to **TRUE** to share the same real prepared statement.

<set-tx-query-timeout>

Whether to enable query timeout based on the length of time remaining until the transaction times out. Included for future compatible only, and currently has no effect. Defaults to **FALSE**.

<query-timeout>

The maximum time, in seconds, before a query times out. You can override this value by setting <set-tx-query-timeout> to **TRUE**.

<metadata>><type-mapping>

A pointer to the type mapping in **conf/standardjbosscmp.xml**. A legacy from JBoss4.

<validate-on-match>

Whether to validate the connection when the JCA layer matches a managed connection, such as when the connection is checked out of the pool. With the addition of **<background-validation>** this is not required. It is usually not necessary to specify **TRUE** for **<validate-on-match>** in conjunction with specifying **TRUE** for **<background-validation>**. Defaults to **TRUE**.

<prefill>

Whether to attempt to prefill the connection pool to the minimum number of connections. Only *supporting pools* (OnePool) support this feature. A warning is logged if the pool does not support prefilling. Defaults to **TRUE**.

<background-validation>

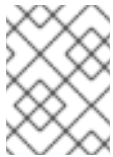
Background connection validation reduces the overall load on the RDBMS system when validating a connection. When using this feature, EAP checks whether the current connection in the pool a separate thread (ConnectionValidator). **<background-validation-minutes>** depends on this value also being set to **TRUE**. Defaults to **FALSE**.

<idle-timeout-minutes>

The maximum time, in minutes, before an idle connection is closed. A value of **0** disables timeout. Defaults to **15** minutes.

<background-validation-minutes>

How often, in minutes, the ConnectionValidator runs. Defaults to **10** minutes.

**NOTE**

You should set this to a small value than **<idle-timeout-minutes>**, unless you have specified **<min-pool-size>** a minimum pool size set.

<url-delimiter>, <url-property>, <url-selector-strategy-class-name>

Parameters dealing with database failover. As of JBoss Enterprise Application Platform 5.1, these are configured as part of the main datasource configuration. In previous versions, **<url-delimiter>** appeared as **<url-delimeter>**.

<stale-connection-checker-class-name>

An implementation of **org.jboss.resource.adapter.jdbc.StateConnectionChecker** that decides whether **SQLExceptions** that notify of bad connections throw the **org.jboss.resource.adapter.jdbc.StateConnectionException** exception.

<max-pool-size>

The maximum number of connections allowed in the pool. Defaults to **20**.

<min-pool-size>

The minimum number of connections maintained in the pool. Unless **<prefill>** is **TRUE**, the pool remains empty until the first use, at which point the pool is filled to the **<min-pool-size>**. When the pool size drops below the **<min-pool-size>** due to idle timeouts, the pool is refilled to the **<min-pool-size>**. Defaults to **0**.

<blocking-timeout-millis>

The length of time, in milliseconds, to wait for a connection to become available when all the connections are checked out. Defaults to **30000**, which is 30 seconds.

<use-fast-fail>

Whether to continue trying to acquire a connection from the pool even if the previous attempt has failed, or begin failover. This is to address performance issues where validation SQL takes significant time and resources to execute. Defaults to **FALSE**.

Parameters for `javax.sql.XADataSource` Usage**<connection-url>**

The JDBC driver connection URL string

<driver-class>

The JDBC driver class implementing the `java.sql.Driver`

<connection-property>

Used to configure the connections retrieved from the `java.sql.Driver`.

Example 3.2. Example <connection-property>

```
<connection-property name="char.encoding">UTF-8</connection-property>
```

Parameters for `javax.sql.XADataSource` Usage**<xa-datasource-class>**

The class implementing the `XADataSource`

<xa-datasource-property>

Properties used to configure the `XADataSource`.

Example 3.3. Example <xa-datasource-property> Declarations

```
<xa-datasource-property name="IfxWAITTIME">10</xa-datasource-property>
<xa-datasource-property name="IfxIFXHOST">myhost.mydomain.com</xa-datasource-property>
<xa-datasource-property name="PortNumber">1557</xa-datasource-property>
<xa-datasource-property name="DatabaseName">mydb</xa-datasource-property>
<xa-datasource-property name="ServerName">myserver</xa-datasource-property>
```

<isSameRM-override-value>

When set to **FALSE**, fixes some problems with Oracle databases.

<no-tx-separate-pools>

Pool transactional and non-transactional connections separately

**WARNING**

Using this option will cause your total pool size to be twice **max-pool-size**, because two actual pools will be created.

Used to fix problems with Oracle.

3.3. DATASOURCE EXAMPLES

For database-specific examples, see [Appendix C, Vendor-Specific Datasource Definitions](#).

3.3.1. Generic Datasource Example

Example 3.4. Generic Datasource Example

```
<datasources>
  <local-tx-datasource>
    <jndi-name>GenericDS</jndi-name>
    <connection-url>[jdbc: url for use with Driver class]</connection-
url>
    <driver-class>[fully qualified class name of java.sql.Driver
implementation]</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- you can include connection properties that will get passed in
the DriverManager.getConnection(props) call-->
    <!-- look at your Driver docs to see what these might be -->
    <connection-property name="char.encoding">UTF-8</connection-
property>
    <transaction-isolation>TRANSACTION_SERIALIZABLE</transaction-
isolation>

    <!--pooling parameters-->
    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->
```

```

    <!-- sql to call on an existing pooled connection when it is
    obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
    connection-sql>
    -->

    <set-tx-query-timeout></set-tx-query-timeout>
    <query-timeout>300</query-timeout> <!-- maximum of 5 minutes for
    queries -->

    <!-- pooling criteria.  USE AT MOST ONE-->
    <!-- If you don't use JAAS login modules or explicit login
    getConnection(usr,pw) but rely on user/pw specified above,
    don't specify anything here -->

    <!-- If you supply the usr/pw from a JAAS login module -->
    <security-domain>MyRealm</security-domain>

    <!-- if your app supplies the usr/pw explicitly getConnection(usr,
    pw) -->
    <application-managed-security></application-managed-security>

    <!--Anonymous depends elements are copied verbatim into the
    ConnectionManager mbean config-->
    <depends>myapp.service:service=DoSomethingService</depends>

</local-tx-datasource>

<!-- you can include regular mbean configurations like this one -->
<mbean code="org.jboss.tm.XidFactory"
name="jboss:service=XidFactory">
  <attribute name="Pad">true</attribute>
</mbean>

<!-- Here's an xa example -->
<xa-datasource>
  <jndi-name>GenericXADS</jndi-name>
  <xa-datasource-class>[fully qualified name of class implementing
  javax.sql.XADataSource goes here]</xa-datasource-class>
  <xa-datasource-property name="SomeProperty">SomePropertyValue</xa-
  datasource-property>
  <xa-datasource-property
  name="SomeOtherProperty">SomeOtherValue</xa-datasource-property>

  <user-name>x</user-name>
  <password>y</password>
  <transaction-isolation>TRANSACTION_SERIALIZABLE</transaction-
  isolation>

  <!--pooling parameters-->
  <min-pool-size>5</min-pool-size>
  <max-pool-size>100</max-pool-size>
  <blocking-timeout-millis>5000</blocking-timeout-millis>

```

```

<idle-timeout-minutes>15</idle-timeout-minutes>
<!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->

<!-- sql to call on an existing pooled connection when it is
obtained from pool
<check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
-->

<!-- pooling criteria.  USE AT MOST ONE-->
<!-- If you don't use JAAS login modules or explicit login
getConnection(usr,pw) but rely on user/pw specified above,
don't specify anything here -->

<!-- If you supply the usr/pw from a JAAS login module -->
<security-domain></security-domain>

<!-- if your app supplies the usr/pw explicitly getConnection(usr,
pw) -->
<application-managed-security></application-managed-security>

</xa-datasource>

</datasources>

```

3.3.2. Configuring a DataSource for Remote Usage

JBoss EAP supports accessing a DataSource from a remote client. See [Example 3.5, “Configuring a DataSource for Remote Usage”](#) for the change that gives the client the ability to look up the DataSource from JNDI, which is to specify **use-java-context=false**.

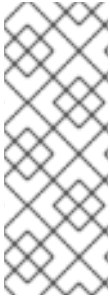
Example 3.5. Configuring a Datasource for Remote Usage

```

<datasources>
  <local-tx-datasource>
    <jndi-name>GenericDS</jndi-name>
    <use-java-context>false</use-java-context>
    <connection-url>...</connection-url>
    ...

```

This causes the DataSource to be bound under the JNDI name **GenericDS** instead of the default of **java:/GenericDS**, which restricts the lookup to the same Virtual Machine as the EAP server.



NOTE

Use of the `<use-java-context>` setting is not recommended in a production environment. It requires accessing a connection pool remotely and this can cause unexpected problems, since connections are not serializable. Also, transaction propagation is not supported, since it can lead to connection leaks if unreliability is present, such as in a system crash or network failure. A remote session bean facade is the preferred way to access a datasource remotely.

3.3.3. Configuring a Datasource to Use Login Modules

Procedure 3.1. Configuring a Datasource to Use Login Modules

1. Add the `<security-domain-parameter>` to the XML file for the datasource.

```
<datasources>
  <local-tx-datasource>
    ...
    <security-domain>MyDomain</security-domain>
    ...
  </local-tx-datasource>
</datasources>
```

2. Add an application policy to the `login-config.xml` file.

The authentication section needs to include the configuration for your login-module. For example, to encrypt the database password, use the **SecureIdentityLoginModule** login module.

```
<application-policy name="MyDomain">
  <authentication>
    <login-module
code="org.jboss.resource.security.SecureIdentityLoginModule"
flag="required">
      <module-option name="username">scott</module-option>
      <module-option name="password">-170dd0fbd8c13748</module-
option>
      <module-option
name="managedConnectionFactoryName">jboss.jca:service=LocalTxCM,name
=OracleDSJAAS</module-option>
    </login-module>
  </authentication>
</application-policy>
```

3. If you plan to fetch the data source connection from a web application, authentication must be enabled for the web application, so that the **Subject** is populated.
4. If users need the ability to connect anonymously, add an additional login module to the application-policy, to populate the security credentials.
5. Add the **UsersRolesLoginModule** module to the beginning of the chain. The **usersProperties** and **rolesProperties** parameters can be directed to dummy files.

```
<login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
```

```
flag="required">
  <module-option name="unauthenticatedIdentity">nobody</module-
option>
  <module-option
name="usersProperties">props/users.properties</module-option>
  <module-option
name="rolesProperties">props/roles.properties</module-option>
</login-module>
```


CHAPTER 4. DEPLOYMENT

Deploying applications on JBoss AS is very easy. You just need to copy the application into the `jboss-as/server/production/deploy` directory. You can replace default with different server profiles such as `all` or `minimal` or `production`. We will cover those later in this chapter. JBoss AS constantly scans the `deploy` directory to pick up new applications or any changes to existing applications. So, you can "hot deploy" application on the fly while JBoss AS is still running.

4.1. DEPLOYABLE APPLICATION TYPES

You can deploy several different types of enterprise applications in JBoss AS:

- The WAR application archive (e.g., `myapp.war`) packages a Java EE web application in a JAR file. It contains servlet classes, view pages, libraries, and deployment descriptors such as `web.xml`, `faces-config.xml`, and `jboss-web.xml` etc..
- The EAR application archive (e.g., `myapp.ear`) packages a Java EE enterprise application in a JAR file. It typically contains a WAR file for the web module, JAR files for EJB modules, as well as deployment descriptors such as `application.xml` and `jboss-app.xml` etc..
- The SAR application archive (e.g., `myservice.sar`) packages a JBoss service in a JAR file. It is mostly used by JBoss internal services. Please see more in [Chapter 5, The JBoss JMX Microkernel](#).
- The `*-ds.xml` file defines connections to external databases. The data source can then be reused by all applications and services in JBoss AS via the internal JNDI.
- You can deploy XML files with MBean service definitions. If you have the appropriate JAR files available in the `deploy` or `lib` directories, the MBeans specified in the XML files will be started. This is the way how you start many JBoss AS internal services, such as the JMS queues.
- You can also deploy JAR files containing EJBs or other service objects directly in JBoss AS.



NOTE

The WAR, EAR, and SAR deployment packages are really just JAR files with special XML deployment descriptors in directories like `META-INF` and `WEB-INF`. JBoss AS allows you to deploy those archives as expanded directories instead of JAR files. That allows you to make changes to web pages etc on the fly without re-deploying the entire application. If you do need to re-deploy the exploded directory without re-start the server, you can just "touch" the deployment descriptors (e.g., the `WEB-INF/web.xml` in a WAR and the `META-INF/application.xml` in an EAR) to update their timestamps.

4.2. STANDARD SERVER CONFIGURATIONS

The JBoss Enterprise Platform ships with four server configurations. You can choose which configuration to start by passing the `-c` parameter to the server startup script. For instance, command `run.sh -c all` would start the server in the `all` configuration. Each configuration is contained in a directory named `jboss-as/server/[config name]/`. You can look into each server configuration's directory to see the default services, applications, and libraries supported in the configuration.

- The minimal configuration starts the core server container without any of the enterprise services. It is a good starting point if you want to build a customized version of JBoss AS that only contains the servers you need.

- The default configuration is the mostly common used configuration for application developers. It supports the standard J2EE 1.4 and most of the Java EE 5.0 programming APIs (e.g., JSF and EJB3).
- The all configuration is the default configuration with clustering support and other enterprise extensions.
- The production configuration is based on the all configuration but with key parameters pre-tuned for production deployment.

The detailed services and APIs supported in each of those configurations will be discussed throughout this book. In this section, we focus on the optimization we did for the production configuration.

4.2.1. The production Configuration

To start the server in the production configuration, you can use the following command under Linux / Unix:

```
cd /path/to/jboss-as
RUN_CONF=server/production/run.conf bin/run.sh -c production
```

Or, you can simply copy the `jboss-as/server/production/run.conf` file to `jboss-as/bin` directory and start the server with `run.sh -c production` command. Below is a list of optimizations we specifically did for the production configuration:

- In the `jboss-as/server/production/run.conf` file, we expanded the memory size of the server to 1.7 GB. We added the `-server` tag to JVM startup command on all platforms except for Darwin (Mac OS X). If the JVM is BEA jRockit, the `-Xgc:gencon` parameter is also added.
- We configured the key generation algorithm to use the database to generate HiLo keys in order to generate the correct keys in a cluster environment (see `deploy/uuid-key-generator.sar/META-INF/jboss-service.xml`).
- We set the `ScanPeriod` parameter to 60000 in `conf/jboss-minimal.xml` and `conf/jboss-service.xml`, so that JBoss AS does not spend too much time constantly scanning the deploy directory for new or updated deployments.
- We removed the connection monitoring in `deploy/jbossjca-service.xml`. The connection monitoring feature helps catch unclosed connections that would otherwise cause leaks in the connection pools in development. However, it is a global point of contention that should be turned off (`false`) in production.
- Logging is a big contention point in many production applications. In the production configuration, we removed the console logging and increased the logging level to `WARN` and `ERROR` for most packages. Please see details in `conf/jboss-log4j.xml`.

4.2.2. Further Tuning from the production Configuration

In addition to the standard optimization in the production configuration, there are a couple of simple techniques you can use to improve the performance and stability of your server.

The production configuration increases the JVM heap memory size to 1.7 GB. You should probably change it to fit your own server. For instance, if have a 64 bit server with several GBs of RAM, you can probably increase this value as long as you also use a 64 bit JVM. If your server has less than 2 GB RAM, you should decrease that value accordingly. In the `production/run.conf` file, the `-Xmx` and `-Xms` parameters specify the maximum and minimum heap sizes respectively. It is recommended that you set

the `-Xmx` and `-Xms` to the same value to avoid dynamic re-sizing of the heap, which is a source of instability in many JVMs. You could also consider turning on parallel GC options if you are using the Sun JVM on a multi-core machine. The following is an example setup you might use as a reference. Please see the Sun JVM documentation for more details on this startup parameters.

```
JAVA_OPTS="-Xms1740m -Xmx1740m -XX:PermSize=256m -XX:MaxPermSize=512
-XX:+UseConcMarkSweepGC -XX:+CMSPermGenSweepingEnabled
-XX:+CMSClassUnloadingEnabled"
```

In the embedded Tomcat module, you can turn off the development mode so that the server does not constantly monitor the changes in JSP files. To do that, edit the `deploy/jboss-web.deployer/conf/web.xml` file and add the development attribute to the `JspServlet`.

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  ... ..
  <init-param>
    <param-name>development</param-name>
    <param-value>>false</param-value>
  </init-param>
  ... ..
```

In Tomcat, you could adjust the size of the thread pool. If you have multi-core CPUs or more than one CPUs on your server, it might be beneficial to increase the thread pool beyond the default 250. On the other hand, if you have a slow server, decreasing the thread pool will decrease the overhead on the server. The thread pool size can be adjusted via the `deploy/jboss-web.deployer/server.xml` file.

```
... ..
<Connector port="8080" address="${jboss.bind.address}"
  maxThreads="250" maxHttpHeaderSize="8192"
  emptySessionPath="true" protocol="HTTP/1.1"
  enableLookups="false" redirectPort="8443" acceptCount="100"
  connectionTimeout="20000" disableUploadTimeout="true" />
... ..
```

In addition, JBoss AS needs to use a relational database to store runtime data. In a production environment, you should use a production quality database to replace the embedded HSQL database. Please see [Appendix B, Use Alternative Databases with JBoss AS](#) for more information on how to setup alternative databases for the JBoss AS.

PART II. JBOSS AS INFRASTRUCTURE

CHAPTER 5. THE JBOSS JMX MICROKERNEL

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

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

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

5.1. AN INTRODUCTION TO 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 5.1, “The JBoss JMX integration bus and the standard JBoss components” 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.

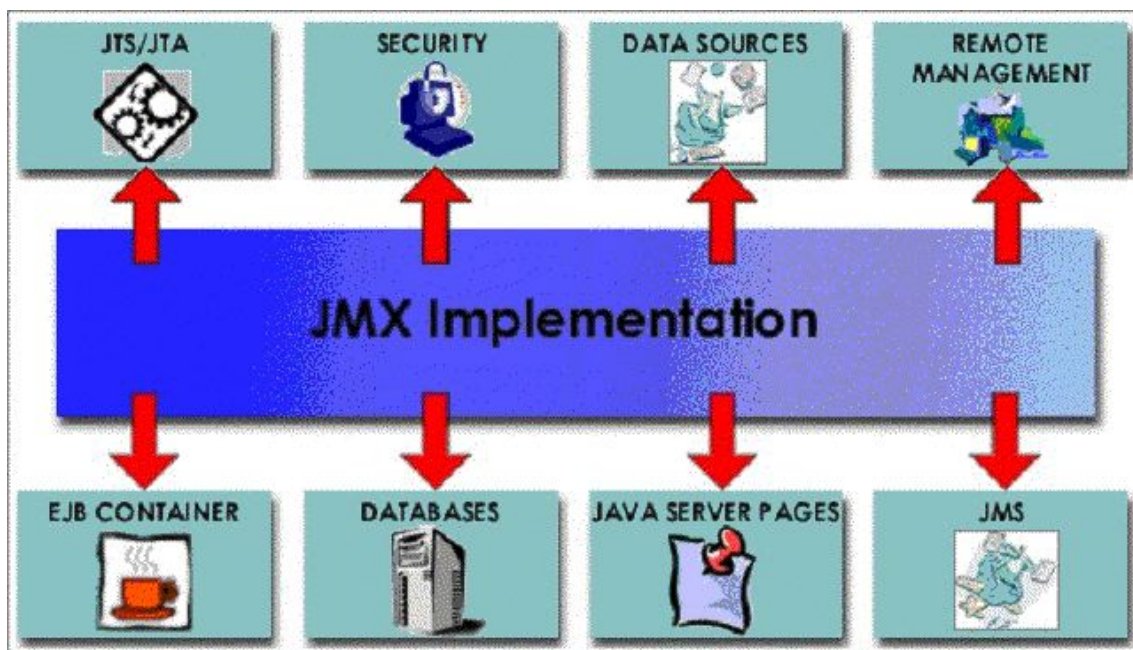


Figure 5.1. The JBoss JMX integration bus and the standard JBoss components

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, v1.2, which is available from the JSR003 Web page at <http://jcp.org/en/jsr/detail?id=3>. The

material in this JMX overview section is derived from the 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, 2002).

JMX is 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 5.2, “The Relationship between the components of the JMX architecture”](#) 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

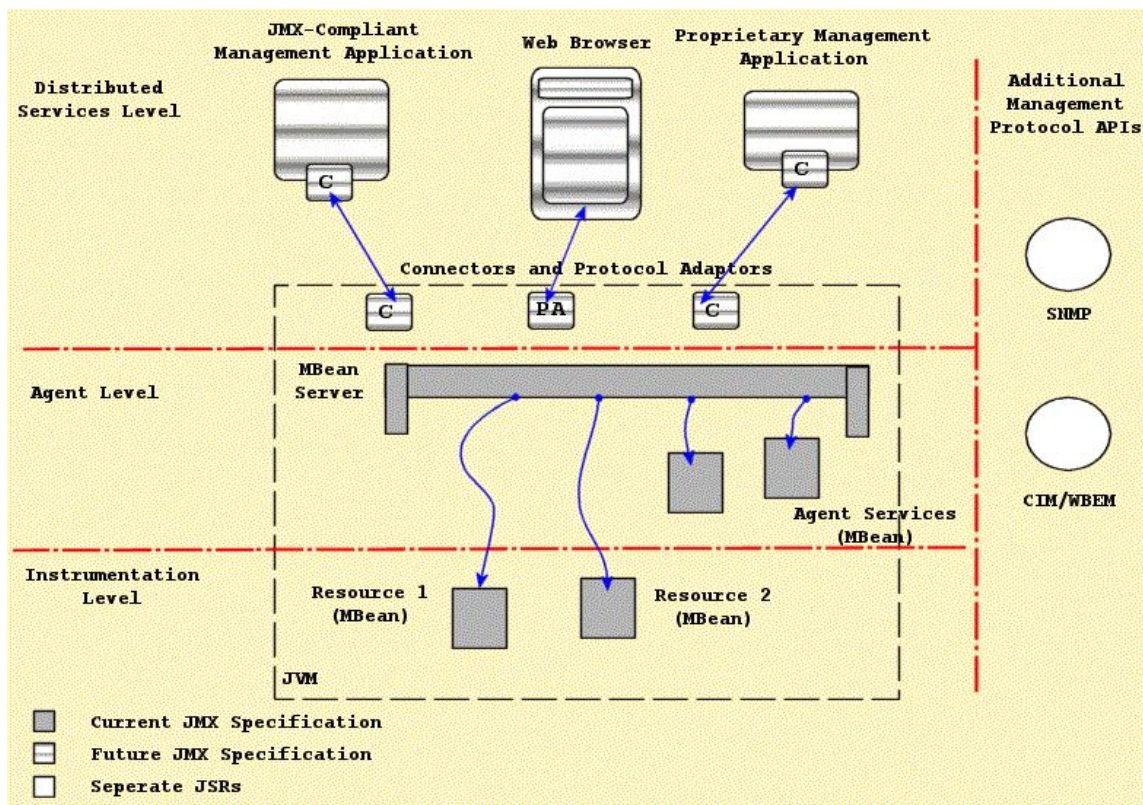


Figure 5.2. The Relationship between the components of the JMX architecture

5.1.1. 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*.

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.

5.1.2. 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 both an html adaptor as well as 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 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 [Section 5.1.3, “Distributed Services Level”](#) in the next section. The agent does not need to know anything about the connectors or management applications that interact with the agent and its MBeans.

5.1.3. 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 by the component boxes with the horizontal lines in [Figure 5.2, “The Relationship between the components of the JMX architecture”](#). 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.

5.1.4. 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

5.1.4.1. 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:

- Attribute 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 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, self-describing, user-friendly data types for universal manageability.
- **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 XMBeans are an implementation of Model MBeans.

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

5.1.4.2. Notification Model

JMX Notifications are an extension of the Java event model. Both the MBean server 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 operations on the MBean server that allow for the registration of notification listeners.

5.1.4.3. 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 MBean server with which the MBeans are registered. The metadata classes cover an MBean's attributes, operations, notifications, and constructors. For each of these, the metadata includes a name, a description, and its particular characteristics. For example, one characteristic of an attribute is whether it is readable, writable, or both. The metadata for an operation contains the signature of its parameter and return types.

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

5.1.4.4. 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 MBean server is a registry for MBeans that makes the MBean management interface available for use by management applications. The MBean never directly exposes the MBean object itself; rather, its management interface is exposed through metadata and operations available in the MBean server 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 MBean server 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 MBean server the adaptor connects to. An example adaptor is an HTML adaptor that allows for the inspection and editing of MBeans using a Web browser. As was indicated in [Figure 5.2, “The Relationship between the components of the JMX architecture”](#), there are no protocol adaptors defined by the current JMX specification. Later versions of the specification will address the need for remote access protocols in standard ways.

A connector is an interface used by management applications to provide a common API for accessing the MBean server 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 a custom HTML adaptor implementation and a custom JBoss RMI adaptor.

5.1.4.5. 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. However, JBoss does not rely on any of these standard agent services.

5.2. JBOSS JMX IMPLEMENTATION ARCHITECTURE

5.2.1. The JBoss ClassLoader Architecture

JBoss employs a class loading architecture that facilitates sharing of classes across deployment units and hot deployment of services and applications. Before discussing the JBoss specific class loading model, we need to understand the nature of Java's type system and how class loaders fit in.

5.2.2. Class Loading and Types in Java

Class loading is a fundamental part of all server architectures. Arbitrary services and their supporting classes must be loaded into the server framework. This can be problematic due to the strongly typed nature of Java. Most developers know that the type of a class in Java is a function of the fully qualified name of the class. However the type is also a function of the `java.lang.ClassLoader` that is used to define that class. This additional qualification of type is necessary to ensure that environments in which classes may be loaded from arbitrary locations would be type-safe.

However, in a dynamic environment like an application server, and especially JBoss with its support for hot deployment are that class cast exceptions, linkage errors and illegal access errors can show up in ways not seen in more static class loading contexts. Let's take a look at the meaning of each of these exceptions and how they can happen.

5.2.2.1. ClassCastException - I'm Not Your Type

A `java.lang.ClassCastException` results whenever an attempt is made to cast an instance to an incompatible type. A simple example is trying to obtain a **String** from a **List** into which a **URL** was placed:

```
ArrayList array = new ArrayList();
array.add(new URL("file:/tmp"));
String url = (String) array.get(0);

java.lang.ClassCastException: java.net.URL
at org.jboss.book.jmx.ex0.ExCCEa.main(Ex1CCE.java:16)
```

The **ClassCastException** tells you that the attempt to cast the array element to a **String** failed because the actual type was **URL**. This trivial case is not what we are interested in however. Consider the case of a JAR being loaded by different class loaders. Although the classes loaded through each class loader are identical in terms of the bytecode, they are completely different types as viewed by the Java type system. An example of this is illustrated by the code shown in [Example 5.1](#), “The `ExCCEc` class used to demonstrate `ClassCastException` due to duplicate class loaders”.

Example 5.1. The `ExCCEc` class used to demonstrate `ClassCastException` due to duplicate class loaders

```
package org.jboss.book.jmx.ex0;

import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.lang.reflect.Method;

import org.apache.log4j.Logger;

import org.jboss.util.ChapterExRepository;
import org.jboss.util.Debug;

/**
 * An example of a ClassCastException that
 * results from classes loaded through
 * different class loaders.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ExCCEc
{
    public static void main(String[] args) throws Exception
    {
        ChapterExRepository.init(ExCCEc.class);

        String chapDir = System.getProperty("j2eechapter.dir");
        Logger ucl0Log = Logger.getLogger("UCL0");
        File jar0 = new File(chapDir+"/j0.jar");
        ucl0Log.info("jar0 path: "+jar0.toString());
        URL[] cp0 = {jar0.toURL()};
        URLClassLoader ucl0 = new URLClassLoader(cp0);
        Thread.currentThread().setContextClassLoader(ucl0);
```

```

        Class objClass =
ucl0.loadClass("org.jboss.book.jmx.ex0.ExObj");
        StringBuffer buffer = new
            StringBuffer("ExObj Info");
        Debug.displayClassInfo(objClass, buffer, false);
        ucl0Log.info(buffer.toString());
        Object value = objClass.newInstance();

        File jar1 = new File(chapDir+"/j0.jar");
        Logger ucl1Log = Logger.getLogger("UCL1");
        ucl1Log.info("jar1 path: "+jar1.toString());
        URL[] cp1 = {jar1.toURL()};
        URLClassLoader ucl1 = new URLClassLoader(cp1);
        Thread.currentThread().setContextClassLoader(ucl1);
        Class ctxClass2 =
ucl1.loadClass("org.jboss.book.jmx.ex0.ExCtx");
        buffer.setLength(0);
        buffer.append("ExCtx Info");
        Debug.displayClassInfo(ctxClass2, buffer, false);
        ucl1Log.info(buffer.toString());
        Object ctx2 = ctxClass2.newInstance();

        try {
            Class[] types = {Object.class};
            Method useValue =
                ctxClass2.getMethod("useValue", types);
            Object[] margs = {value};
            useValue.invoke(ctx2, margs);
        } catch (Exception e) {
            ucl1Log.error("Failed to invoke ExCtx.useValue", e);
            throw e;
        }
    }
}

```

Example 5.2. The ExCtx, ExObj, and ExObj2 classes used by the examples

```

package org.jboss.book.jmx.ex0;

import java.io.IOException;
import org.apache.log4j.Logger;
import org.jboss.util.Debug;

/**
 * A classes used to demonstrate various class
 * loading issues
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ExCtx
{
    ExObj value;

    public ExCtx()

```

```

        throws IOException
    {
        value = new ExObj();
        Logger log = Logger.getLogger(ExCtx.class);
        StringBuffer buffer = new StringBuffer("ctor.ExObj");
        Debug.displayClassInfo(value.getClass(), buffer, false);
        log.info(buffer.toString());
        ExObj2 obj2 = value.ivar;
        buffer.setLength(0);
        buffer = new StringBuffer("ctor.ExObj.ivar");
        Debug.displayClassInfo(obj2.getClass(), buffer, false);
        log.info(buffer.toString());
    }

    public Object getValue()
    {
        return value;
    }

    public void useValue(Object obj)
        throws Exception
    {
        Logger log = Logger.getLogger(ExCtx.class);
        StringBuffer buffer = new
            StringBuffer("useValue2.arg class");
        Debug.displayClassInfo(obj.getClass(), buffer, false);
        log.info(buffer.toString());
        buffer.setLength(0);
        buffer.append("useValue2.ExObj class");
        Debug.displayClassInfo(ExObj.class, buffer, false);
        log.info(buffer.toString());
        ExObj ex = (ExObj) obj;
    }

    void pkgUseValue(Object obj)
        throws Exception
    {
        Logger log = Logger.getLogger(ExCtx.class);
        log.info("In pkgUseValue");
    }
}

package org.jboss.book.jmx.ex0;

import java.io.Serializable;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ExObj
    implements Serializable
{
    public ExObj2 ivar = new ExObj2();
}

```

```

package org.jboss.book.jmx.ex0;

import java.io.Serializable;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ExObj2
    implements Serializable
{
}

```

The **ExCCEc.main** method uses reflection to isolate the classes that are being loaded by the class loaders **uc10** and **uc11** from the application class loader. Both are setup to load classes from the **output/jmx/j0.jar**, the contents of which are:

```

[examples]$ jar -tf output/jmx/j0.jar
...
org/jboss/book/jmx/ex0/ExCtx.class
org/jboss/book/jmx/ex0/ExObj.class
org/jboss/book/jmx/ex0/ExObj2.class

```

We will run an example that demonstrates how a class cast exception can occur and then look at the specific issue with the example. See [Appendix A, Book Example Installation](#) for instructions on installing the examples accompanying the book, and then run the example from within the examples directory using the following command:

```

[examples]$ ant -Dchap=jmx -Dex=0c run-example
...
[java] java.lang.reflect.InvocationTargetException
[java]     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
[java]     at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:
39)
[java]     at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorIm
pl
        .java:25)
[java]     at java.lang.reflect.Method.invoke(Method.java:585)
[java]     at org.jboss.book.jmx.ex0.ExCCEc.main(ExCCEc.java:58)
[java] Caused by: java.lang.ClassCastException:
org.jboss.book.jmx.ex0.ExObj
[java]     at org.jboss.book.jmx.ex0.ExCtx.useValue(ExCtx.java:44)
[java]     ... 5 more

```

Only the exception is shown here. The full output can be found in the **logs/jmx-ex0c.log** file. At line 55 of **ExCCEc.java** we are invoking **ExCCEctx.useValue(Object)** on the instance loaded and created in lines 37-48 using **uc11**. The **ExObj** passed in is the one loaded and created in lines 25-35 via **uc10**. The exception results when the **ExCtx.useValue** code attempts to cast the argument passed in to a **ExObj**. To understand why this fails consider the debugging output from the **jmx-ex0c.log** file shown in [Example 5.3, "The jmx-ex0c.log debugging output for the ExObj classes seen"](#).

Example 5.3. The jmx-ex0c.log debugging output for the ExObj classes seen

```

[INFO,UCL0] ExObj Info
org.jboss.book.jmx.ex0.ExObj(f8968f).ClassLoader=java.net.URLClassLoader
@2611a7
..java.net.URLClassLoader@2611a7
....file:/Users/orb/proj/jboss/jboss-
docs/jbossas/j2ee/examples/output/jmx/j0.jar
++++CodeSource: (file:/Users/orb/proj/jboss/jboss-
docs/jbossas/j2ee/examples/output/
                    jmx/j0.jar <no signer certificates>)
Implemented Interfaces:
++interface java.io.Serializable(41b571)
++++ClassLoader: null
++++Null CodeSource
[INFO,ExCtx] useValue2.ExObj class
org.jboss.book.jmx.ex0.ExObj(bc8e1e).ClassLoader=java.net.URLClassLoader
@6bd8ea
..java.net.URLClassLoader@6bd8ea
....file:/Users/orb/proj/jboss/jboss-
docs/jbossas/j2ee/examples/output/jmx/j0.jar
++++CodeSource: (file:/Users/orb/proj/jboss/jboss-
docs/jbossas/j2ee/examples/output/
                    jmx/j0.jar <no signer certificates>)
Implemented Interfaces:
++interface java.io.Serializable(41b571)
++++ClassLoader: null
++++Null CodeSource

```

The first output prefixed with **[INFO, UCL0]** shows that the **ExObj** class loaded at line **ExCCEc.java:31** has a hash code of **f8968f** and an associated **URLClassLoader** instance with a hash code of **2611a7**, which corresponds to **ucl0**. This is the class used to create the instance passed to the **ExCtx.useValue** method. The second output prefixed with **[INFO, ExCtx]** shows that the **ExObj** class as seen in the context of the **ExCtx.useValue** method has a hash code of **bc8e1e** and a **URLClassLoader** instance with an associated hash code of **6bd8ea**, which corresponds to **ucl1**. So even though the **ExObj** classes are the same in terms of actual bytecode since it comes from the same **j0.jar**, the classes are different as seen by both the **ExObj** class hash codes, and the associated **URLClassLoader** instances. Hence, attempting to cast an instance of **ExObj** from one scope to the other results in the **ClassCastException**.

This type of error is common when redeploying an application to which other applications are holding references to classes from the redeployed application. For example, a standalone WAR accessing an EJB. If you are redeploying an application, all dependent applications must flush their class references. Typically this requires that the dependent applications themselves be redeployed.

An alternate means of allowing independent deployments to interact in the presence of redeployment would be to isolate the deployments by configuring the EJB layer to use the standard call-by-value semantics rather than the call-by-reference JBoss will default to for components collocated in the same VM. An example of how to enable call-by-value semantics is presented in [Chapter 20, EJBs on JBoss](#)

5.2.2.2. IllegalAccessException - Doing what you should not

A **java.lang.IllegalAccessException** is thrown when one attempts to access a method or

member that visibility qualifiers do not allow. Typical examples are attempting to access private or protected methods or instance variables. Another common example is accessing package protected methods or members from a class that appears to be in the correct package, but is really not due to caller and callee classes being loaded by different class loaders. An example of this is illustrated by the code shown in [Example 5.4, “The ExIAEd class used to demonstrate `IllegalAccessException` due to duplicate class loaders”](#).

Example 5.4. The ExIAEd class used to demonstrate `IllegalAccessException` due to duplicate class loaders

```
package org.jboss.book.jmx.ex0;

import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.lang.reflect.Method;

import org.apache.log4j.Logger;

import org.jboss.util.ChapterExRepository;
import org.jboss.util.Debug;

/**
 * An example of IllegalAccessExceptions due to
 * classes loaded by two class loaders.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ExIAEd
{
    public static void main(String[] args) throws Exception
    {
        ChapterExRepository.init(ExIAEd.class);

        String chapDir = System.getProperty("j2eechapter.dir");
        Logger ucl0Log = Logger.getLogger("UCL0");
        File jar0 = new File(chapDir+"/j0.jar");
        ucl0Log.info("jar0 path: "+jar0.toString());
        URL[] cp0 = {jar0.toURL()};
        URLClassLoader ucl0 = new URLClassLoader(cp0);
        Thread.currentThread().setContextClassLoader(ucl0);

        StringBuffer buffer = new
            StringBuffer("ExIAEd Info");
        Debug.displayClassInfo(ExIAEd.class, buffer, false);
        ucl0Log.info(buffer.toString());

        Class ctxClass1 = ucl0.loadClass("org.jboss.book.jmx.ex0.ExCtx");
        buffer.setLength(0);
        buffer.append("ExCtx Info");
        Debug.displayClassInfo(ctxClass1, buffer, false);
        ucl0Log.info(buffer.toString());
        Object ctx0 = ctxClass1.newInstance();

        try {
            Class[] types = {Object.class};
```



```

        Method useValue =
ctxClass1.getDeclaredMethod("pkgUseValue", types);
        Object[] margs = {null};
        useValue.invoke(ctx0, margs);
    } catch (Exception e) {
        ucl0Log.error("Failed to invoke ExCtx.pkgUseValue", e);
    }
}
}

```

The **ExIAEd.main** method uses reflection to load the **ExCtx** class via the **ucl0** class loader while the **ExIEAd** class was loaded by the application class loader. We will run this example to demonstrate how the **IllegalAccessException** can occur and then look at the specific issue with the example. Run the example using the following command:

```

[examples]$ ant -Dchap=jmx -Dex=0d run-example
Buildfile: build.xml
...
[java] java.lang.IllegalAccessException: Class
org.jboss.book.jmx.ex0.ExIAEd
    can not access a member of class org.jboss.book.jmx.ex0.ExCtx with
modifiers ""
[java]     at
sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
[java]     at java.lang.reflect.Method.invoke(Method.java:578)
[java]     at org.jboss.book.jmx.ex0.ExIAEd.main(ExIAEd.java:48)

```

The truncated output shown here illustrates the **IllegalAccessException**. The full output can be found in the **logs/jmx-ex0d.log** file. At line 48 of **ExIAEd.java** the **ExCtx.pkgUseValue(Object)** method is invoked via reflection. The **pkgUseValue** method has package protected access and even though both the invoking class **ExIAEd** and the **ExCtx** class whose method is being invoked reside in the **org.jboss.book.jmx.ex0** package, the invocation is seen to be invalid due to the fact that the two classes are loaded by different class loaders. This can be seen by looking at the debugging output from the **jmx-ex0d.log** file.

```

[INFO,UCL0] ExIAEd Info
org.jboss.book.jmx.ex0.ExIAEd(7808b9).ClassLoader=sun.misc.Launcher$AppCla
ssLoader@a9c85c
..sun.misc.Launcher$AppClassLoader@a9c85c
...
[INFO,UCL0] ExCtx Info
org.jboss.book.jmx.ex0.ExCtx(64c34e).ClassLoader=java.net.URLClassLoader@a
9c85c
..java.net.URLClassLoader@5d88a
...

```

The **ExIAEd** class is seen to have been loaded via the default application class loader instance **sun.misc.Launcher\$AppClassLoader@a9c85c**, while the **ExCtx** class was loaded by the **java.net.URLClassLoader@a9c85c** instance. Because the classes are loaded by different class loaders, access to the package protected method is seen to be a security violation. So, not only is type a function of both the fully qualified class name and class loader, the package scope is as well.

An example of how this can happen in practice is to include the same classes in two different SAR

deployments. If classes in the deployment have a package protected relationship, users of the SAR service may end up loading one class from SAR class loading at one point, and then load another class from the second SAR at a later time. If the two classes in question have a protected access relationship an **IllegalAccessError** will result. The solution is to either include the classes in a separate jar that is referenced by the SARs, or to combine the SARs into a single deployment. This can either be a single SAR, or an EAR that includes both SARs.

5.2.2.3. LinkageErrors - Making Sure You Are Who You Say You Are

Loading constraints validate type expectations in the context of class loader scopes to ensure that a class **X** is consistently the same class when multiple class loaders are involved. This is important because Java allows for user defined class loaders. Linkage errors are essentially an extension of the class cast exception that is enforced by the VM when classes are loaded and used.

To understand what loading constraints are and how they ensure type-safety we will first introduce the nomenclature of the Liang and Bracha paper along with an example from this paper. There are two type of class loaders, initiating and defining. An initiating class loader is one that a **ClassLoader.loadClass** method has been invoked on to initiate the loading of the named class. A defining class loader is the loader that calls one of the **ClassLoader.defineClass** methods to convert the class byte code into a **Class** instance. The most complete expression of a class is given by **<C, Ld>^{Li}**, where **C** is the fully qualified class name, **Ld** is the defining class loader, and **Li** is the initiating class loader. In a context where the initiating class loader is not important the type may be represented by **<C, Ld>**, while when the defining class loader is not important, the type may be represented by **C^{Li}**. In the latter case, there is still a defining class loader, it's just not important what the identity of the defining class loader is. Also, a type is completely defined by **<C, Ld>**. The only time the initiating loader is relevant is when a loading constraint is being validated. Now consider the classes shown in [Example 5.5, "Classes demonstrating the need for loading constraints"](#).

Example 5.5. Classes demonstrating the need for loading constraints

```
class <C,L1> {
    void f() {
        <Spoofed, L1>L1x = <Delegated, L2>L2
        x.secret_value = 1; // Should not be allowed
    }
}

class <Delegated,L2> {
    static <Spoofed, L2>L3 g() {...}
}

class <Spoofed, L1> {
    public int secret_value;
}

class <Spoofed, L2> {
    private int secret_value;
}
```

The class **C** is defined by **L1** and so **L1** is used to initiate loading of the classes **Spoofed** and

Delegated referenced in the **C.f()** method. The **Spoofed** class is defined by **L1**, but **Delegated** is defined by **L2** because **L1** delegates to **L2**. Since **Delegated** is defined by **L2**, **L2** will be used to initiate loading of **Spoofed** in the context of the **Delegated.g()** method. In this example both **L1** and **L2** define different versions of **Spoofed** as indicated by the two versions shown at the end of [Example 5.5, “Classes demonstrating the need for loading constraints”](#). Since **C.f()** believes **x** is an instance of **<Spoofed, L1>** it is able to access the private field **secret_value** of **<Spoofed, L2>** returned by **Delegated.g()** due to the 1.1 and earlier Java VM's failure to take into account that a class type is determined by both the fully qualified name of the class and the defining class loader.

Java addresses this problem by generating loader constraints to validate type consistency when the types being used are coming from different defining class loaders. For the [Example 5.5, “Classes demonstrating the need for loading constraints”](#) example, the VM generates a constraint **Spoofed^{L1}=Spoofed^{L2}** when the first line of method **C.f()** is verified to indicate that the type **Spoofed** must be the same regardless of whether the load of **Spoofed** is initiated by **L1** or **L2**. It does not matter if **L1** or **L2**, or even some other class loader defines **Spoofed**. All that matters is that there is only one **Spoofed** class defined regardless of whether **L1** or **L2** was used to initiate the loading. If **L1** or **L2** have already defined separate versions of **Spoofed** when this check is made a **LinkageError** will be generated immediately. Otherwise, the constraint will be recorded and when **Delegated.g()** is executed, any attempt to load a duplicate version of **Spoofed** will result in a **LinkageError**.

Now let's take a look at how a **LinkageError** can occur with a concrete example. [Example 5.6, “A concrete example of a LinkageError”](#) gives the example main class along with the custom class loader used.

Example 5.6. A concrete example of a LinkageError

```
package org.jboss.book.jmx.ex0;
import java.io.File;
import java.net.URL;

import org.apache.log4j.Logger;
import org.jboss.util.ChapterExRepository;
import org.jboss.util.Debug;

/**
 * An example of a LinkageError due to classes being defined by more
 * than one class loader in a non-standard class loading environment.
 *
 * @author Scott.Stark@jboss.orgn
 * @version $Revision: 1.1 $
 */
public class ExLE
{
    public static void main(String[] args)
    throws Exception
    {
        ChapterExRepository.init(ExLE.class);

        String chapDir = System.getProperty("j2eechapter.dir");
        Logger ucl0Log = Logger.getLogger("UCL0");
        File jar0 = new File(chapDir+"/j0.jar");
        ucl0Log.info("jar0 path: "+jar0.toString());
        URL[] cp0 = {jar0.toURL()};
        Ex0URLClassLoader ucl0 = new Ex0URLClassLoader(cp0);
        Thread.currentThread().setContextClassLoader(ucl0);
    }
}
```

```

        Class ctxClass1 =
ucl0.loadClass("org.jboss.book.jmx.ex0.ExCtx");
        Class obj2Class1 =
ucl0.loadClass("org.jboss.book.jmx.ex0.ExObj2");
        StringBuffer buffer = new StringBuffer("ExCtx Info");
        Debug.displayClassInfo(ctxClass1, buffer, false);
        ucl0Log.info(buffer.toString());
        buffer.setLength(0);
        buffer.append("ExObj2 Info, UCL0");
        Debug.displayClassInfo(obj2Class1, buffer, false);
        ucl0Log.info(buffer.toString());

        File jar1 = new File(chapDir+"/j1.jar");
        Logger ucl1Log = Logger.getLogger("UCL1");
        ucl1Log.info("jar1 path: "+jar1.toString());
        URL[] cp1 = {jar1.toURL()};
        Ex0URLClassLoader ucl1 = new Ex0URLClassLoader(cp1);
        Class obj2Class2 =
ucl1.loadClass("org.jboss.book.jmx.ex0.ExObj2");
        buffer.setLength(0);
        buffer.append("ExObj2 Info, UCL1");
        Debug.displayClassInfo(obj2Class2, buffer, false);
        ucl1Log.info(buffer.toString());

        ucl0.setDelegate(ucl1);
        try {
            ucl0Log.info("Try ExCtx.newInstance()");
            Object ctx0 = ctxClass1.newInstance();
            ucl0Log.info("ExCtx.ctor succeeded, ctx0: "+ctx0);
        } catch(Throwable e) {
            ucl0Log.error("ExCtx.ctor failed", e);
        }
    }
}

package org.jboss.book.jmx.ex0;

import java.net.URLClassLoader;
import java.net.URL;

import org.apache.log4j.Logger;

/**
 * A custom class loader that overrides the standard parent delegation
 * model
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class Ex0URLClassLoader extends URLClassLoader
{
    private static Logger log =
Logger.getLogger(Ex0URLClassLoader.class);
    private Ex0URLClassLoader delegate;

    public Ex0URLClassLoader(URL[] urls)

```

```

    {
        super(urls);
    }

    void setDelegate(Ex0URLClassLoader delegate)
    {
        this.delegate = delegate;
    }

    protected synchronized Class loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        Class clazz = null;
        if (delegate != null) {
            log.debug(Integer.toHexString(hashCode()) +
                "; Asking delegate to loadClass: " + name);
            clazz = delegate.loadClass(name, resolve);
            log.debug(Integer.toHexString(hashCode()) +
                "; Delegate returned: "+clazz);
        } else {
            log.debug(Integer.toHexString(hashCode()) +
                "; Asking super to loadClass: "+name);
            clazz = super.loadClass(name, resolve);
            log.debug(Integer.toHexString(hashCode()) +
                "; Super returned: "+clazz);
        }
        return clazz;
    }

    protected Class findClass(String name)
        throws ClassNotFoundException
    {
        Class clazz = null;
        log.debug(Integer.toHexString(hashCode()) +
            "; Asking super to findClass: "+name);
        clazz = super.findClass(name);
        log.debug(Integer.toHexString(hashCode()) +
            "; Super returned: "+clazz);
        return clazz;
    }
}

```

The key component in this example is the **URLClassLoader** subclass **Ex0URLClassLoader**. This class loader implementation overrides the default parent delegation model to allow the **uc10** and **uc11** instances to both load the **ExObj2** class and then setup a delegation relationship from **uc10** to **uc11**. At lines 30 and 31, the **uc10Ex0URLClassLoader** is used to load the **ExCtx** and **ExObj2** classes. At line 45 of **ExLE.main** the **uc11Ex0URLClassLoader** is used to load the **ExObj2** class again. At this point both the **uc10** and **uc11** class loaders have defined the **ExObj2** class. A delegation relationship from **uc10** to **uc11** is then setup at line 51 via the **uc10.setDelegate(uc11)** method call. Finally, at line 54 of **ExLE.main** an instance of **ExCtx** is created using the class loaded via **uc10**. The **ExCtx** class is the same as presented in [Example 5.4, “The ExIAEd class used to demonstrate IllegalAccessException due to duplicate class loaders”](#), and the constructor was:

```
public ExCtx()
```

```

        throws IOException
    {
        value = new ExObj();
        Logger log = Logger.getLogger(ExCtx.class);
        StringBuffer buffer = new StringBuffer("ctor.ExObj");
        Debug.displayClassInfo(value.getClass(), buffer, false);
        log.info(buffer.toString());
        ExObj2 obj2 = value.ivar;
        buffer.setLength(0);
        buffer = new StringBuffer("ctor.ExObj.ivar");
        Debug.displayClassInfo(obj2.getClass(), buffer, false);
        log.info(buffer.toString());
    }

```

Now, since the **ExCtx** class was defined by the **uc10** class loader, and at the time the **ExCtx** constructor is executed, **uc10** delegates to **uc11**, line 24 of the **ExCtx** constructor involves the following expression which has been rewritten in terms of the complete type expressions:

```
<ExObj2,uc10>uc10 obj2 = <ExObj,uc11>uc10 value * ivar
```

This generates a loading constraint of **ExObj2^{uc10} = ExObj2^{uc11}** since the **ExObj2** type must be consistent across the **uc10** and **uc11** class loader instances. Because we have loaded **ExObj2** using both **uc10** and **uc11** prior to setting up the delegation relationship, the constraint will be violated and should generate a **LinkageError** when run. Run the example using the following command:

```

[examples]$ ant -Dchap=jmx -Dex=0e run-example
Buildfile: build.xml
...
[java] java.lang.LinkageError: loader constraints violated when linking
      org/jboss/book/jmx/ex0/ExObj2 class
[java]   at org.jboss.book.jmx.ex0.ExCtx.<init>(ExCtx.java:24)
[java]   at
sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
[java]   at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAcc
essor
      Impl.java:39)
[java]   at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstr
uctor
      AccessorImpl.java:27)
[java]   at
java.lang.reflect.Constructor.newInstance(Constructor.java:494)
[java]   at java.lang.Class.newInstance0(Class.java:350)
[java]   at java.lang.Class.newInstance(Class.java:303)
[java]   at org.jboss.book.jmx.ex0.ExLE.main(ExLE.java:53)

```

As expected, a **LinkageError** is thrown while validating the loader constraints required by line 24 of the **ExCtx** constructor.

5.2.2.3.1. Debugging Class Loading Issues

Debugging class loading issues comes down to finding out where a class was loaded from. A useful tool for this is the code snippet shown in [Example 5.7, “Obtaining debugging information for a Class”](#) taken from the `org.jboss.util.Debug` class of the book examples.

Example 5.7. Obtaining debugging information for a Class

```

Class clazz =...;
StringBuffer results = new StringBuffer();

ClassLoader cl = clazz.getClassLoader();
results.append("\n" + clazz.getName() + "(" +
              Integer.toHexString(clazz.hashCode()) + ").ClassLoader="
+ cl);
ClassLoader parent = cl;

while (parent != null) {
    results.append("\n.." + parent);
    URL[] urls = getClassLoaderURLs(parent);

    int length = urls != null ? urls.length : 0;
    for(int u = 0; u < length; u++) {
        results.append("\n..." + urls[u]);
    }

    if (showParentClassLoaders == false) {
        break;
    }
    if (parent != null) {
        parent = parent.getParent();
    }
}

CodeSource clazzCS = clazz.getProtectionDomain().getCodeSource();
if (clazzCS != null) {
    results.append("\n++++CodeSource: " + clazzCS);
} else {
    results.append("\n++++Null CodeSource");
}

```

Firstly, every **Class** object knows its defining **ClassLoader** and this is available via the **getClassLoader()** method. This defines the scope in which the **Class** type is known as we have just seen in the previous sections on class cast exceptions, illegal access exceptions and linkage errors. From the **ClassLoader** you can view the hierarchy of class loaders that make up the parent delegation chain. If the class loader is a **URLClassLoader** you can also see the URLs used for class and resource loading.

The defining **ClassLoader** of a **Class** cannot tell you from what location that **Class** was loaded. To determine this you must obtain the **java.security.ProtectionDomain** and then the **java.security.CodeSource**. It is the **CodeSource** that has the URL p location from which the class originated. Note that not every **Class** has a **CoPdeSource**. If a class is loaded by the bootstrap class loader then its **CodeSource** will be null. This will be the case for all classes in the **java.*** and **javax.*** packages, for example.

Beyond that it may be useful to view the details of classes being loaded into the JBoss server. You can enable verbose logging of the JBoss class loading layer using a Log4j configuration fragment like that shown in [Example 5.8, “An example log4j.xml configuration fragment for enabling verbose class loading logging”](#).

Example 5.8. An example log4j.xml configuration fragment for enabling verbose class loading logging

```

<appender name="UCL" class="org.apache.log4j.FileAppender">
  <param name="File" value="${jboss.server.home.dir}/log/uc1.log"/>
  <param name="Append" value="false"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="[%r,%c{1},%t] %m%n"/>
  </layout>
</appender>

<category name="org.jboss.mx.loading" additivity="false">
  <priority value="TRACE" class="org.jboss.logging.XLevel"/>
  <appender-ref ref="UCL"/>
</category>

```

This places the output from the classes in the **org.jboss.mx.loading** package into the **uc1.log** file of the server configurations log directory. Although it may not be meaningful if you have not looked at the class loading code, it is vital information needed for submitting bug reports or questions regarding class loading problems.

5.2.2.4. Inside the JBoss Class Loading Architecture

Now that we have the role of class loaders in the Java type system defined, let's take a look at the JBoss class loading architecture. [Figure 5.3, "The core JBoss class loading components"](#).

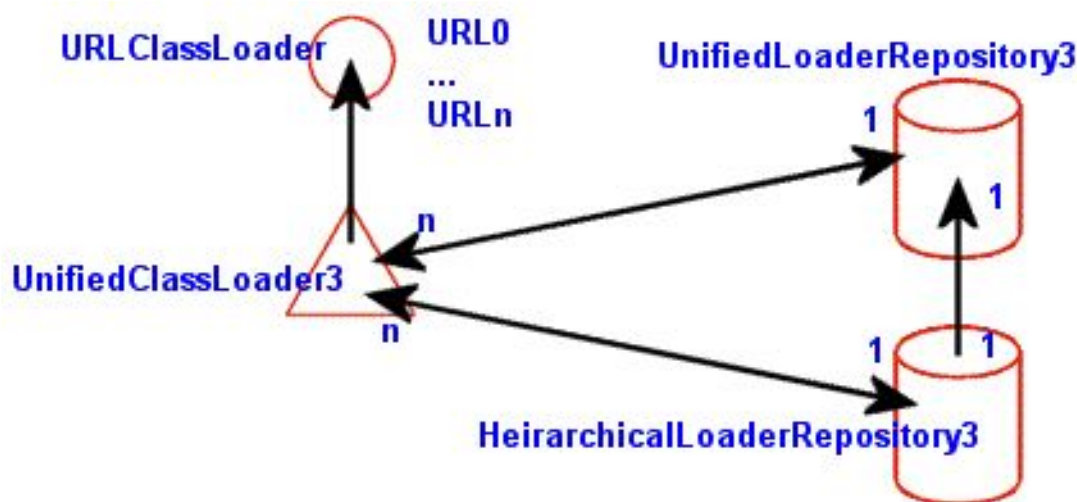


Figure 5.3. The core JBoss class loading components

The central component is the **org.jboss.mx.loading.UnifiedClassLoader3** (UCL) class loader. This is an extension of the standard **java.net.URLClassLoader** that overrides the standard parent delegation model to use a shared repository of classes and resources. This shared repository is the **org.jboss.mx.loading.UnifiedLoaderRepository3**. Every UCL is associated with a single **UnifiedLoaderRepository3**, and a **UnifiedLoaderRepository3** typically has many UCLs. A UCL may have multiple URLs associated with it for class and resource loading. Deployers use the top-level deployment's UCL as a shared class loader and all deployment archives are assigned to this class loader. We will talk about the JBoss deployers and their interaction with the class loading system in more detail later in [Section 5.4.2, "JBoss MBean Services"](#).

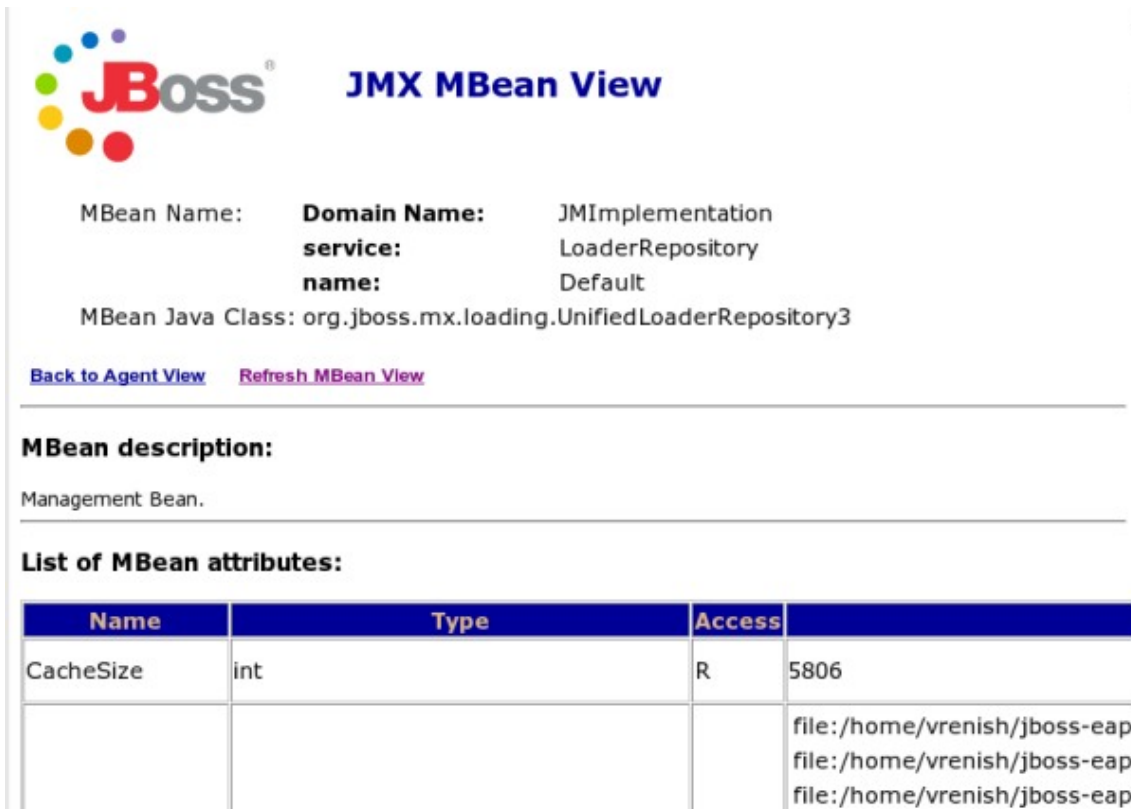
When a UCL is asked to load a class, it first looks to the repository cache it is associated with to see if

the class has already been loaded. Only if the class does not exist in the repository will it be loaded into the repository by the UCL. By default, there is a single **UnifiedLoaderRepository3** shared across all UCL instances. This means the UCLs form a single flat class loader namespace. The complete sequence of steps that occur when a **UnifiedClassLoader3.loadClass(String, boolean)** method is called is:

1. Check the **UnifiedLoaderRepository3** classes cache associated with the **UnifiedClassLoader3**. If the class is found in the cache it is returned.
2. Else, ask the **UnifiedClassLoader3** if it can load the class. This is essentially a call to the superclass **URLClassLoader.loadClass(String, boolean)** method to see if the class is among the URLs associated with the class loader, or visible to the parent class loader. If the class is found it is placed into the repository classes cache and returned.
3. Else, the repository is queried for all UCLs that are capable of providing the class based on the repository package name to UCL map. When a UCL is added to a repository an association between the package names available in the URLs associated with the UCL is made, and a mapping from package names to the UCLs with classes in the package is updated. This allows for a quick determination of which UCLs are capable of loading the class. The UCLs are then queried for the requested class in the order in which the UCLs were added to the repository. If a UCL is found that can load the class it is returned, else a **java.lang.ClassNotFoundException** is thrown.

5.2.2.4.1. Viewing Classes in the Loader Repository

Another useful source of information on classes is the **UnifiedLoaderRepository** itself. This is an MBean that contains operations to display class and package information. The default repository is located under a standard JMX name of **JMImplementation:name=Default,service=LoaderRepository**, and its MBean can be accessed via the JMX console by following its link from the front page. The JMX console view of this MBean is shown in [Figure 5.4, “The default class LoaderRepository MBean view in the JMX console”](#).



The screenshot shows the JBoss JMX MBean View for the `LoaderRepository` MBean. It displays the MBean Name, Domain Name, JMIImplementation, and MBean Java Class. Below this, there is a section for the MBean description and a table of MBean attributes.

MBean Name: `Domain Name:` `JMIImplementation`
service: `LoaderRepository`
name: `Default`
MBean Java Class: `org.jboss.mx.loading.UnifiedLoaderRepository3`

[Back to Agent View](#) [Refresh MBean View](#)

MBean description:
 Management Bean.

List of MBean attributes:

Name	Type	Access	
CacheSize	int	R	5806
			file:/home/vrenish/jboss-eap-4.3/jboss-as/server/production/deploy/ejb3.deployer/
			file:/home/vrenish/jboss-eap-4.3/jboss-as/server/production/deploy/jboss-messaging.sar/

Figure 5.4. The default class `LoaderRepository` MBean view in the JMX console

Two useful operations you will find here are `getPackageClassLoaders(String)` and `displayClassInfo(String)`. The `getPackageClassLoaders` operation returns a set of class loaders that have been indexed to contain classes or resources for the given package name. The package name must have a trailing period. If you type in the package name `org.jboss.ejb.`, the following information is displayed:

```
[org.jboss.mx.loading.UnifiedClassLoader3@1950198{
  url=null ,addedOrder=2},
org.jboss.mx.loading.UnifiedClassLoader3@89e2f1{
  url=file:/home/vrenish/jboss-eap-4.3/jboss-
as/server/production/deploy/ejb3.deployer/ ,addedOrder=3},
org.jboss.mx.loading.UnifiedClassLoader3@1555185{
  url=file:/home/vrenish/jboss-eap-4.3/jboss-
as/server/production/deploy/jboss-messaging.sar/ ,addedOrder=12}]
```

This is the string representation of the set. It shows three `UnifiedClassLoader3` instances. The primary url is indicated by the value shown in `url`. The order in which the class loader is added to the repository is indicated by the value shown in `addedOrder`. It is the class loader that owns all of the JARs in the `lib` directory of the server configuration (e.g., `server/production/lib`).

To view the information for a given class, use the `displayClassInfo` operation, passing in the fully qualified name of the class to view. For example, if we use `org.jboss.jmx.adaptor.html.HtmlAdaptorServlet` which is from the package we just looked at, the following description is displayed:

```
org.jboss.jmx.adaptor.html.HtmlAdaptorServlet Information
Not loaded in repository cache
```

```
### Instance0 via UCL: WebappClassLoader
```

```

delegate: false
repositories:
  /WEB-INF/classes/
-----> Parent Classloader:
java.net.FactoryURLClassLoader@2f5dda

```

The information is a dump of the information for the Class instance in the loader repository if one has been loaded, followed by the class loaders that are seen to have the class file available. If a class is seen to have more than one class loader associated with it, then there is the potential for class loading related errors.

5.2.2.4.2. Scoping Classes

If you need to deploy multiple versions of an application you need to use deployment based scoping. With deployment based scoping, each deployment creates its own class loader repository in the form of a **HeirarchicalLoaderRepository3** that looks first to the **UnifiedClassLoader3** instances of the deployment units included in the EAR before delegating to the default **UnifiedLoaderRepository3**. To enable an EAR specific loader repository, you need to create a **META-INF/jboss-app.xml** descriptor as shown in [Example 5.9, “An example jboss-app.xml descriptor for enabled scoped class loading at the EAR level.”](#).

Example 5.9. An example jboss-app.xml descriptor for enabled scoped class loading at the EAR level.

```

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

```

The value of the **loader-repository** element is the JMX object name to assign to the repository created for the EAR. This must be unique and valid JMX ObjectName, but the actual name is not important.



NOTE

JDK provided classes cannot be scoped. Meaning that a deployment cannot contain any JDK classes or (when using a war deployment) they must be excluded via the **FilteredPackages** attribute within the **jboss-service.xml** file.

5.2.2.4.3. The Complete Class Loading Model

The previous discussion of the core class loading components introduced the custom **UnifiedClassLoader3** and **UnifiedLoaderRepository3** classes that form a shared class loading space. The complete class loading picture must also include the parent class loader used by **UnifiedClassLoader3s** as well as class loaders introduced for scoping and other specialty class loading purposes. [Figure 5.5, “A complete class loader view”](#) shows an outline of the class hierarchy that would exist for an EAR deployment containing EJBs and WARs.

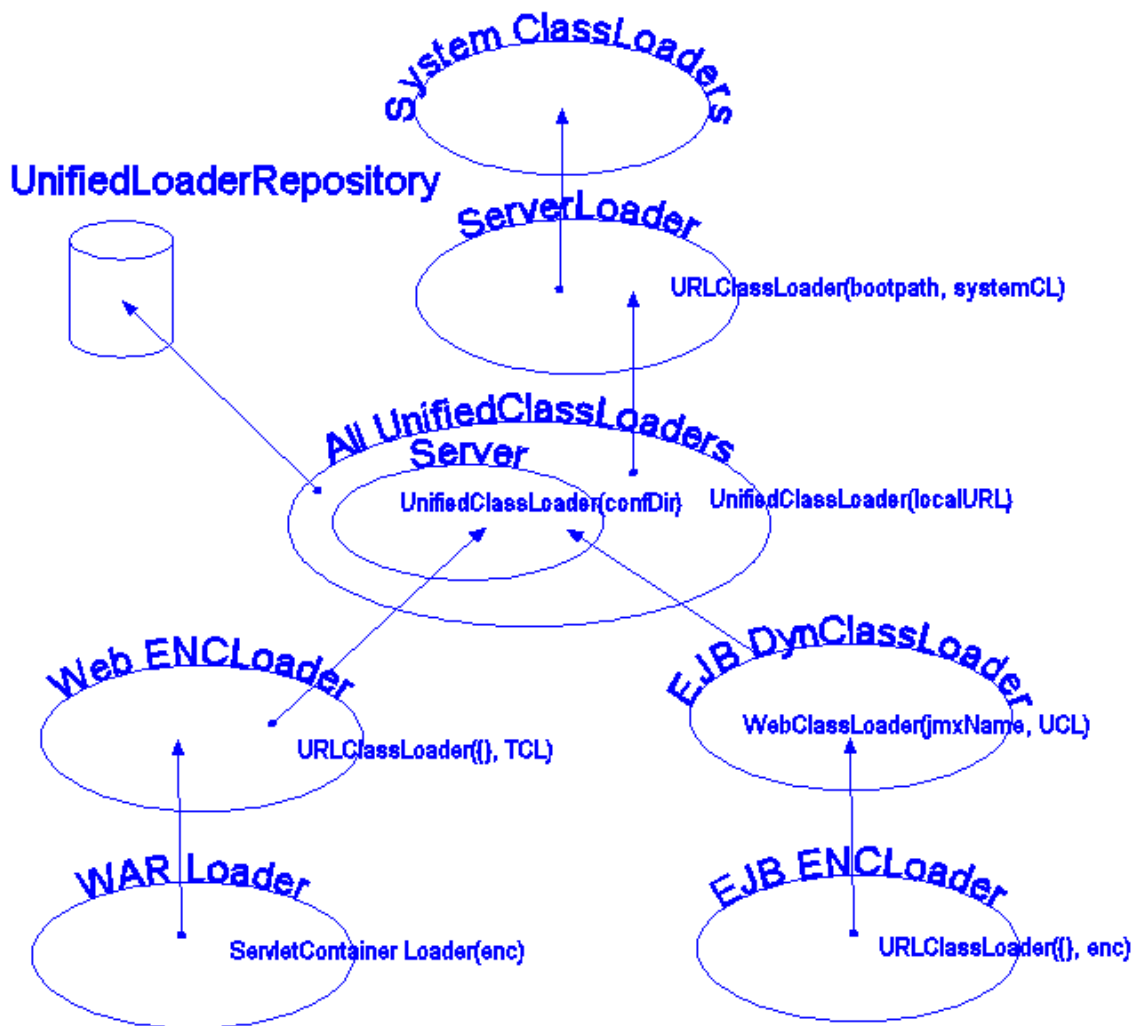


Figure 5.5. A complete class loader view

The following points apply to this figure:

- **System ClassLoaders:** The System ClassLoaders node refers to either the thread context class loader (TCL) of the VM main thread or of the thread of the application that is loading the JBoss server if it is embedded.
- **ServerLoader:** The ServerLoader node refers to the a **URLClassLoader** that delegates to the System ClassLoaders and contains the following boot URLs:
 - All URLs referenced via the **jboss.boot.library.list** system property. These are path specifications relative to the **libraryURL** defined by the **jboss.lib.url** property. If there is no **jboss.lib.url** property specified, it defaults to **jboss.home.url + /lib/**. If there is no **jboss.boot.library** property specified, it defaults to **jaxp.jar**, **log4j-boot.jar**, **jboss-common.jar**, and **jboss-system.jar**.
 - The JBoss JMX jar and GNU regex jar, **jboss-jmx.jar** and **gnu-regexp.jar**.
 - Oswego concurrency classes JAR, **concurrent.jar**
 - Any JARs specified as libraries via **-L** command line options
 - Any other JARs or directories specified via **-C** command line options

- **Server:** The *Server* node represents a collection of UCLs created by the `org.jboss.system.server.Server` interface implementation. The default implementation creates UCLs for the `patchDir` entries as well as the server `conf` directory. The last UCL created is set as the JBoss main thread context class loader. This will be combined into a single UCL now that multiple URLs per UCL are supported.
- **All UnifiedClassLoader3s:** The *All UnifiedClassLoader3* node represents the UCLs created by deployers. This covers EARs, jars, WARs, SARs and directories seen by the deployment scanner as well as JARs referenced by their manifests and any nested deployment units they may contain. This is a flat namespace and there should not be multiple instances of a class in different deployment JARs. If there are, only the first loaded will be used and the results may not be as expected. There is a mechanism for scoping visibility based on EAR deployment units that we discussed in [Section 5.2.2.4.2, "Scoping Classes"](#). Use this mechanism if you need to deploy multiple versions of a class in a given JBoss server.
- **EJB DynClassLoader:** The *EJB DynClassLoader* node is a subclass of `URLClassLoader` that is used to provide RMI dynamic class loading via the simple HTTP WebService. It specifies an empty `URL[]` and delegates to the TCL as its parent class loader. If the WebService is configured to allow system level classes to be loaded, all classes in the `UnifiedLoaderRepository3` as well as the system classpath are available via HTTP.
- **EJB ENCLoader:** The *EJB ENCLoader* node is a `URLClassLoader` that exists only to provide a unique context for an EJB deployment's `java:comp` JNDI context. It specifies an empty `URL[]` and delegates to the *EJB DynClassLoader* as its parent class loader.
- **Web ENCLoader:** The *Web ENCLoader* node is a `URLClassLoader` that exists only to provide a unique context for a web deployment's `java:comp` JNDI context. It specifies an empty `URL[]` and delegates to the TCL as its parent class loader.
- **WAR Loader:** The *WAR Loader* is a servlet container specific classloader that delegates to the *Web ENCLoader* as its parent class loader. The default behavior is to load from its parent class loader and then the WAR `WEB-INF/classes` and `lib` directories. If the servlet 2.3 class loading model is enabled it will first load from the its `WEB-INF` directories and then the parent class loader.

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

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

Disadvantages include:

- Existing deployments may need to be repackaged to avoid duplicate classes. Duplication of classes in a loader repository can lead to class cast exceptions and linkage errors depending on how the classes are loaded.
- Deployments that depend on different versions of a given class need to be isolated in separate EARs and a unique `HeirarchicalLoaderRepository3` defined using a `jboss-app.xml` descriptor.

5.2.3. JBoss XMBeans

XMBeans are the JBoss JMX implementation version of the JMX model MBean. XMBeans have the

richness of the dynamic MBean metadata without the tedious programming required by a direct implementation of the **DynamicMBean** interface. The JBoss model MBean implementation allows one to specify the management interface of a component through a XML descriptor, hence the X in XMBean. In addition to providing a simple mechanism for describing the metadata required for a dynamic MBean, XMBeans also allow for the specification of attribute persistence, caching behavior, and even advanced customizations like the MBean implementation interceptors. The high level elements of the **jboss_xmbean_1_2.dtd** for the XMBean descriptor is given in [Figure 5.6, “The JBoss 1.0 XMBean DTD Overview \(jboss_xmbean_1_2.dtd\)”](#).

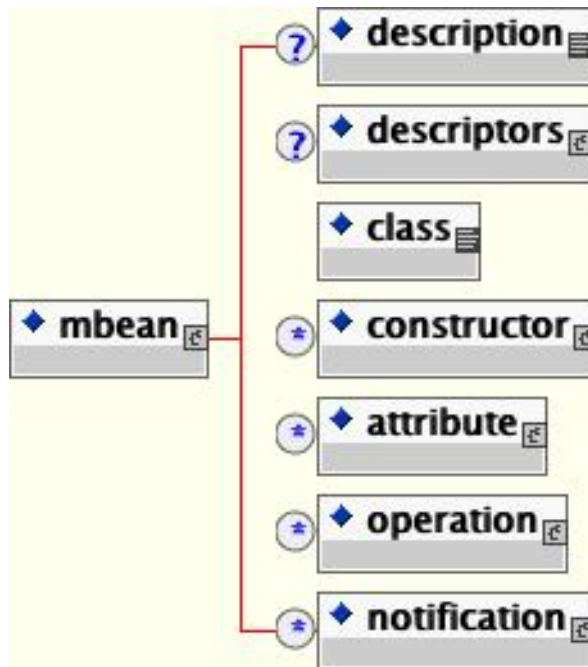


Figure 5.6. The JBoss 1.0 XMBean DTD Overview (jboss_xmbean_1_2.dtd)

The **mbean** element is the root element of the document containing the required elements for describing the management interface of one MBean (constructors, attributes, operations and notifications). It also includes an optional **description** element, which can be used to describe the purpose of the MBean, as well as an optional **descriptors** element which allows for persistence policy specification, attribute caching, etc.

5.2.3.1. Descriptors

The **descriptors** element contains all the descriptors for a containing element, as subelements. The descriptors suggested in the JMX specification as well as those used by JBoss have predefined elements and attributes, whereas custom descriptors have a generic descriptor element with **name** and **value** attributes as show in [Figure 5.7, “ The descriptors element content model”](#).

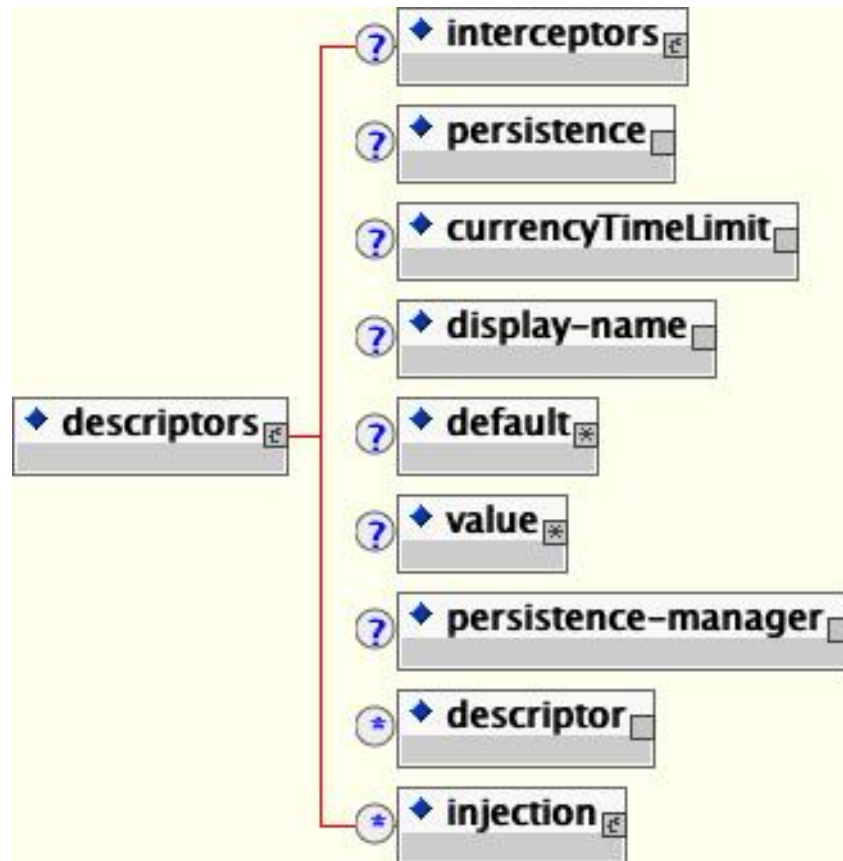


Figure 5.7. The descriptors element content model

The key descriptors child elements include:

- **interceptors:** The **interceptors** element specifies a customized stack of interceptors that will be used in place of the default stack. Currently this is only used when specified at the MBean level, but it could define a custom attribute or operation level interceptor stack in the future. The content of the interceptors element specifies a custom interceptor stack. If no interceptors element is specified the standard **ModelMBean** interceptors will be used. The standard interceptors are:
 - `org.jboss.mx.interceptor.PersistenceInterceptor`
 - `org.jboss.mx.interceptor.MBeanAttributeInterceptor`
 - `org.jboss.mx.interceptor.ObjectReferenceInterceptor`

When specifying a custom interceptor stack you would typically include the standard interceptors along with your own unless you are replacing the corresponding standard interceptor.

Each interceptor element content value specifies the fully qualified class name of the interceptor implementation. The class must implement the **org.jboss.mx.interceptor.Interceptor** interface. The interceptor class must also have either a no-arg constructor, or a constructor that accepts a **javax.management.MBeanInfo**.

The interceptor elements may have any number of attributes that correspond to JavaBean style properties on the interceptor class implementation. For each **interceptor** element attribute specified, the interceptor class is queried for a matching setter method. The attribute value is converted to the true type of the interceptor class property using the **java.beans.PropertyEditor** associated with the type. It is an error to specify an attribute for which there is no setter or **PropertyEditor**.

- **persistence**: The **persistence** element allows the specification of the **persistPolicy**, **persistPeriod**, **persistLocation**, and **persistName** persistence attributes suggested by the JMX specification. The persistence element attributes are:
 - **persistPolicy**: The **persistPolicy** attribute defines when attributes should be persisted and its value must be one of
 - **Never**: attribute values are transient values that are never persisted
 - **OnUpdate**: attribute values are persisted whenever they are updated
 - **OnTimer**: attribute values are persisted based on the time given by the **persistPeriod**.
 - **NoMoreOftenThan**: attribute values are persisted when updated but no more often than the **persistPeriod**.
 - **persistPeriod**: The **persistPeriod** attribute gives the update frequency in milliseconds if the **persistPolicy** attribute is **NoMoreOftenThan** or **OnTimer**.
 - **persistLocation**: The **persistLocation** attribute specifies the location of the persistence store. Its form depends on the JMX persistence implementation. Currently this should refer to a directory into which the attributes will be serialized if using the default JBoss persistence manager.
 - **persistName**: The **persistName** attribute can be used in conjunction with the **persistLocation** attribute to further qualify the persistent store location. For a directory **persistLocation** the **persistName** specifies the file to which the attributes are stored within the directory.
- **currencyTimeLimit**: The **currencyTimeLimit** element specifies the time in seconds that a cached value of an attribute remains valid. Its value attribute gives the time in seconds. A value of 0 indicates that an attribute value should always be retrieved from the MBean and never cached. A value of -1 indicates that a cache value is always valid.
- **display-name**: The **display-name** element specifies the human friendly name of an item.
- **default**: The **default** element specifies a default value to use when a field has not been set. Note that this value is not written to the MBean on startup as is the case with the **jboss-service.xml** attribute element content value. Rather, the default value is used only if there is no attribute accessor defined, and there is no value element defined.
- **value**: The **value** element specifies a management attribute's current value. Unlike the **default** element, the **value** element is written through to the MBean on startup provided there is a setter method available.
- **persistence-manager**: The **persistence-manager** element gives the name of a class to use as the persistence manager. The **value** attribute specifies the class name that supplies the **org.jboss.mx.persistence.PersistenceManager** interface implementation. The only implementation currently supplied by JBoss is the **org.jboss.mx.persistence.ObjectStreamPersistenceManager** which serializes the **ModelMBeanInfo** content to a file using Java serialization.

- **descriptor**: The **descriptor** element specifies an arbitrary descriptor not known to JBoss. Its **name** attribute specifies the type of the descriptor and its **value** attribute specifies the descriptor value. The **descriptor** element allows for the attachment of arbitrary management metadata.
- **injection**: The **injection** element describes an injection point for receiving information from the microkernel. Each injection point specifies the type and the set method to use to inject the information into the resource. The **injection** element supports type attributes:
 - **id**: The **id** attribute specifies the injection point type. The current injection point types are:
 - **MBeanServerType**: An *MBeanServerType* injection point receives a reference to the *MBeanServer* that the XMBBean is registered with.
 - **MBeanInfoType**: An *MBeanInfoType* injection point receives a reference to the XMBBean *ModelMBeanInfo* metadata.
 - **ObjectNameType**: The *ObjectName* injection point receives the *ObjectName* that the XMBBean is registered under.
- **setMethod**: The *setMethod* attribute gives the name of the method used to set the injection value on the resource. The set method should accept values of the type corresponding to the injection point type.

Note that any of the constructor, attribute, operation or notification elements may have a **descriptors** element to specify the specification defined descriptors as well as arbitrary extension descriptor settings.

5.2.3.2. The Management Class

The **class** element specifies the fully qualified name of the managed object whose management interface is described by the XMBBean descriptor.

5.2.3.3. The Constructors

The **constructor** element(s) specifies the constructors available for creating an instance of the managed object. The constructor element and its content model are shown in [Figure 5.8, “The XMBBean constructor element and its content model”](#).

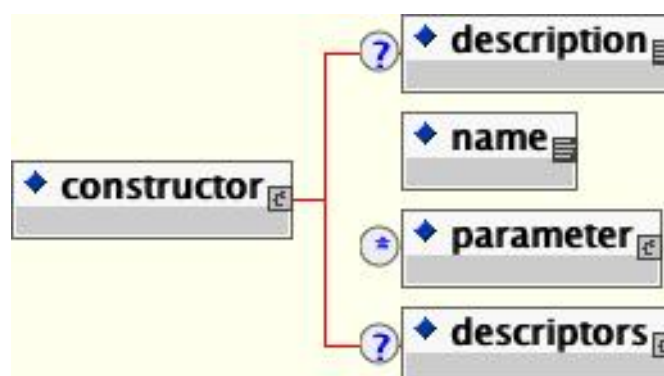


Figure 5.8. The XMBBean constructor element and its content model

The key child elements are:

- **description**: A description of the constructor.
- **name**: The name of the constructor, which must be the same as the implementation class.

- **parameter:** The parameter element describes a constructor parameter. The parameter element has the following attributes:
 - **description:** An optional description of the parameter.
 - **name:** The required variable name of the parameter.
 - **type:** The required fully qualified class name of the parameter type.
- **descriptors:** Any descriptors to associate with the constructor metadata.

5.2.3.4. The Attributes

The **attribute** element(s) specifies the management attributes exposed by the MBean. The attribute element and its content model are shown in Figure 5.9, “The XMLElement attribute element and its content model”.

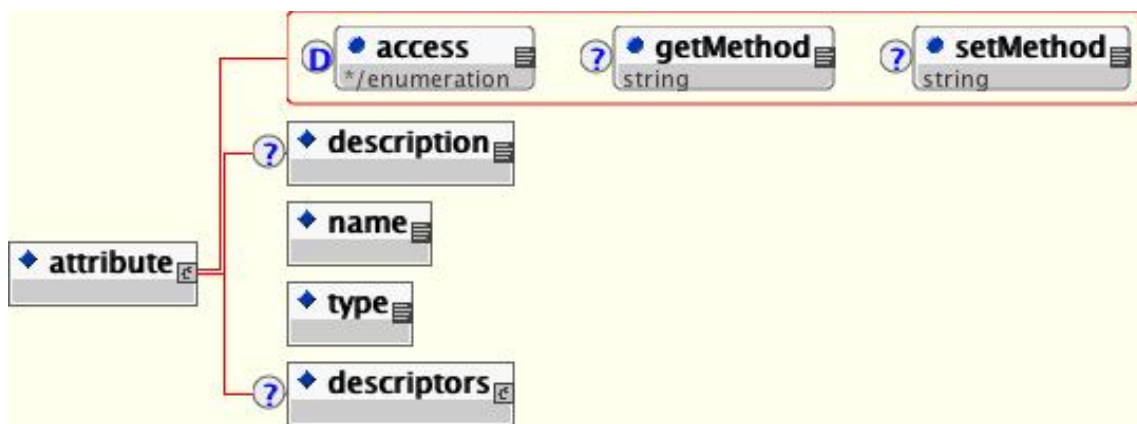


Figure 5.9. The XMLElement attribute element and its content model

The **attribute** element supported attributes include:

- **access:** The optional **access** attribute defines the read/write access modes of an attribute. It must be one of:
 - **read-only:** The attribute may only be read.
 - **write-only:** The attribute may only be written.
 - **read-write:** The attribute is both readable and writable. This is the implied default.
- **getMethod:** The **getMethod** attribute defines the name of the method which reads the named attribute. This must be specified if the managed attribute should be obtained from the MBean instance.
- **setMethod:** The **setMethod** attribute defines the name of the method which writes the named attribute. This must be specified if the managed attribute should be obtained from the MBean instance.

The key child elements of the attribute element include:

- **description:** A description of the attribute.
- **name:** The name of the attribute as would be used in the `MBeanServer.getAttribute()` operation.

- **type**: The fully qualified class name of the attribute type.
- **descriptors**: Any additional descriptors that affect the attribute persistence, caching, default value, etc.

5.2.3.5. The Operations

The management operations exposed by the XMBean are specified via one or more operation elements. The operation element and its content model are shown in [Figure 5.10, “The XMBean operation element and its content model”](#).

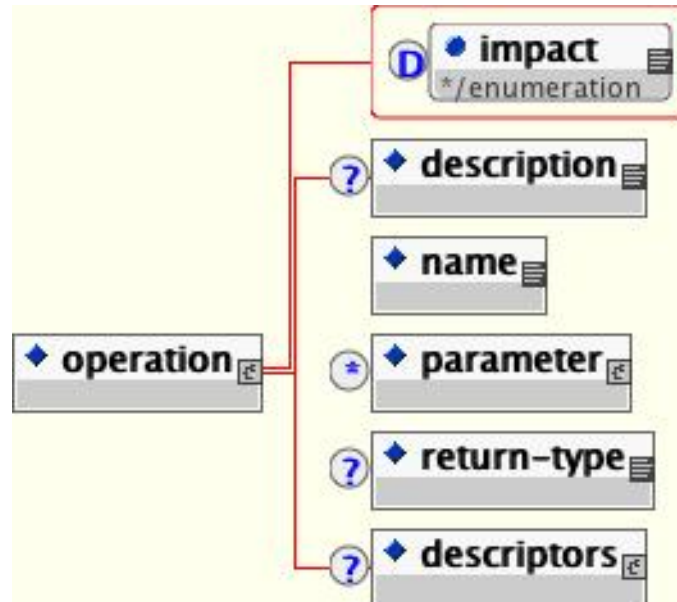


Figure 5.10. The XMBean operation element and its content model

The impact attribute defines the impact of executing the operation and must be one of:

- **ACTION**: The operation changes the state of the MBean component (write operation)
- **INFO**: The operation should not alter the state of the MBean component (read operation).
- **ACTION_INFO**: The operation behaves like a read/write operation.

The child elements are:

- **description**: This element specifies a human readable description of the operation.
- **name**: This element contains the operation's name
- **parameter**: This element describes the operation's signature.
- **return-type**: This element contains a fully qualified class name of the return type from this operation. If not specified, it defaults to void.
- **descriptors**: Any descriptors to associate with the operation metadata.

5.2.3.6. Notifications

The **notification** element(s) describes the management notifications that may be emitted by the XMBean. The notification element and its content model is shown in [Figure 5.11, “The XMBean notification element and content model”](#).

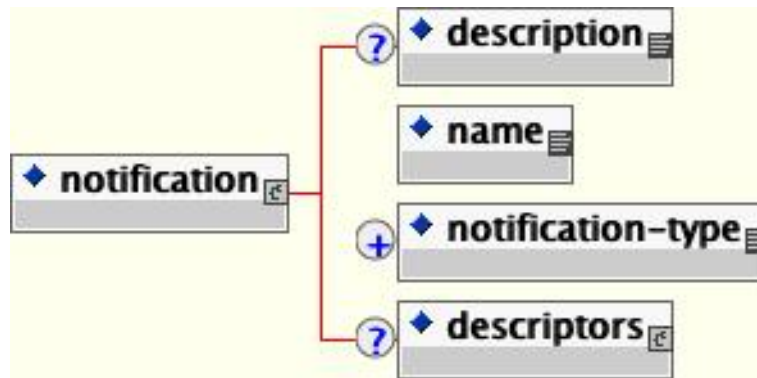


Figure 5.11. The XMBEAN notification element and content model

The child elements are:

- **description:** This element gives a human readable description of the notification.
- **name:** This element contains the fully qualified name of the notification class.
- **notification-type:** This element contains the dot-separated notification type string.
- **descriptors:** Any descriptors to associate with the notification metadata.

5.3. CONNECTING TO THE JMX SERVER

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

5.3.1. Inspecting the Server - the JMX Console Web Application

JBoss comes with its own implementation of a JMX HTML adaptor that allows one to view the server's MBeans using a standard web browser. The default URL for the console web application is <http://localhost:8080/jmx-console/>. If you browse this location you will see something similar to that presented in Figure 5.12, “The JBoss JMX console web application agent view”.

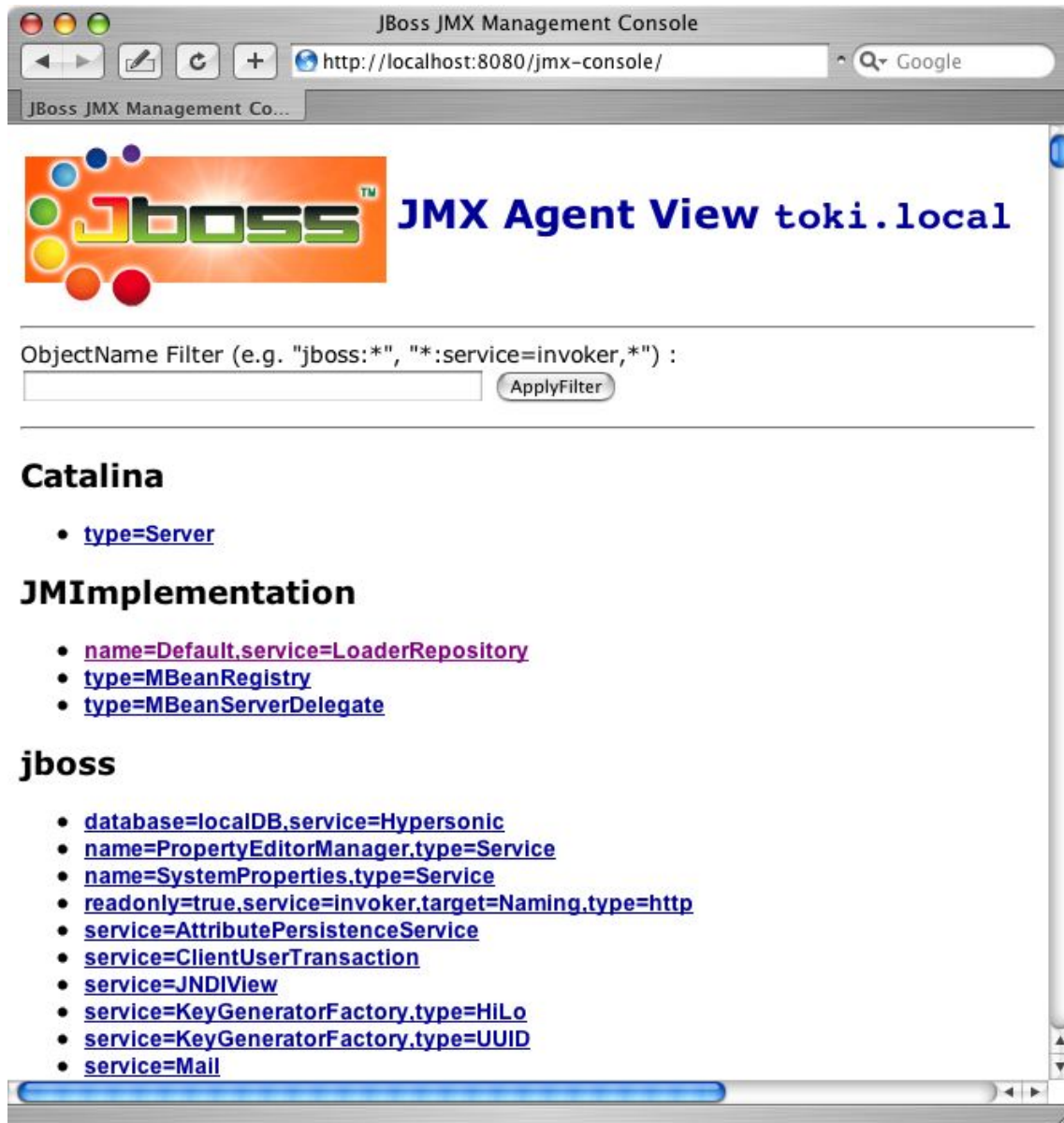


Figure 5.12. The JBoss JMX console web application agent view

The top view is called the agent view and it provides a listing of all MBeans registered with the **MBeanServer** sorted by the domain portion of the MBean's **ObjectName**. Under each domain are the MBeans under that domain. When you select one of the MBeans you will be taken to the MBean view. This allows one to view and edit an MBean's attributes as well as invoke operations. As an example, [Figure 5.13](#), "The MBean view for the "jboss.system:type=Server" MBean" shows the MBean view for the **jboss.system:type=Server** MBean.

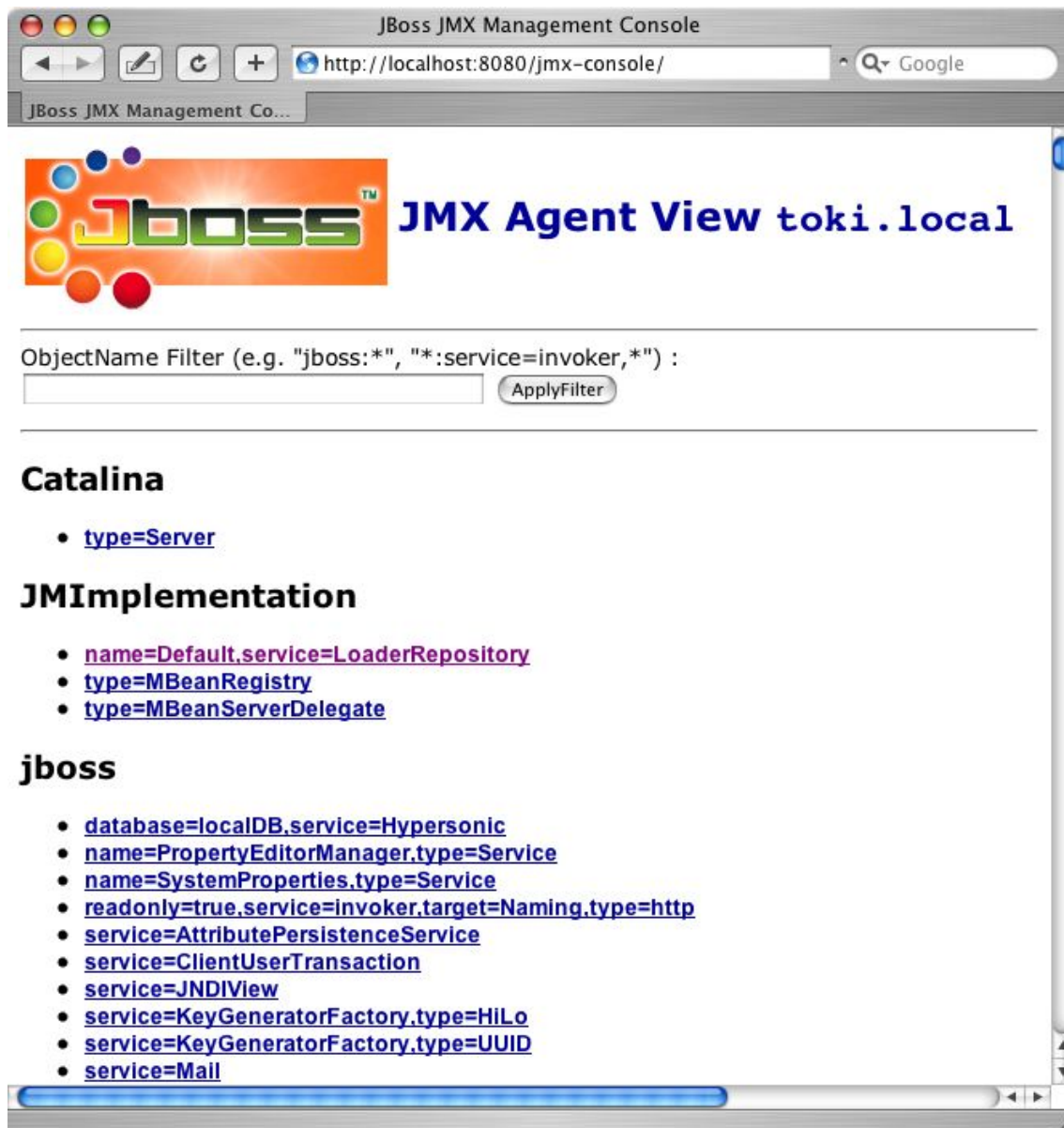


Figure 5.13. The MBean view for the "jboss.system:type=Server" MBean

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

5.3.1.1. Securing the JMX Console

Since the JMX console web application is just a standard servlet, it may be secured using standard J2EE role based security. The **jmx-console.war** that is deployed as an unpacked WAR that includes template settings for quickly enabling simple username and password based access restrictions. If you look at the **jmx-console.war** in the **server/production/deploy** directory you will find the **web.xml** and **jboss-web.xml** descriptors in the **WEB-INF** directory. The **jmx-console-roles.properties** and **jmx-console-users.properties** files are located in the **server/production/conf/props** directory.

By uncommenting the security sections of the **web.xml** and **jboss-web.xml** descriptors as shown in [Example 5.10, "The jmx-console.war web.xml descriptors with the security elements uncommented."](#), you enable HTTP basic authentication that restricts access to the JMX Console application to the user

admin with password **admin**. The username and password are determined by the **admin=admin** line in the **jmx-console-users.properties** file.

Example 5.10. The jmx-console.war web.xml descriptors with the security elements uncommented.

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

    <!-- A security constraint that restricts access to the HTML JMX
console
    to users with the role JBossAdmin. Edit the roles to what you
want and
    uncomment the WEB-INF/jboss-web.xml/security-domain element to
enable
    secured access to the HTML JMX console.
-->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HtmlAdaptor</web-resource-name>
            <description> An example security config that only allows
users with
                the role JBossAdmin to access the HTML JMX console web
                application </description>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>JBossAdmin</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>JBoss JMX Console</realm-name>
    </login-config>
    <security-role>
        <role-name>JBossAdmin</role-name>
    </security-role>
</web-app>
```

Example 5.11. The jmx-console.war jboss-web.xml descriptors with the security elements uncommented.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web
    PUBLIC "-//JBoss//DTD Web Application 2.3//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_3_0.dtd">
<jboss-web>
    <!--
```



```

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

```

Make these changes and then when you try to access the JMX Console URL, you will see a dialog similar to that shown in [Figure 5.14, “The JMX Console basic HTTP login dialog.”](#)



Figure 5.14. The JMX Console basic HTTP login dialog.

You probably to use the properties files for securing access to the JMX console application. To see how to properly configure the security settings of web applications see [Chapter 10, Security on JBoss](#).

5.3.2. Connecting to JMX Using RMI

JBoss supplies an RMI interface for connecting to the JMX MBeanServer. This interface is **org.jboss.jmx.adaptor.rmi.RMIAdaptor**. The **RMIAdaptor** interface is bound into JNDI in the default location of **jmx/invoker/RMIAdaptor** as well as **jmx/rmi/RMIAdaptor** for backwards compatibility with older clients.

[Example 5.12, “A JMX client that uses the RMIAdaptor”](#) shows a client that makes use of the **RMIAdaptor** interface to query the **MBeanInfo** for the **JNDIView** MBean. It also invokes the MBean's **list(boolean)** method and displays the result.

Example 5.12. A JMX client that uses the RMIAdaptor

```

public class JMXBrowser
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {

```



```

        InitialContext ic = new InitialContext();
        RMIAdaptor server = (RMIAdaptor)
ic.lookup("jmx/invoker/RMIAdaptor");

        // Get the MBeanInfo for the JNDIView MBean
        ObjectName name = new ObjectName("jboss:service=JNDIView");
        MBeanInfo info = server.getMBeanInfo(name);
        System.out.println("JNDIView Class: " + info.getClassName());

        MBeanOperationInfo[] opInfo = info.getOperations();
        System.out.println("JNDIView Operations: ");
        for(int o = 0; o < opInfo.length; o++) {
            MBeanOperationInfo op = opInfo[o];

            String returnType = op.getReturnType();
            String opName      = op.getName();
            System.out.print(" + " + returnType + " " + opName + "(");

            MBeanParameterInfo[] params = op.getSignature();
            for(int p = 0; p < params.length; p++) {
                MBeanParameterInfo paramInfo = params[p];

                String pname = paramInfo.getName();
                String type  = paramInfo.getType();

                if (pname.equals(type)) {
                    System.out.print(type);
                } else {
                    System.out.print(type + " " + name);
                }

                if (p < params.length-1) {
                    System.out.print(',');
                }
            }
            System.out.println(")");
        }

        // Invoke the list(boolean) op
        String[] sig      = {"boolean"};
        Object[] opArgs = {Boolean.TRUE};
        Object result = server.invoke(name, "list", opArgs, sig);

        System.out.println("JNDIView.list(true) output:\n"+result);
    }
}

```

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

```

[examples]$ ant -Dchap=jmx -Dex=4 run-example
...

run-example4:
[java] JNDIView Class: org.jboss.mx.modelmbean.XMBean

```

```

[java] JNDIView Operations:
[java] + java.lang.String list(boolean jboss:service=JNDIView)
[java] + java.lang.String listXML()
[java] + void create()
[java] + void start()
[java] + void stop()
[java] + void destroy()
[java] + void jbossInternalLifecycle(java.lang.String
jboss:service=JNDIView)
[java] + java.lang.String getName()
[java] + int getState()
[java] + java.lang.String getStateString()
[java] JNDIView.list(true) output:
[java] <h1>java: Namespace</h1>
[java] <pre>
[java] +- XAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
[java] +- DefaultDS (class: javax.sql.DataSource)
[java] +- SecurityProxyFactory (class:
org.jboss.security.SubjectSecurityProxyFactory)
[java] +- DefaultJMSProvider (class:
org.jboss.jms.jndi.JNDIProviderAdapter)
[java] +- comp (class: javax.naming.Context)
[java] +- JmsXA (class:
org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
[java] +- ConnectionFactory (class:
org.jboss.mq.SpyConnectionFactory)
[java] +- jaas (class: javax.naming.Context)
[java] | +- JmsXARealm (class:
org.jboss.security.plugins.SecurityDomainContext)
[java] | +- messaging (class:
org.jboss.security.plugins.SecurityDomainContext)
[java] | +- HsqlDbRealm (class:
org.jboss.security.plugins.SecurityDomainContext)
[java] +- timedCacheFactory (class: javax.naming.Context)
[java] Failed to lookup: timedCacheFactory, errmsg=null
[java] +- TransactionPropagationContextExporter (class:
org.jboss.tm.TransactionPropag
ationContextFactory)
[java] +- StdJMSPool (class:
org.jboss.jms.asf.StdServerSessionPoolFactory)
[java] +- Mail (class: javax.mail.Session)
[java] +- TransactionPropagationContextImporter (class:
org.jboss.tm.TransactionPropag
ationContextImporter)
[java] +- TransactionManager (class: org.jboss.tm.TxManager)
[java] </pre>
[java] <h1>Global JNDI Namespace</h1>
[java] <pre>
[java] +- XAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
[java] +- UIL2ConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.Lin
kRef)
[java] +- UserTransactionSessionFactory (proxy: $Proxy11 implements
interface org.jbos

```

```

s.tm.usertx.interfaces.UserTransactionSessionFactory)
    [java] +- HTTPConnectionFactory (class:
org.jboss.mq.SpyConnectionFactory)
    [java] +- console (class: org.jnp.interfaces.NamingContext)
    [java] | +- PluginManager (proxy: $Proxy36 implements interface
org.jboss.console.ma
nager.PluginManagerMBean)
    [java] +- UIL2XAConnectionFactory[link -> XAConnectionFactory]
(class: javax.naming
.LinkRef)
    [java] +- UUIDKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.uuid.UUID
KeyGeneratorFactory)
    [java] +- HTTPXAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
    [java] +- topic (class: org.jnp.interfaces.NamingContext)
    [java] | +- testDurableTopic (class: org.jboss.mq.SpyTopic)
    [java] | +- testTopic (class: org.jboss.mq.SpyTopic)
    [java] | +- securedTopic (class: org.jboss.mq.SpyTopic)
    [java] +- queue (class: org.jnp.interfaces.NamingContext)
    [java] | +- A (class: org.jboss.mq.SpyQueue)
    [java] | +- testQueue (class: org.jboss.mq.SpyQueue)
    [java] | +- ex (class: org.jboss.mq.SpyQueue)
    [java] | +- DLQ (class: org.jboss.mq.SpyQueue)
    [java] | +- D (class: org.jboss.mq.SpyQueue)
    [java] | +- C (class: org.jboss.mq.SpyQueue)
    [java] | +- B (class: org.jboss.mq.SpyQueue)
    [java] +- ConnectionFactory (class:
org.jboss.mq.SpyConnectionFactory)
    [java] +- UserTransaction (class:
org.jboss.tm.usertx.client.ClientUserTransaction)
    [java] +- jmx (class: org.jnp.interfaces.NamingContext)
    [java] | +- invoker (class: org.jnp.interfaces.NamingContext)
    [java] | | +- RMIAdaptor (proxy: $Proxy35 implements interface
org.jboss.jmx.adapt
or.rmi.RMIAdaptor, interface org.jboss.jmx.adaptor.rmi.RMIAdaptorExt)
    [java] | +- rmi (class: org.jnp.interfaces.NamingContext)
    [java] | | +- RMIAdaptor[link -> jmx/invoker/RMIAdaptor]
(class: javax.naming.L
inkRef)
    [java] +- HiLoKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.hilo.HiLo
KeyGeneratorFactory)
    [java] +- UILXAConnectionFactory[link -> XAConnectionFactory]
(class: javax.naming.
LinkRef)
    [java] +- UILConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.Link
Ref)
    [java] </pre>

```

5.3.3. Command Line Access to JMX

JBoss provides a simple command line tool that allows for interaction with a remote JMX server instance. This tool is called *twiddle* (for twiddling bits via JMX) and is located in the **bin** directory of the

distribution. Twiddle is a command execution tool, not a general command shell. It is run using either the **twiddle.sh** or **twiddle.bat** scripts, and passing in a **-h** (**--help**) argument provides the basic syntax, and **--help-commands** shows what you can do with the tool:

```
[bin]$ ./twiddle.sh -h
A JMX client to 'twiddle' with a remote JBoss server.

usage: twiddle.sh [options] <command> [command_arguments]

options:
  -h, --help                Show this help message
  --help-commands           Show a list of commands
  -H=<command>              Show command specific help
  -c=command.properties     Specify the command.properties file to use
  -D<name>[=<value>]        Set a system property
  --                        Stop processing options
  -s, --server=<url>        The JNDI URL of the remote server
  -a, --adapter=<name>      The JNDI name of the RMI adapter to use
  -q, --quiet               Be somewhat more quiet
```

5.3.3.1. Connecting twiddle to a Remote Server

By default the twiddle command will connect to the localhost at port 1099 to lookup the default **jmx/rmi/RMIAdaptor** binding of the **RMIAdaptor** service as the connector for communicating with the JMX server. To connect to a different server/port combination you can use the **-s** (**--server**) option:

```
[bin]$ ./twiddle.sh -s toki serverinfo -d jboss
[bin]$ ./twiddle.sh -s toki:1099 serverinfo -d jboss
```

To connect using a different RMIAdaptor binding use the **-a** (**--adapter**) option:

```
[bin]$ ./twiddle.sh -s toki -a jmx/rmi/RMIAdaptor serverinfo -d jboss
```

5.3.3.2. Sample twiddle Command Usage

To access basic information about a server, use the **serverinfo** command. This currently supports:

```
[bin]$ ./twiddle.sh -H serverinfo
Get information about the MBean server

usage: serverinfo [options]

options:
  -d, --domain              Get the default domain
  -c, --count               Get the MBean count
  -l, --list                List the MBeans
  --                        Stop processing options
[bin]$ ./twiddle.sh --server=toki serverinfo --count
460
[bin]$ ./twiddle.sh --server=toki serverinfo --domain
jboss
```

To query the server for the name of MBeans matching a pattern, use the **query** command. This currently supports:

```
[bin]$ ./twiddle.sh -H query
Query the server for a list of matching MBeans

usage: query [options] <query>
options:
    -c, --count      Display the matching MBean count
    --              Stop processing options
Examples:
    query all mbeans: query '*:*'
    query all mbeans in the jboss.j2ee domain: query 'jboss.j2ee:*'
[bin]$ ./twiddle.sh -s toki query 'jboss:service=invoker,*'
jboss:readonly=true,service=invoker,target=Naming,type=http
jboss:service=invoker,type=jrmp
jboss:service=invoker,type=local
jboss:service=invoker,type=pooled
jboss:service=invoker,type=http
jboss:service=invoker,target=Naming,type=http
```

To get the attributes of an MBean, use the get command:

```
[bin]$ ./twiddle.sh -H get
Get the values of one or more MBean attributes

usage: get [options] <name> [<attr>+]
    If no attribute names are given all readable attributes are retrieved
options:
    --noprefix      Do not display attribute name prefixes
    --              Stop processing options
[bin]$ ./twiddle.sh get jboss:service=invoker,type=jrmp RMIObjectPort
StateString
RMIObjectPort=4444
StateString=Started
[bin]$ ./twiddle.sh get jboss:service=invoker,type=jrmp
ServerAddress=0.0.0.0
RMIClientSocketFactoryBean=null
StateString=Started
State=3
RMIServerSocketFactoryBean=org.jboss.net.sockets.DefaultSocketFactory@ad093076
EnableClassCaching=false
SecurityDomain=null
RMIServerSocketFactory=null
Backlog=200
RMIObjectPort=4444
Name=JRMPInvoker
RMIClientSocketFactory=null
```

To query the MBeanInfo for an MBean, use the info command:

```
[bin]$ ./twiddle.sh -H info
Get the metadata for an MBean
```

```
usage: info <mbean-name>
  Use '*' to query for all attributes
[bin]$ Description: Management Bean.
+++ Attributes:
  Name: ServerAddress
  Type: java.lang.String
  Access: rw
  Name: RMIClientSocketFactoryBean
  Type: java.rmi.server.RMIClientSocketFactory
  Access: rw
  Name: StateString
  Type: java.lang.String
  Access: r-
  Name: State
  Type: int
  Access: r-
  Name: RMIServerSocketFactoryBean
  Type: java.rmi.server.RMIServerSocketFactory
  Access: rw
  Name: EnableClassCaching
  Type: boolean
  Access: rw
  Name: SecurityDomain
  Type: java.lang.String
  Access: rw
  Name: RMIServerSocketFactory
  Type: java.lang.String
  Access: rw
  Name: Backlog
  Type: int
  Access: rw
  Name: RMIObjectPort
  Type: int
  Access: rw
  Name: Name
  Type: java.lang.String
  Access: r-
  Name: RMIClientSocketFactory
  Type: java.lang.String
  Access: rw
+++ Operations:
  void start()
  void jbossInternalLifecycle(java.lang.String java.lang.String)
  void create()
  void stop()
  void destroy()
```

To invoke an operation on an MBean, use the invoker command:

```
[bin]$ ./twiddle.sh -H invoke
Invoke an operation on an MBean

usage: invoke [options] <query> <operation> (<arg>)*

options:
  -q, --query-type[=<type>]    Treat object name as a query
```

--

Stop processing options

query type:

f[first] Only invoke on the first matching name [default]

a[all] Invoke on all matching names

`[bin]$./twiddle.sh invoke jboss:service=JNDIView list true`

<h1>java: Namespace</h1>

<pre>

+- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)

+- DefaultDS (class: javax.sql.DataSource)

+- SecurityProxyFactory (class:

org.jboss.security.SubjectSecurityProxyFactory)

+- DefaultJMSProvider (class: org.jboss.jms.jndi.JNDIProviderAdapter)

+- comp (class: javax.naming.Context)

+- JmsXA (class:

org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)

+- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)

+- jaas (class: javax.naming.Context)

| +- JmsXARealm (class:

org.jboss.security.plugins.SecurityDomainContext)

| +- messaging (class:

org.jboss.security.plugins.SecurityDomainContext)

| +- HsqlDbRealm (class:

org.jboss.security.plugins.SecurityDomainContext)

+- timedCacheFactory (class: javax.naming.Context)

Failed to lookup: timedCacheFactory, errmsg=null

+- TransactionPropagationContextExporter (class:

org.jboss.tm.TransactionPropagationContext

Factory)

+- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)

+- Mail (class: javax.mail.Session)

+- TransactionPropagationContextImporter (class:

org.jboss.tm.TransactionPropagationContext

Importer)

+- TransactionManager (class: org.jboss.tm.TxManager)

</pre>

<h1>Global JNDI Namespace</h1>

<pre>

+- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)

+- UIL2ConnectionFactory[link -> ConnectionFactory] (class:

javax.naming.LinkRef)

+- UserTransactionSessionFactory (proxy: \$Proxy11 implements interface

org.jboss.tm.usertx.

interfaces.UserTransactionSessionFactory)

+- HTTPConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)

+- console (class: org.jnp.interfaces.NamingContext)

| +- PluginManager (proxy: \$Proxy36 implements interface

org.jboss.console.manager.Plugin

ManagerMBean)

+- UIL2XAConnectionFactory[link -> XAConnectionFactory] (class:

javax.naming.LinkRef)

+- UUIDKeyGeneratorFactory (class:

org.jboss.ejb.plugins.keygenerator.uuid.UUIDKeyGenerator

Factory)

+- HTTPXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)

+- topic (class: org.jnp.interfaces.NamingContext)

```

|   +- testDurableTopic (class: org.jboss.mq.SpyTopic)
|   +- testTopic (class: org.jboss.mq.SpyTopic)
|   +- securedTopic (class: org.jboss.mq.SpyTopic)
+- queue (class: org.jnp.interfaces.NamingContext)
|   +- A (class: org.jboss.mq.SpyQueue)
|   +- testQueue (class: org.jboss.mq.SpyQueue)
|   +- ex (class: org.jboss.mq.SpyQueue)
|   +- DLQ (class: org.jboss.mq.SpyQueue)
|   +- D (class: org.jboss.mq.SpyQueue)
|   +- C (class: org.jboss.mq.SpyQueue)
|   +- B (class: org.jboss.mq.SpyQueue)
+- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
+- UserTransaction (class:
org.jboss.tm.usertx.client.ClientUserTransaction)
  +- jmx (class: org.jnp.interfaces.NamingContext)
    |   +- invoker (class: org.jnp.interfaces.NamingContext)
    |   |   +- RMIAdaptor (proxy: $Proxy35 implements interface
org.jboss.jmx.adaptor.rmi.RMIAd
aport,interface org.jboss.jmx.adaptor.rmi.RMIAdaptorExt)
    |   |   +- rmi (class: org.jnp.interfaces.NamingContext)
    |   |   |   +- RMIAdaptor[link -> jmx/invoker/RMIAdaptor] (class:
javax.naming.LinkRef)
    |   |   |   +- HiLoKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.hilo.HiLoKeyGenerator
Factory)
    |   |   |   +- UILXAConnectionFactory[link -> XAConnectionFactory] (class:
javax.naming.LinkRef)
    |   |   |   +- UILConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.LinkRef)
  </pre>

```

5.3.4. Connecting to JMX Using Any Protocol

With the detached invokers and a somewhat generalized proxy factory capability, you can really talk to the JMX server using the **InvokerAdaptorService** and a proxy factory service to expose an **RMIAdaptor** or similar interface over your protocol of choice. We will introduce the detached invoker notion along with proxy factories in [Section 5.6, “Remote Access to Services, Detached Invokers”](#). See [Section 5.6.1, “A Detached Invoker Example, the MBeanServer Invoker Adaptor Service”](#) for an example of an invoker service that allows one to access the MBean server using the **RMIAdaptor** interface over any protocol for which a proxy factory service exists.

5.4. USING JMX AS A MICROKERNEL

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. All other manageable MBean components are plugged into JBoss by registering with the MBean server. The kernel in that sense is only an framework, 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.

5.4.1. The Startup Process

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

1. The run start script initiates the boot sequence using the **org.jboss.Main.main(String[])** method entry point.
2. The **Main.main** method creates a thread group named **jboss** and then starts a thread belonging to this thread group. This thread invokes the **Main.boot** method.
3. The **Main.boot** method processes the **Main.main** arguments and then creates an **org.jboss.system.server.ServerLoader** using the system properties along with any additional properties specified as arguments.
4. The XML parser libraries, **jboss-jmx.jar**, **concurrent.jar** and extra libraries and classpaths given as arguments are registered with the **ServerLoader**.
5. The JBoss server instance is created using the **ServerLoader.load(ClassLoader)** method with the current thread context class loader passed in as the **ClassLoader** argument. The returned server instance is an implementation of the **org.jboss.system.server.Server** interface. The creation of the server instance entails:
 - Creating a **java.net.URLClassLoader** with the URLs of the jars and directories registered with the **ServerLoader**. This **URLClassLoader** uses the **ClassLoader** passed in as its parent and it is pushed as the thread context class loader.
 - The class name of the implementation of the **Server** interface to use is determined by the **jboss.server.type** property. This defaults to **org.jboss.system.server.ServerImpl**.
 - The **Server** implementation class is loaded using the **URLClassLoader** created in step 6 and instantiated using its no-arg constructor. The thread context class loader present on entry into the **ServerLoader.load** method is restored and the server instance is returned.
6. The server instance is initialized with the properties passed to the **ServerLoader** constructor using the **Server.init(Properties)** method.
7. The server instance is then started using the **Server.start()** method. The default implementation performs the following steps:
 - Set the thread context class loader to the class loader used to load the **ServerImpl** class.
 - Create an **MBeanServer** under the **jboss** domain using the **MBeanServerFactory.createMBeanServer(String)** method.
 - Register the **ServerImpl** and **ServerConfigImpl** MBeans with the MBean server.
 - Initialize the unified class loader repository to contain all JARs in the optional patch directory as well as the server configuration file conf directory, for example, **server/production/conf**. For each JAR and directory an **org.jboss.mx.loading.UnifiedClassLoader** is created and registered with the unified repository. One of these **UnifiedClassLoader** is then set as the thread context class loader. This effectively makes all **UnifiedClassLoaders** available through the thread context class loader.
 - The **org.jboss.system.ServiceController** MBean is created. The **ServiceController** manages the JBoss MBean services life cycle. We will discuss the JBoss MBean services notion in detail in [Section 5.4.2, “JBoss MBean Services”](#).

- The **org.jboss.deployment.MainDeployer** is created and started. The **MainDeployer** manages deployment dependencies and directing deployments to the correct deployer.
- The **org.jboss.deployment.JARDeployer** is created and started. The **JARDeployer** handles the deployment of JARs that are simple library JARs.
- The **org.jboss.deployment.SARDeployer** is created and started. The **SARDeployer** handles the deployment of JBoss MBean services.
- The **MainDeployer** is invoked to deploy the services defined in the **conf/jboss-service.xml** of the current server file set.
- Restore the thread context class loader.

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

5.4.2. JBoss MBean Services

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

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

5.4.2.1. The SARDeployer MBean

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

The **SARDeployer** handles services archives. A service archive can be either a jar that ends with a **.sar** suffix and contains a **META-INF/jboss-service.xml** descriptor, or a standalone XML descriptor with a naming pattern that matches ***-service.xml**. The DTD for the service descriptor is **jboss-service_4.2.dtd** and is shown in [Figure 5.15, “The DTD for the MBean service descriptor parsed by the SARDeployer”](#).

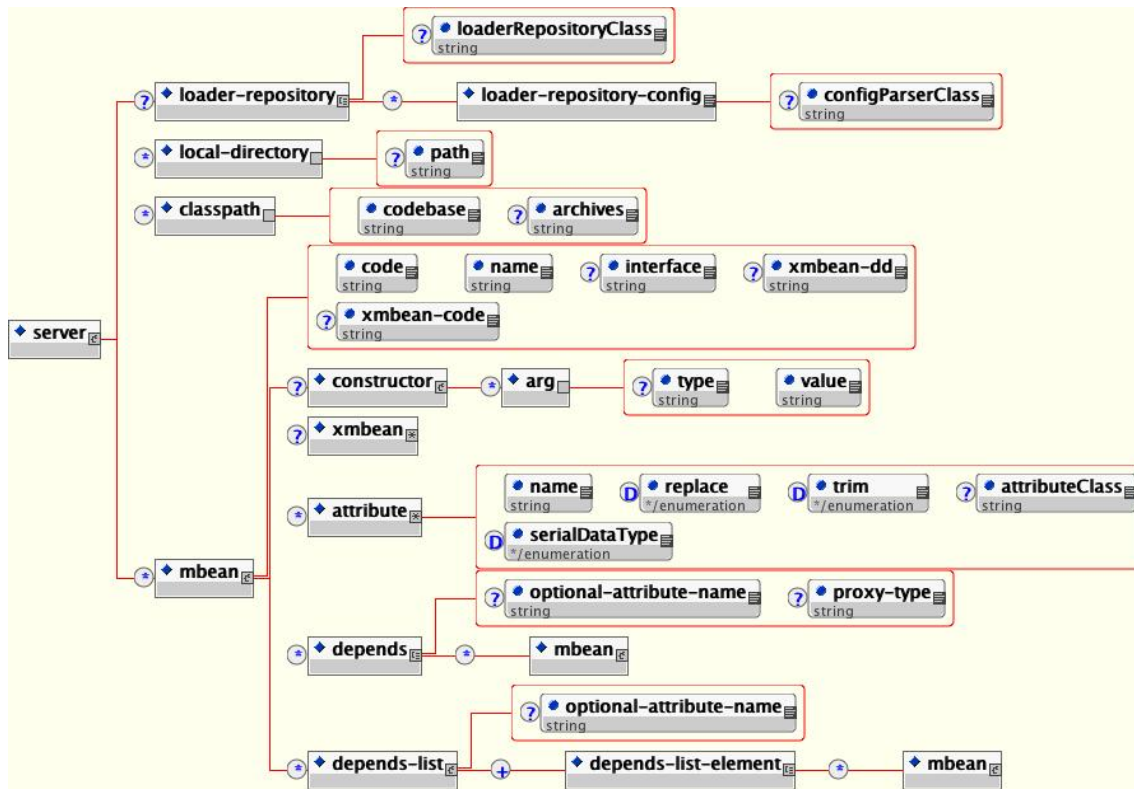


Figure 5.15. The DTD for the MBean service descriptor parsed by the SARDeployer

The elements of the DTD are:

- **loader-repository**: This element specifies the name of the **UnifiedLoaderRepository** MBean to use for the SAR to provide SAR level scoping of classes deployed in the sar. It is a unique JMX **ObjectName** string. It may also specify an arbitrary configuration by including a **loader-repository-config** element. The optional **loaderRepositoryClass** attribute specifies the fully qualified name of the loader repository implementation class. It defaults to **org.jboss.mx.loading.HeirachicalLoaderRepository3**.
 - **loader-repository-config**: This optional element specifies an arbitrary configuration that may be used to configure the **loadRepositoryClass**. The optional **configParserClass** attribute gives the fully qualified name of the **org.jboss.mx.loading.LoaderRepositoryFactory.LoaderRepositoryConfigParser** implementation to use to parse the **loader-repository-config** content.
- **local-directory**: This element specifies a path within the deployment archive that should be copied to the **server/<config>/db** directory for use by the MBean. The path attribute is the name of an entry within the deployment archive.
- **classpath**: This element specifies one or more external JARs that should be deployed with the MBean(s). The optional archives attribute specifies a comma separated list of the JAR names to load, or the * wild card to signify that all jars should be loaded. The wild card only works with file URLs, and http URLs if the web server supports the WEBDAV protocol. The codebase attribute specifies the URL from which the JARs specified in the archive attribute should be loaded. If the codebase is a path rather than a URL string, the full URL is built by treating the codebase value as a path relative to the JBoss distribution **server/<config>** directory. The order of JARs specified in the archives as well as the ordering across multiple classpath element is used as the classpath ordering of the JARs. Therefore, if you have patches or inconsistent versions of classes that require a certain ordering, use this feature to ensure the correct ordering.
- **mbean**: This element specifies an MBean service. The required code attribute gives the fully

qualified name of the MBean implementation class. The required `name` attribute gives the JMX **ObjectName** of the MBean. The optional `xmbean-dd` attribute specifies the path to the XMBean resource if this MBean service uses the JBoss XMBean descriptor to define a Model MBean management interface.

- **constructor**: The **constructor** element defines a non-default constructor to use when instantiating the MBean. The **arg** element specifies the constructor arguments in the order of the constructor signature. Each **arg** has a **type** and **value** attribute.
- **attribute**: Each attribute element specifies a name/value pair of the attribute of the MBean. The name of the attribute is given by the `name` attribute, and the attribute element body gives the value. The body may be a text representation of the value, or an arbitrary element and child elements if the type of the MBean attribute is `org.w3c.dom.Element`. For text values, the text is converted to the attribute type using the JavaBean `java.beans.PropertyEditor` mechanism.
- **server/mbean/depends** and **server/mbean/depends-list**: these elements specify a dependency from the MBean using the element to the MBean(s) named by the **depends** or **depends-list** elements. [Section 5.4.2.4, “Specifying Service Dependencies”](#). Note that the dependency value can be another mbean element which defines a nested mbean.

MBean attribute values don't need to be hardcoded literal strings. Service files may contain references to system properties using the `${name}` notation, where **name** is the name of a Java system property. The value of this system property, as would be returned from the call `System.getProperty("name")`. Multiple properties can be specified separated by commas like `${name1,name2,name3}`. If there is no system property named **name1**, **name2** will be tried and then **name3**. This allows multiple levels of substitution to be used. Finally, a default value can be added using a colon separator. The substitution `${name:default value}` would substitute the text `"default value"` if the system property **name** didn't exist. If none of the listed properties exist and no default value is given, no substitution will occur.

When the **SARDeployer** is asked to deploy a service performs several steps. [Figure 5.16, “A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service”](#) is a sequence diagram that shows the init through start phases of a service.

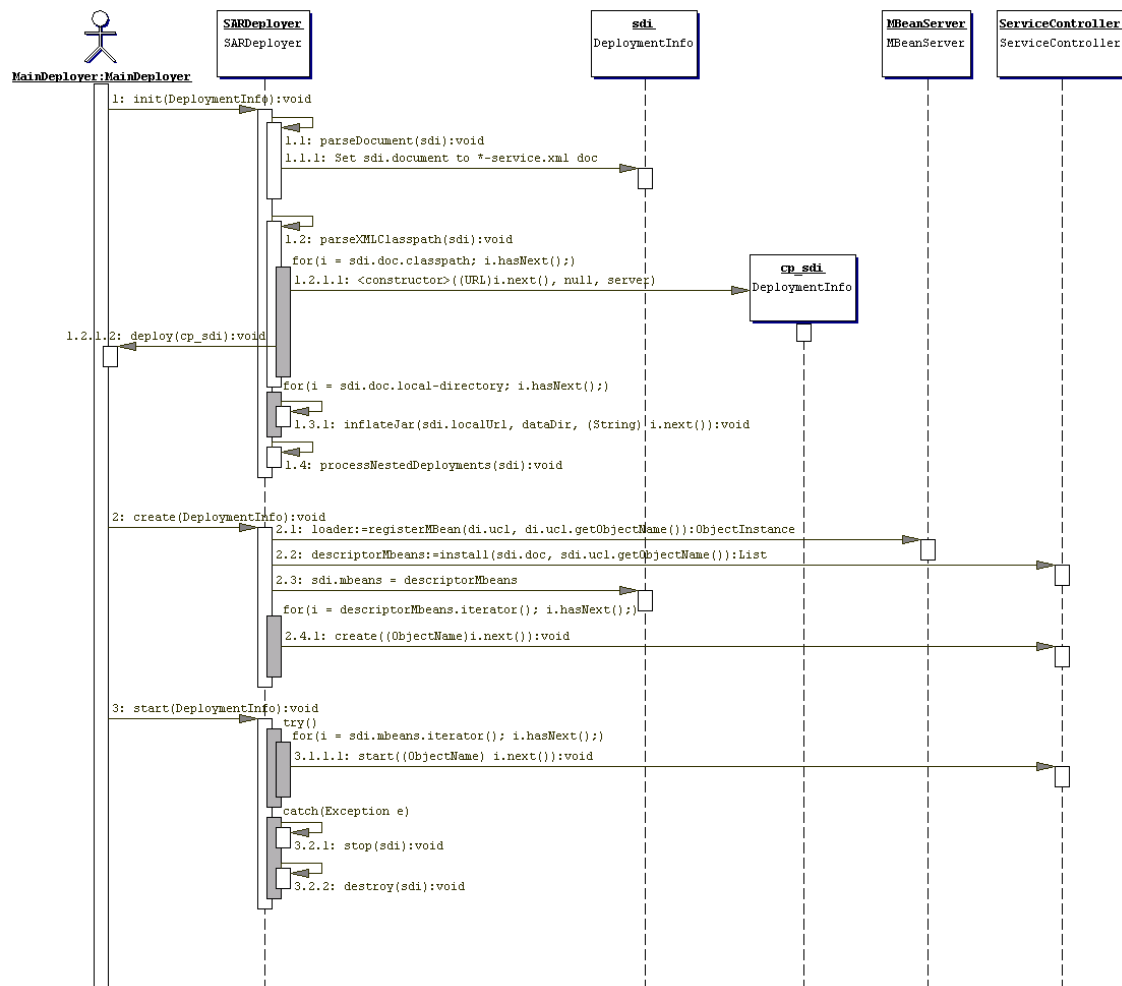


Figure 5.16. A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service

In Figure 5.16, “A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service” the following is illustrated:

- Methods prefixed with 1.1 correspond to the load and parse of the XML service descriptor.
- Methods prefixed with 1.2 correspond to processing each classpath element in the service descriptor to create an independent deployment that makes the jar or directory available through a **UnifiedClassLoader** registered with the unified loader repository.
- Methods prefixed with 1.3 correspond to processing each **local-directory** element in the service descriptor. This does a copy of the SAR elements specified in the path attribute to the **server/<config>/db** directory.
- Method 1.4. Process each deployable unit nested in the service a child deployment is created and added to the service deployment info subdeployment list.
- Method 2.1. The **UnifiedClassLoader** of the SAR deployment unit is registered with the MBean Server so that is can be used for loading of the SAR MBeans.
- Method 2.2. For each MBean element in the descriptor, create an instance and initialize its attributes with the values given in the service descriptor. This is done by calling the **ServiceController.install** method.
- Method 2.4.1. For each MBean instance created, obtain its JMX **ObjectName** and ask the ServiceController to handle the create step of the service life cycle. The **ServiceController**

handles the dependencies of the MBean service. Only if the service's dependencies are satisfied is the service create method invoked.

- Methods prefixed with 3.1 correspond to the start of each MBean service defined in the service descriptor. For each MBean instance created, obtain its JMX ObjectName and ask the **ServiceController** to handle the start step of the service life cycle. The **ServiceController** handles the dependencies of the MBean service. Only if the service's dependencies are satisfied is the service start method invoked.

5.4.2.2. The Service Life Cycle Interface

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

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

The **ServiceController** MBean invokes the methods of the **Service** interface at the appropriate times of the service life cycle. We'll discuss the methods in more detail in the **ServiceController** section.

5.4.2.3. The ServiceController MBean

JBoss manages dependencies between MBeans via the **org.jboss.system.ServiceController** custom MBean. The SARDeployer delegates to the ServiceController when initializing, creating, starting, stopping and destroying MBean services. [Figure 5.17, “The interaction between the SARDeployer and ServiceController to start a service”](#) shows a sequence diagram that highlights interaction between the **SARDeployer** and **ServiceController**.

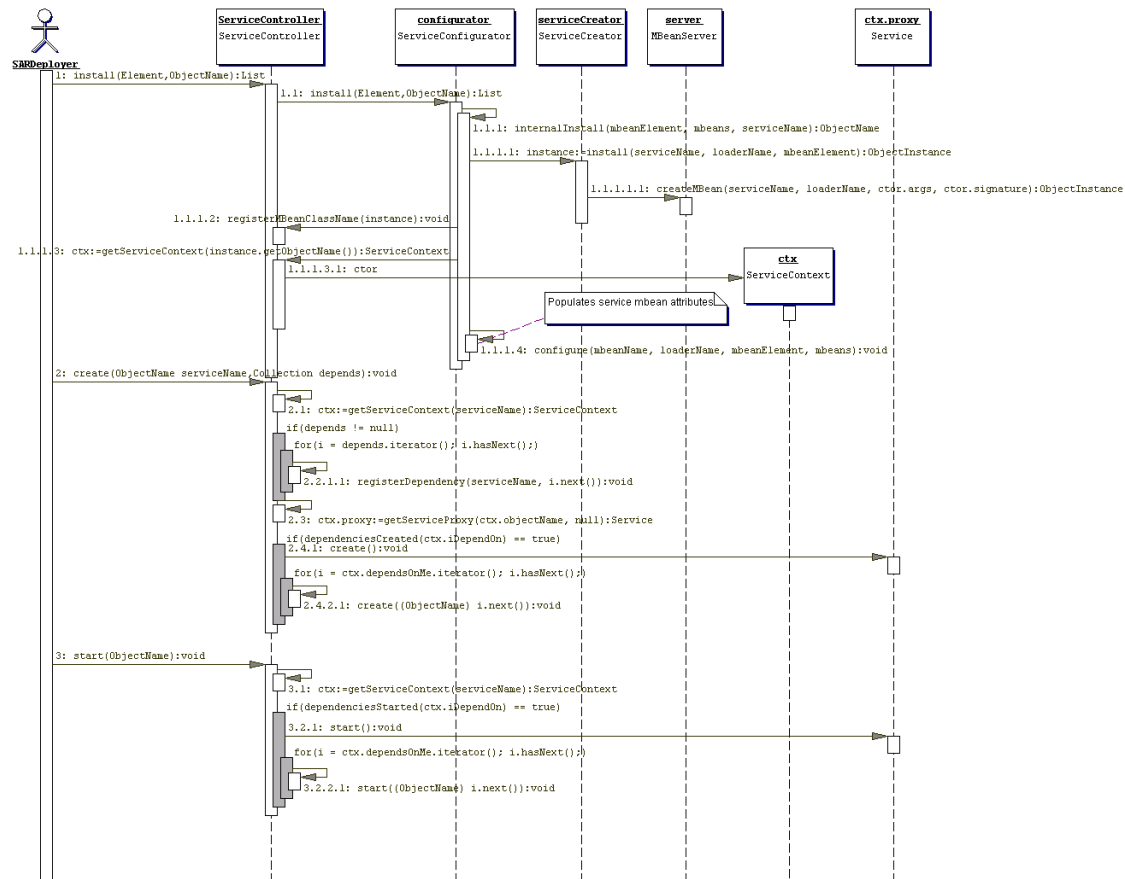


Figure 5.17. The interaction between the SARDeployer and ServiceController to start a service

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

5.4.2.3.1. The create(ObjectName) method

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

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

5.4.2.3.2. The start(ObjectName) method

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

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

5.4.2.3.3. The stop(ObjectName) method

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

5.4.2.3.4. The destroy(ObjectName) method

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

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

5.4.2.4. Specifying Service Dependencies

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

```
<mbean code="org.jboss.mq.server.jmx.Topic"
      name="jms.topic:service=Topic,name=testTopic">
  <!-- Declare a dependency on the "jboss.mq:service=DestinationManager"
and
      bind this name to the DestinationManager attribute -->
  <depends optional-attribute-name="DestinationManager">
    jboss.mq:service=DestinationManager
  </depends>

  <!-- Declare a dependency on the "jboss.mq:service=SecurityManager"
and
      bind this name to the SecurityManager attribute -->
  <depends optional-attribute-name="SecurityManager">
    jboss.mq:service=SecurityManager
  </depends>

  <!-- ... -->

  <!-- Declare a dependency on the
      "jboss.mq:service=CacheManager" without
      any binding of the name to an attribute-->
  <depends>jboss.mq:service=CacheManager</depends>
</mbean>

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



```

list-element>
    <depends-list-element>jms.topic:service=Topic,name=B</depends-
list-element>
    <depends-list-element>jms.topic:service=Topic,name=C</depends-
list-element>
    </depends-list>
</mbean>

```

Another difference between the **depends** and **depends-list** elements is that the value of the **depends** element may be a complete MBean service configuration rather than just the **ObjectName** of the service. [Example 5.13](#), “An example of using the **depends** element to specify the complete configuration of a depended on service.” shows an example from the **hsqldb-service.xml** descriptor. In this listing the **org.jboss.resource.connectionmanager.RARDeployment** service configuration is defined using a nested **mbean** element as the **depends** element value. This indicates that the **org.jboss.resource.connectionmanager.LocalTxConnectionManager** MBean depends on this service. The **jboss.jca:service=LocalTxDS,name=hsqldbDS** **ObjectName** will be bound to the **ManagedConnectionFactoryName** attribute of the **LocalTxConnectionManager** class.

Example 5.13. An example of using the **depends element to specify the complete configuration of a depended on service.**

```

<mbean
code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
name="jboss.jca:service=LocalTxCM,name=hsqldbDS">
  <depends optional-attribute-name="ManagedConnectionFactoryName">
    <!-- embedded mbean -->
    <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
name="jboss.jca:service=LocalTxDS,name=hsqldbDS">
      <attribute name="JndiName">DefaultDS</attribute>
      <attribute name="ManagedConnectionFactoryProperties">
        <properties>
          <config-property name="ConnectionURL"
type="java.lang.String">
            jdbc:hsqldb:hsq://localhost:1476
          </config-property>
          <config-property name="DriverClass"
type="java.lang.String">
            org.hsqldb.jdbcDriver
          </config-property>
          <config-property name="UserName"
type="java.lang.String">
            sa
          </config-property>
          <config-property name="Password"
type="java.lang.String"/>
        </properties>
      </attribute>
    <!-- ... -->
    </mbean>
  </depends>
  <!-- ... -->
</mbean>

```

5.4.2.5. Identifying Unsatisfied Dependencies

The **ServiceController** MBean supports two operations that can help determine which MBeans are not running due to unsatisfied dependencies. The first operation is **listIncompletelyDeployed**. This returns a **java.util.List** of **org.jboss.system.ServiceContext** objects for the MBean services that are not in the **RUNNING** state.

The second operation is **listWaitingMBeans**. This operation returns a **java.util.List** of the JMX **ObjectName**s of MBean services that cannot be deployed because the class specified by the code attribute is not available.

5.4.2.6. Hot Deployment of Components, the URLDeploymentScanner

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

- **URLs**: A comma separated list of URL strings for the locations that should be watched for changes. Strings that do not correspond to valid URLs are treated as file paths. Relative file paths are resolved against the server home URL, for example, **JBOSS_DIST/server/production** for the production config file set. If a URL represents a file then the file is deployed and watched for subsequent updates or removal. If a URL ends in **/** to represent a directory, then the contents of the directory are treated as a collection of deployables and scanned for content that are to be watched for updates or removal. The requirement that a URL end in a **/** to identify a directory follows the RFC2518 convention and allows discrimination between collections and directories that are simply unpacked archives.

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

Example URLs include:

- **deploy/** scans **\${jboss.server.url}/deploy/**, which is local or remote depending on the URL used to boot the server
- **\${jboss.server.home.dir}/deploy/** scans **\${jboss.server.home.dir}/deploy**, which is always local
- **file:/var/opt/myapp.ear** deploys **myapp.ear** from a local location
- **file:/var/opt/apps/** scans the specified directory
- **http://www.test.com/netboot/myapp.ear** deploys **myapp.ear** from a remote location
- **http://www.test.com/netboot/apps/** scans the specified remote location using WebDAV. This will only work if the remote http server supports the WebDAV PROPFIND command.
- **ScanPeriod**: The time in milliseconds between runs of the scanner thread. The default is 5000 (5 seconds).
- **URLComparator**: The class name of a **java.util.Comparator** implementation used to specify a deployment ordering for deployments found in a scanned directory. The implementation must be able to compare two **java.net.URL** objects passed to its compare method. The default setting is the **org.jboss.deployment.DeploymentSorter** class which

orders based on the deployment URL suffix. The ordering of suffixes is: **deployer**, **deployer.xml**, **sar**, **rar**, **ds.xml**, **service.xml**, **har**, **jar**, **war**, **wsr**, **ear**, **zip**, **bsh**, **last**.

An alternate implementation is the

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

- **Filter:** The class name of a **java.io.FileFilter** implementation that is used to filter the contents of scanned directories. Any file not accepted by this filter will not be deployed. The default is **org.jboss.deployment.scanner.DeploymentFilter** which is an implementation that rejects the following patterns:

```
"#*", "%*", " ", "*", ".*", "$*", "*#", "*$", "*%", "*.BAK", "*.old", "*.orig", "*.rej", "*.bak",
"*.sh", "*", "v", "*~", ".make.state", ".nse_depinfo", "CVS", "CVS.admin", "RCS",
"RCSLOG", "SCCS", "TAGS", "core", "tags"
```

- **RecursiveSearch:** This property indicates whether or not deploy subdirectories are seen to be holding deployable content. If this is false, deploy subdirectories that do not contain a dot (.) in their name are seen to be unpackaged JARs with nested subdeployments. If true, then deploy subdirectories are just groupings of deployable content. The difference between the two views shows is related to the depth first deployment model JBoss supports. The false setting which treats directories as unpackaged JARs with nested content triggers the deployment of the nested content as soon as the JAR directory is deployed. The true setting simply ignores the directory and adds its content to the list of deployable packages and calculates the order based on the previous filter logic. The default is true.
- **Deployer:** The JMX **ObjectName** string of the MBean that implements the **org.jboss.deployment.Deployer** interface operations. The default setting is to use the **MainDeployer** created by the bootstrap startup process.

5.4.3. Writing JBoss MBean Services

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

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

Which approach you choose depends on whether or not you want your code to be coupled to JBoss specific code. If you don't, then you would use the first approach. If you don't care about dependencies on JBoss classes, the simplest approach is to have your MBean interface extend from

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

5.4.3.1. A Standard 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.

Version one of the classes, shown in [Example 5.14, “JNDIMapMBean interface and implementation based on the service interface method pattern”](#), is based on the service interface method pattern. This version of the interface declares the **start** and **stop** methods needed to start up correctly without using any JBoss-specific classes.

Example 5.14. JNDIMapMBean interface and implementation based on the service interface method pattern

```
package org.jboss.book.jmx.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.book.jmx.ex1;

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

public class JNDIMap implements JNDIMapMBean
{
    private String jndiName;
    private HashMap contextMap = new HashMap();
    private boolean started;

    public String getJndiName()
    {
```

```

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

    public void start() throws Exception
    {
        started = true;
        rebind();
    }

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

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

    private void unbind(String jndiName)
    {
        try {
            InitialContext rootCtx = new InitialContext();
            rootCtx.unbind(jndiName);
            NonSerializableFactory.unbind(jndiName);
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}

```

Version two of the classes, shown in [Example 5.14, “JNDIMapMBean interface and implementation based on the service interface method pattern”](#), use the JBoss **ServiceMBean** interface and **ServiceMBeanSupport** class. In this version, the implementation class extends the **ServiceMBeanSupport** class and overrides the **startService** and **stopService** methods.

JNDIMapMBean also implements the abstract **getName** method to return a descriptive name for the MBean. The **JNDIMapMBean** interface extends the **org.jboss.system.ServiceMBean** interface and only declares the setter and getter methods for the **JndiName** attribute because it inherits the service life cycle methods from **ServiceMBean**. This is the third approach mentioned at the start of the [Section 5.4.2, “JBoss MBean Services”](#).

Example 5.15. JNDIMap MBean interface and implementation based on the ServiceMBean interface and ServiceMBeanSupport class

```
package org.jboss.book.jmx.ex2;

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

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

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

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

    public String getJndiName()
    {
        return jndiName;
    }

    public void setJndiName(String jndiName)
        throws NamingException
    {
        String oldName = this.jndiName;
        this.jndiName = jndiName;
        if (super.getState() == STARTED) {
            unbind(oldName);
            try {
                rebind();
            } catch (Exception e) {
                NamingException ne = new NamingException("Failed to
update jndiName");
                ne.setRootCause(e);
                throw ne;
            }
        }
    }
}
```

```

    }

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

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

    private void rebind() throws NamingException
    {
        InitialContext rootCtx = new InitialContext();
        Name fullName = rootCtx.getNameParser("").parse(jndiName);
        log.info("fullName="+fullName);
        NonSerializableFactory.rebind(fullName, contextMap, true);
    }

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

```

The source code for these MBeans along with the service descriptors is located in the **examples/src/main/org/jboss/book/jmx/{ex1,ex2}** directories.

The jboss-service.xml descriptor for the first version is shown below.

```

<!-- The SAR META-INF/jboss-service.xml descriptor -->
<server>
    <mbean code="org.jboss.book.jmx.ex1.JNDIMap"
          name="j2eechap2.ex1:service=JNDIMap">
        <attribute name="JndiName">inmemory/maps/MapTest</attribute>
        <depends>jboss:service=Naming</depends>
    </mbean>
</server>

```

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. The corresponding client code is shown below.

```

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

```

5.4.3.2. XMBean Examples

In this section we will develop a variation of the **JNDIMap** MBean introduced in the preceding section that exposes its management metadata using the JBoss XMBean framework. Our core managed component will be exactly the same core code from the **JNDIMap** class, but it will not implement any specific management related interface. We will illustrate the following capabilities not possible with a standard MBean:

- The ability to add rich descriptions to attribute and operations
- The ability to expose notification information
- The ability to add persistence of attributes
- The ability to add custom interceptors for security and remote access through a typed interface

5.4.3.2.1. Version 1, The Annotated JNDIMap XMBean

Let's start with a simple XMBean variation of the standard MBean version of the JNDIMap that adds the descriptive information about the attributes and operations and their arguments. The following listing shows the **jboss-service.xml** descriptor and the **jndimap-xmbean1.xml** XMBean descriptor. The source can be found in the **src/main/org/jboss/book/jmx/xmbean** directory of the book examples.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE server PUBLIC
    "-//JBoss//DTD MBean Service 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-
service_3_2.dtd">
<server>
    <mbean code="org.jboss.book.jmx.xmbean.JNDIMap"
        name="j2eechap2.xmbean:service=JNDIMap"
        xmbean-dd="META-INF/jndimap-xmbean.xml">
        <attribute name="JndiName">inmemory/maps/MapTest</attribute>
        <depends>jboss:service=Naming</depends>
    </mbean>
</server>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
    "-//JBoss//DTD JBOSS XMBean 1.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">
<mbean>
    <description>The JNDIMap XMBean Example Version 1</description>
    <descriptors>
        <persistence persistPolicy="Never" persistPeriod="10"
            persistLocation="data/JNDIMap.data" persistName="JNDIMap"/>
        <currencyTimeLimit value="10"/>
        <state-action-on-update value="keep-running"/>
    </descriptors>
    <class>org.jboss.test.jmx.xmbean.JNDIMap</class>
    <constructor>
        <description>The default constructor</description>
        <name>JNDIMap</name>
    </constructor>
```



```

    <!-- Attributes -->
    <attribute access="read-write" getMethod="getJndiName"
setMethod="setJndiName">
        <description>
            The location in JNDI where the Map we manage will be bound
        </description>
        <name>JndiName</name>
        <type>java.lang.String</type>
        <descriptors>
            <default value="inmemory/maps/MapTest"/>
        </descriptors>
    </attribute>
    <attribute access="read-write" getMethod="getInitialValues"
        setMethod="setInitialValues">
        <description>The array of initial values that will be placed into
the
            map associated with the service. The array is a collection of
            key,value pairs with elements[0,2,4,...2n] being the keys and
            elements [1,3,5,...,2n+1] the associated values. The
of the
            "[Ljava.lang.String;" type signature is the VM representation

            java.lang.String[] type. </description>
        <name>InitialValues</name>
        <type>[Ljava.lang.String;</type>
        <descriptors>
            <default value="key0,value0"/>
        </descriptors>
    </attribute>
    <!-- Operations -->
    <operation>
        <description>The start lifecycle operation</description>
        <name>start</name>
    </operation>
    <operation>
        <description>The stop lifecycle operation</description>
        <name>stop</name>
    </operation>
    <operation impact="ACTION">
        <description>Put a value into the map</description>
        <name>put</name>
        <parameter>
            <description>The key the value will be store
under</description>
            <name>key</name>
            <type>java.lang.Object</type>
        </parameter>
        <parameter>
            <description>The value to place into the map</description>
            <name>value</name>
            <type>java.lang.Object</type>
        </parameter>
    </operation>
    <operation impact="INFO">
        <description>Get a value from the map</description>
        <name>get</name>
        <parameter>

```

```

        <description>The key to lookup in the map</description>
        <name>get</name>
        <type>java.lang.Object</type>
    </parameter>
    <return-type>java.lang.Object</return-type>
</operation>
<!-- Notifications -->
<notification>
    <description>The notification sent whenever a value is get into
the map
        managed by the service</description>
    <name>javax.management.Notification</name>
    <notification-
type>org.jboss.book.jmx.xmbean.JNDIMap.get</notification-type>
    </notification>
    <notification>
    <description>The notification sent whenever a value is put into
the map
        managed by the service</description>
    <name>javax.management.Notification</name>
    <notification-
type>org.jboss.book.jmx.xmbean.JNDIMap.put</notification-type>
    </notification>
</mbean>

```

You can build, deploy and test the XMBean as follows:

```

[examples]$ ant -Dchap=jmx -Dex=xmbean1 run-example
...
run-examplexmbean1:
    [java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
    [java] JNDIMap Operations:
    [java] + void start()
    [java] + void stop()
    [java] + void put(java.lang.Object
chap2.xmbean:service=JNDIMap,java.lang.Object
        chap2.xmbean:service=JNDIMap)
    [java] + java.lang.Object get(java.lang.Object
chap2.xmbean:service=JNDIMap)
    [java] name=chap2.xmbean:service=JNDIMap
    [java] listener=org.jboss.book.jmx.xmbean.TestXMBean1$Listener@f38cf0
    [java] key=key0, value=value0
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
        service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.put]
[message=]
    [java] JNDIMap.put(key1, value1) successful
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
        service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.get]
[message=]
    [java] JNDIMap.get(key0): null
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
        service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.get]
[message=]

```

```

[java] JNDIMap.get(key1): value1
[java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
    service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.put]
[message=]
[java] handleNotification, event:
javax.management.AttributeChangeNotification[source
    =chap2.xmbean:service=JNDIMap][type=jmx.attribute.change]
[message=InitialValues
    changed from javax.management.Attribute@82a72a to
    javax.management.Attribute@acdb96]

```

The functionality is largely the same as the Standard MBean with the notable exception of the JMX notifications. A Standard MBean has no way of declaring that it will emit notifications. An XMBEAN may declare the notifications it emits using notification elements as is shown in the version 1 descriptor. We see the notifications from the get and put operations on the test client console output. Note that there is also an **jmx.attribute.change notification** emitted when the **InitialValues** attribute was changed. This is because the **ModelMBean** interface extends the **ModelMBeanNotificationBroadcaster** which supports **AttributeChangeNotificationListeners**.

The other major difference between the Standard and XMBEAN versions of JNDIMap is the descriptive metadata. Look at the **chap2.xmbean:service=JNDIMap** in the JMX Console, and you will see the attributes section as shown in [Figure 5.18, “The Version 1 JNDIMapXMBEAN jmx-console view”](#).

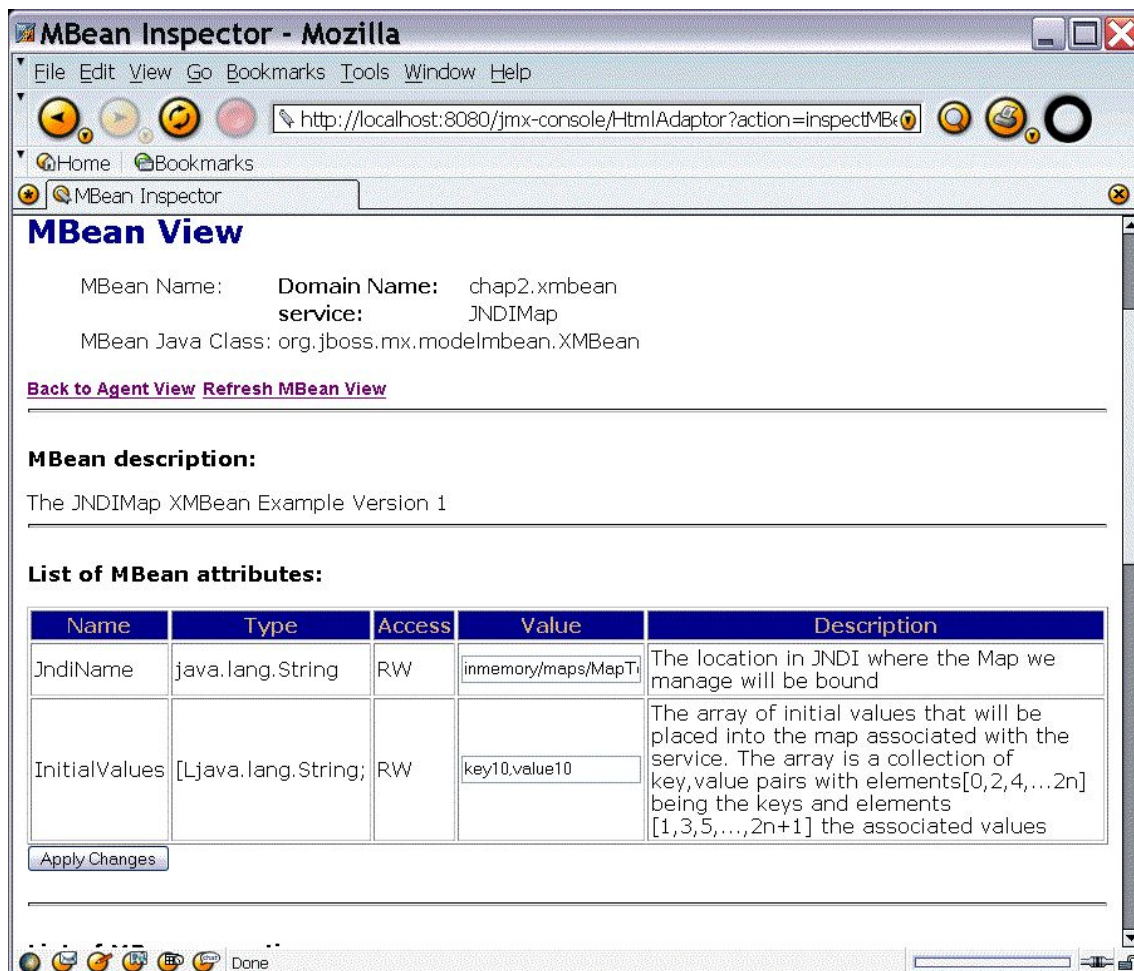


Figure 5.18. The Version 1 JNDIMapXMBEAN jmx-console view

Notice that the JMX Console now displays the full attribute description as specified in the XMBEAN

descriptor rather than **MBean Attribute** text seen in standard MBean implementations. Scroll down to the operations and you will also see that these now also have nice descriptions of their function and parameters.

5.4.3.2.2. Version 2, Adding Persistence to the JNDIMap XMBEAN

In version 2 of the XMBEAN we add support for persistence of the XMBEAN attributes. The updated XMBEAN deployment descriptor is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
    "-//JBoss//DTD JBOSS XMBean 1.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">
<mbean>
    <description>The JNDIMap XMBEAN Example Version 2</description>
    <descriptors>
        <persistence persistPolicy="OnUpdate" persistPeriod="10"
            persistLocation="${jboss.server.data.dir}"
            persistName="JNDIMap.ser"/>
        <currencyTimeLimit value="10"/>
        <state-action-on-update value="keep-running"/>
        <persistence-manager
            value="org.jboss.mx.persistence.ObjectStreamPersistenceManager"/>
    </descriptors> <class>org.jboss.test.jmx.xmbean.JNDIMap</class>
    <constructor>
        <description>The default constructor</description>
        <name>JNDIMap</name>
    </constructor>
    <!-- Attributes -->
    <attribute access="read-write" getMethod="getJndiName"
        setMethod="setJndiName">
        <description>
            The location in JNDI where the Map we manage will be bound
        </description>
        <name>JndiName</name>
        <type>java.lang.String</type>
        <descriptors>
            <default value="inmemory/maps/MapTest"/>
        </descriptors>
    </attribute>
    <attribute access="read-write" getMethod="getInitialValues"
        setMethod="setInitialValues">
        <description>The array of initial values that will be placed into
the
            map associated with the service. The array is a collection of
            key,value pairs with elements[0,2,4,...2n] being the keys and
            elements [1,3,5,...,2n+1] the associated values</description>
        <name>InitialValues</name>
        <type>[Ljava.lang.String;</type>
        <descriptors>
            <default value="key0,value0"/>
        </descriptors>
    </attribute>
    <!-- Operations -->
    <operation>
        <description>The start lifecycle operation</description>
```

```

        <name>start</name>
    </operation>
    <operation>
        <description>The stop lifecycle operation</description>
        <name>stop</name>
    </operation>
    <operation impact="ACTION">
        <description>Put a value into the nap</description>
        <name>put</name>
        <parameter>
            <description>The key the value will be store
under</description>
            <name>key</name>
            <type>java.lang.Object</type>
        </parameter>
        <parameter>
            <description>The value to place into the map</description>
            <name>value</name>
            <type>java.lang.Object</type>
        </parameter>
    </operation>
    <operation impact="INFO">
        <description>Get a value from the map</description>
        <name>get</name>
        <parameter>
            <description>The key to lookup in the map</description>
            <name>get</name>
            <type>java.lang.Object</type>
        </parameter>
        <return-type>java.lang.Object</return-type>
    </operation>
    <!-- Notifications -->
    <notification>
        <description>The notification sent whenever a value is get into
the map
            managed by the service</description>
        <name>javax.management.Notification</name>
        <notification-
type>org.jboss.book.jmx.xmlbean.JNDIMap.get</notification-type>
    </notification>
    <notification>
        <description>The notification sent whenever a value is put into
the map
            managed by the service</description>
        <name>javax.management.Notification</name>
        <notification-
type>org.jboss.book.jmx.xmlbean.JNDIMap.put</notification-type>
    </notification>
</mbean>

```

Build, deploy and test the version 2 XMBean as follows:

```

[examples]$ ant -Dchap=jmx -Dex=xmbean2 -Djboss.deploy.conf=default run-
example
...
run-examplexmbean2:

```

```

[java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
[java] JNDIMap Operations:
[java] + void start()
[java] + void stop()
[java] + void put(java.lang.Object
chap2.xmbean:service=JNDIMap,java.lang.Object cha
p2.xmbean:service=JNDIMap)
[java] + java.lang.Object get(java.lang.Object
chap2.xmbean:service=JNDIMap)
[java] + java.lang.String getJndiName()
[java] + void setJndiName(java.lang.String
chap2.xmbean:service=JNDIMap)
[java] + [Ljava.lang.String; getInitialValues()
[java] + void setInitialValues([Ljava.lang.String;
chap2.xmbean:service=JNDIMap)
[java] handleNotification, event: null
[java] key=key10, value=value10
[java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=7,ti
meStamp=10986326
93716,message=null,userData=null]
[java] JNDIMap.put(key1, value1) successful
[java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=8,ti
meStamp=10986326
93857,message=null,userData=null]
[java] JNDIMap.get(key0): null
[java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=9,ti
meStamp=10986326
93896,message=null,userData=null]
[java] JNDIMap.get(key1): value1
[java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=10,t
imeStamp=1098632
693925,message=null,userData=null]

```

There is nothing manifestly different about this version of the XMBean at this point because we have done nothing to test that changes to attribute value are actually persisted. Perform this test by running example `xmbean2a` several times:

```

[examples] ant -Dchap=jmx -Dex=xmbean2a run-example
...
[java] InitialValues.length=2
[java] key=key10, value=value10

```

```

[examples] ant -Dchap=jmx -Dex=xmbean2a run-example
...
[java] InitialValues.length=4
[java] key=key10, value=value10
[java] key=key2, value=value2

```

```
[examples] ant -Dchap=jmx -Dex=xmbean2a run-example
...
[java] InitialValues.length=6
[java] key=key10, value=value10
[java] key=key2, value=value2
[java] key=key3, value=value3
```

The **org.jboss.book.jmx.xmbean.TestXMBeanRestart** used in this example obtains the current **InitialValues** attribute setting, and then adds another key/value pair to it. The client code is shown below.

```
package org.jboss.book.jmx.xmbean;

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

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

/**
 * A client that demonstrates the persistence of the xmbean
 * attributes. Every time it run it looks up the InitialValues
 * attribute, prints it out and then adds a new key/value to the
 * list.
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class TestXMBeanRestart
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        InitialContext ic = new InitialContext();
        RMIAdaptor server = (RMIAdaptor) ic.lookup("jmx/rmi/RMIAdaptor");

        // Get the InitialValues attribute
        ObjectName name = new ObjectName("j2eechap2.xmbean:service=JNDIMap");
        String[] initialValues = (String[])
            server.getAttribute(name, "InitialValues");
        System.out.println("InitialValues.length="+initialValues.length);
        int length = initialValues.length;
        for (int n = 0; n < length; n += 2) {
            String key = initialValues[n];
            String value = initialValues[n+1];

            System.out.println("key="+key+", value="+value);
        }
        // Add a new key/value pair
        String[] newInitialValues = new String[length+2];
        System.arraycopy(initialValues, 0, newInitialValues,
            0, length);
        newInitialValues[length] = "key"+(length/2+1);
```

```

    newInitialValues[length+1] = "value"+(length/2+1);

    Attribute ivalues = new
        Attribute("InitialValues", newInitialValues);
    server.setAttribute(name, ivalues);
}
}

```

At this point you may even shutdown the JBoss server, restart it and then rerun the initial example to see if the changes are persisted across server restarts:

```

[examples]$ ant -Dchap=jmx -Dex=xmbean2 run-example
...

run-examplexmbean2:
    [java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
    [java] JNDIMap Operations:
    [java] + void start()
    [java] + void stop()
    [java] + void put(java.lang.Object
chap2.xmbean:service=JNDIMap,java.lang.Object cha
p2.xmbean:service=JNDIMap)
    [java] + java.lang.Object get(java.lang.Object
chap2.xmbean:service=JNDIMap)
    [java] + java.lang.String getJndiName()
    [java] + void setJndiName(java.lang.String
chap2.xmbean:service=JNDIMap)
    [java] + [Ljava.lang.String; getInitialValues()
    [java] + void setInitialValues([Ljava.lang.String;
chap2.xmbean:service=JNDIMap)
    [java] handleNotification, event: null
    [java] key=key10, value=value10
    [java] key=key2, value=value2
    [java] key=key3, value=value3
    [java] key=key4, value=value4
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=3
,timeStamp=10986
33664712,message=null,userData=null]
    [java] JNDIMap.put(key1, value1) successful
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.get,sequenceNumber=4
,timeStamp=10986
33664821,message=null,userData=null]
    [java] JNDIMap.get(key0): null
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.get,sequenceNumber=5
,timeStamp=10986
33664860,message=null,userData=null]
    [java] JNDIMap.get(key1): value1
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=6

```



```
,timestamp=10986
33664877,message=null,userData=null]
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=7
,timestamp=10986
33664895,message=null,userData=null]
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=8
,timestamp=10986
33664899,message=null,userData=null]
    [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=9
,timestamp=10986
33665614,message=null,userData=null]
```

You see that the last **InitialValues** attribute setting is in fact visible.

5.4.4. Deployment Ordering and Dependencies

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

When a component archive is deployed, its nested deployment units are processed in a depth first ordering. Structuring of components into an archive hierarchy is yet another way to manage deployment ordering. You will need to explicitly state your MBean dependencies if your packaging structure does not happen to resolve the dependencies. Let's consider an example component deployment that consists of an MBean that uses an EJB. Here is the structure of the example EAR.

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

The EAR contains a **jmx-ex3.jar** and **jmx-ex3.sar**. The **jmx-ex3.jar** is the EJB archive and the **jmx-ex3.sar** is the MBean service archive. We have implemented the service as a Dynamic MBean to provide an illustration of their use.

```
package org.jboss.book.jmx.ex3;
```

```
import java.lang.reflect.Method;
import javax.ejb.CreateException;
import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.AttributeNotFoundException;
import javax.management.DynamicMBean;
import javax.management.InvalidAttributeValueException;
import javax.management.JMRuntimeException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanInfo;
import javax.management.MBeanNotificationInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanException;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.ReflectionException;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.jboss.system.ServiceMBeanSupport;

/**
 * An example of a DynamicMBean that exposes select attributes and
 * operations of an EJB as an MBean.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class EjbMBeanAdaptor extends ServiceMBeanSupport
    implements DynamicMBean
{
    private String helloPrefix;
    private String ejbJndiName;
    private EchoLocalHome home;

    /** These are the mbean attributes we expose
     */
    private MBeanAttributeInfo[] attributes = {
        new MBeanAttributeInfo("HelloPrefix", "java.lang.String",
            "The prefix message to append to the
session echo reply",
                                true, // isReadable
                                true, // isWritable
                                false), // isIs
        new MBeanAttributeInfo("EjbJndiName", "java.lang.String",
            "The JNDI name of the session bean local
home",
                                true, // isReadable
                                true, // isWritable
                                false) // isIs
    };

    /**
     * These are the mbean operations we expose
     */
}
```

```

private MBeanOperationInfo[] operations;

/**
 * We override this method to setup our echo operation info. It
 * could also be done in a ctor.
 */
public ObjectName preRegister(MBeanServer server,
                              ObjectName name)
    throws Exception
{
    log.info("preRegister notification seen");

    operations = new MBeanOperationInfo[5];

    Class thisClass = getClass();
    Class[] parameterTypes = {String.class};
    Method echoMethod =
        thisClass.getMethod("echo", parameterTypes);
    String desc = "The echo op invokes the session bean echo method
and"
        + " returns its value prefixed with the helloPrefix attribute
value";
    operations[0] = new MBeanOperationInfo(desc, echoMethod);

    // Add the Service interface operations from our super class
    parameterTypes = new Class[0];
    Method createMethod =
        thisClass.getMethod("create", parameterTypes);
    operations[1] = new MBeanOperationInfo("The
        JBoss Service.create", createMethod);
    Method startMethod =
        thisClass.getMethod("start", parameterTypes);
    operations[2] = new MBeanOperationInfo("The
        JBoss Service.start", startMethod);
    Method stopMethod =
        thisClass.getMethod("stop", parameterTypes);
    operations[3] = new MBeanOperationInfo("The
        JBoss Service.stop", startMethod);
    Method destroyMethod =
        thisClass.getMethod("destroy", parameterTypes);
    operations[4] = new MBeanOperationInfo("The
        JBoss Service.destroy", startMethod);
    return name;
}

// --- Begin ServiceMBeanSupport overrides
protected void createService() throws Exception
{
    log.info("Notified of create state");
}

protected void startService() throws Exception
{
    log.info("Notified of start state");
    InitialContext ctx = new InitialContext();

```

```

        home = (EchoLocalHome) ctx.lookup(ejbJndiName);
    }

    protected void stopService()
    {
        log.info("Notified of stop state");
    }

    // --- End ServiceMBeanSupport overrides

    public String getHelloPrefix()
    {
        return helloPrefix;
    }

    public void setHelloPrefix(String helloPrefix)
    {
        this.helloPrefix = helloPrefix;
    }

    public String getEjbJndiName()
    {
        return ejbJndiName;
    }

    public void setEjbJndiName(String ejbJndiName)
    {
        this.ejbJndiName = ejbJndiName;
    }

    public String echo(String arg)
        throws CreateException, NamingException
    {
        log.debug("Lookup EchoLocalHome@"+ejbJndiName);
        EchoLocal bean = home.create();
        String echo = helloPrefix + bean.echo(arg);
        return echo;
    }

    // --- Begin DynamicMBean interface methods
    /**
     * Returns the management interface that describes this dynamic
     * resource. It is the responsibility of the implementation to
     * make sure the description is accurate.
     *
     * @return the management interface descriptor.
     */
    public MBeanInfo getMBeanInfo()
    {
        String classname = getClass().getName();
        String description = "This is an MBean that uses a session bean in
the"
            + " implementation of its echo operation.";
        MBeanInfo[] constructors = null;
        MBeanNotificationInfo[] notifications = null;
        MBeanInfo mbeanInfo = new MBeanInfo(classname,
                                            description, attributes,
                                            constructors, operations,

```

```

        notifications);
    // Log when this is called so we know when in the
    lifecycle this is used
    Throwable trace = new Throwable("getMBeanInfo trace");
    log.info("Don't panic, just a stack
        trace", trace);
    return mbeanInfo;
}

/**
 * Returns the value of the attribute with the name matching the
 * passed string.
 *
 * @param attribute the name of the attribute.
 * @return the value of the attribute.
 * @exception AttributeNotFoundException when there is no such
 * attribute.
 * @exception MBeanException wraps any error thrown by the
 * resource when
 * getting the attribute.
 * @exception ReflectionException wraps any error invoking the
 * resource.
 */
public Object getAttribute(String attribute)
    throws AttributeNotFoundException,
           MBeanException,
           ReflectionException
{
    Object value = null;
    if (attribute.equals("HelloPrefix")) {
        value = getHelloPrefix();
    } else if (attribute.equals("EjbJndiName")) {
        value = getEjbJndiName();
    } else {
        throw new AttributeNotFoundException("Unknown
            attribute(\"+attribute+\") requested");
    }
    return value;
}

/**
 * Returns the values of the attributes with names matching the
 * passed string array.
 *
 * @param attributes the names of the attribute.
 * @return an {@link AttributeList AttributeList} of name
 * and value pairs.
 */
public AttributeList getAttributes(String[] attributes)
{
    AttributeList values = new AttributeList();
    for (int a = 0; a < attributes.length; a++) {
        String name = attributes[a];
        try {
            Object value = getAttribute(name);
            Attribute attr = new Attribute(name, value);

```

```

        values.add(attr);
    } catch(Exception e) {
        log.error("Failed to find attribute: "+name, e);
    }
}
return values;
}

/**
 * Sets the value of an attribute. The attribute and new value
 * are passed in the name value pair {@link Attribute
 * Attribute}.
 *
 * @see javax.management.Attribute
 *
 * @param attribute the name and new value of the attribute.
 * @exception AttributeNotFoundException when there is no such
 * attribute.
 * @exception InvalidAttributeValueException when the new value
 * cannot be converted to the type of the attribute.
 * @exception MBeanException wraps any error thrown by the
 * resource when setting the new value.
 * @exception ReflectionException wraps any error invoking the
 * resource.
 */
public void setAttribute(Attribute attribute)
    throws AttributeNotFoundException,
        InvalidAttributeValueException,
        MBeanException,
        ReflectionException
{
    String name = attribute.getName();
    if (name.equals("HelloPrefix")) {
        String value = attribute.getValue().toString();
        setHelloPrefix(value);
    } else if(name.equals("EjbJndiName")) {
        String value = attribute.getValue().toString();
        setEjbJndiName(value);
    } else {
        throw new AttributeNotFoundException("Unknown
attribute("+name+") requested");
    }
}

/**
 * Sets the values of the attributes passed as an
 * {@link AttributeList AttributeList} of name and new
 * value pairs.
 *
 * @param attributes the name an new value pairs.
 * @return an {@link AttributeList AttributeList} of name and
 * value pairs that were actually set.
 */
public AttributeList setAttributes(AttributeList attributes)
{
    AttributeList setAttributes = new AttributeList();

```

```

        for(int a = 0; a < attributes.size(); a++) {
            Attribute attr = (Attribute) attributes.get(a);
            try {
                setAttribute(attr);
                setAttributes.add(attr);
            } catch(Exception ignore) {
            }
        }
        return setAttributes;
    }

    /**
     * Invokes a resource operation.
     *
     * @param actionName the name of the operation to perform.
     * @param params the parameters to pass to the operation.
     * @param signature the signartures of the parameters.
     * @return the result of the operation.
     * @exception MBeanException wraps any error thrown by the
     * resource when performing the operation.
     * @exception ReflectionException wraps any error invoking the
     * resource.
     */
    public Object invoke(String actionName, Object[] params,
                        String[] signature)
        throws MBeanException,
               ReflectionException
    {
        Object rtnValue = null;
        log.debug("Begin invoke, actionName="+actionName);
        try {
            if (actionName.equals("echo")) {
                String arg = (String) params[0];
                rtnValue = echo(arg);
                log.debug("Result: "+rtnValue);
            } else if (actionName.equals("create")) {
                super.create();
            } else if (actionName.equals("start")) {
                super.start();
            } else if (actionName.equals("stop")) {
                super.stop();
            } else if (actionName.equals("destroy")) {
                super.destroy();
            } else {
                throw new JMRuntimeException("Invalid state,
                    don't know about op="+actionName);
            }
        } catch(Exception e) {
            throw new ReflectionException(e, "echo failed");
        }

        log.debug("End invoke, actionName="+actionName);
        return rtnValue;
    }

```

```
// --- End DynamicMBean interface methods
```

```
}
```

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

There are two points to this example. First, demonstrate how an MBean can depend on an EJB for some of its functionality and second, how to create MBeans with dynamic management interfaces. If we were to write a standard MBean with a static interface for this example it would look like the following.

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

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

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

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

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

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

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

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

The **jboss-service.xml** descriptor for the MBean is given below. The dependency on the EJB container MBean is highlighted in bold. The format of the EJB container MBean ObjectName is: **"jboss.j2ee:service=EJB,jndiName=" + <home-jndi-name>** where the **<home-jndi-name>** is the EJB home interface JNDI name.

```
<server>
  <mbean code="org.jboss.book.jmx.ex3.EjbMBeanAdaptor"
    name="jboss.book:service=EjbMBeanAdaptor">
    <attribute name="HelloPrefix">AdaptorPrefix</attribute>
    <attribute
name="EjbJndiName">local/j2ee_chap2.EchoBean</attribute>

  <depends>jboss.j2ee:service=EJB,jndiName=local/j2ee_chap2.EchoBean</depend
s>
    </mbean>
</server>
```

Deploy the example ear by running:

```
[examples]$ ant -Dchap=jmx -Dex=3 run-example
```

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

```
14:57:12,906 INFO [EARDeployer] Init J2EE application:
file:/private/tmp/jboss-eap-4.3/jboss-as/server/
production/deploy/j2ee_chap2-ex3.ear
14:57:13,044 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,088 INFO [EjbMBeanAdaptor] preRegister notification seen
14:57:13,093 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
```

```
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,117 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,140 WARN  [EjbMBeanAdaptor] Unexcepected error accessing MBeanInfo
for null
java.lang.NullPointerException
    at
org.jboss.system.ServiceMBeanSupport.postRegister(ServiceMBeanSupport.java
:418)
...
14:57:13,203 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,232 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,420 INFO  [EjbModule] Deploying Chap2EchoInfoBean
14:57:13,443 INFO  [EjbModule] Deploying chap2.EchoBean
14:57:13,488 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,542 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,558 INFO  [EjbMBeanAdaptor] Begin invoke, actionName=create
14:57:13,560 INFO  [EjbMBeanAdaptor] Notified of create state
14:57:13,562 INFO  [EjbMBeanAdaptor] End invoke, actionName=create
14:57:13,604 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
...
14:57:13,621 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:1
53)
```

```

14:57:13,641 INFO [EjbMBeanAdaptor] Begin invoke, actionName=getState
14:57:13,942 INFO [EjbMBeanAdaptor] Begin invoke, actionName=start
14:57:13,944 INFO [EjbMBeanAdaptor] Notified of start state
14:57:13,951 INFO [EjbMBeanAdaptor] Testing Echo
14:57:13,983 INFO [EchoBean] echo, info=echo info, arg=, arg=startService
14:57:13,986 INFO [EjbMBeanAdaptor] echo(startService) = startService
14:57:13,988 INFO [EjbMBeanAdaptor] End invoke, actionName=start
14:57:13,991 INFO [EJBDeployer] Deployed: file:/tmp/jboss-eap-4.3/jboss-
as/server/production/tmp/deploy
/tmp60550jmx-ex3.ear-contents/jmx-ex3.jar
14:57:14,075 INFO [EARDeployer] Started J2EE application: ...

```

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

Now, let's invoke the echo method using the JMX console web application. Go to the JMX Console (<http://localhost:8080/jmx-console>) and find the *service=EjbMBeanAdaptor* in the *jboss.book* domain. Click on the link and scroll down to the *echo* operation section. The view should be like that shown in Figure 5.19, “The EjbMBeanAdaptor MBean operations JMX console view”.

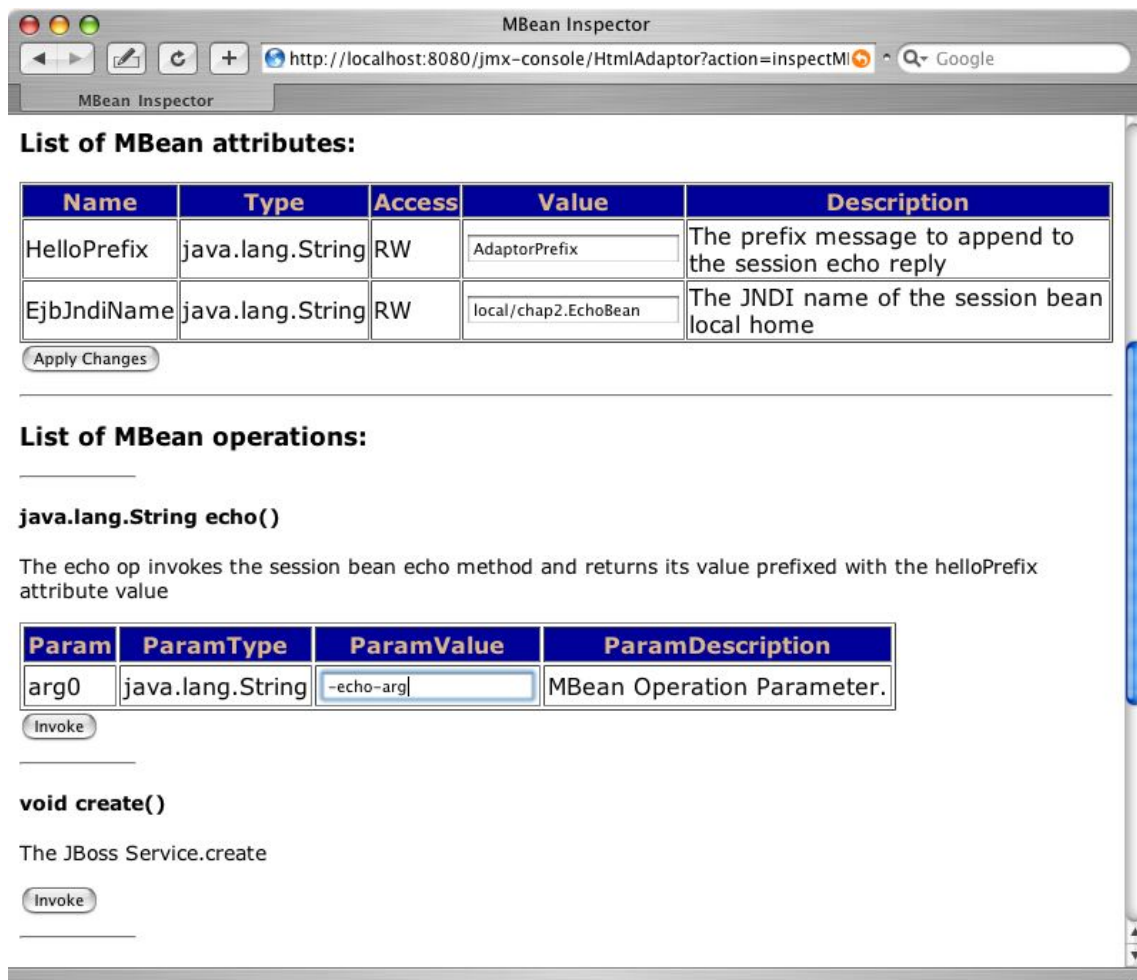


Figure 5.19. The EjbMBeanAdaptor MBean operations JMX console view

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

```
10:51:48,671 INFO [EjbMBeanAdaptor] Begin invoke, actionName=echo
10:51:48,671 INFO [EjbMBeanAdaptor] Lookup
EchoLocalHome@local/j2ee_chap2.EchoBean
10:51:48,687 INFO [EchoBean] echo, info=echo info, arg=, arg=-echo-arg
10:51:48,687 INFO [EjbMBeanAdaptor] Result: AdaptorPrefix-echo-arg
10:51:48,687 INFO [EjbMBeanAdaptor] End invoke, actionName=echo
```

5.5. JBOSS DEPLOYER ARCHITECTURE

JBoss has an extensible deployment architecture that allows one to incorporate components into the bare JBoss JMX microkernel. The **MainDeployer** is the deployment entry point. Requests to deploy a component are sent to the **MainDeployer** and it determines if there is a subdeployer capable of handling the deployment, and if there is, it delegates the deployment to the subdeployer. We saw an example of this when we looked at how the **MainDeployer** used the **SARDeployer** to deploy MBean services. Among the deployers provided with JBoss are:

- **AbstractWebDeployer**: This subdeployer handles web application archives (WARs). It accepts deployment archives and directories whose name ends with a **war** suffix. WARs must have a **WEB-INF/web.xml** descriptor and may have a **WEB-INF/jboss-web.xml** descriptor.
- **EARDeployer**: This subdeployer handles enterprise application archives (EARs). It accepts deployment archives and directories whose name ends with an **ear** suffix. EARs must have a **META-INF/application.xml** descriptor and may have a **META-INF/jboss-app.xml** descriptor.
- **EJBDeployer**: This subdeployer handles enterprise bean jars. It accepts deployment archives and directories whose name ends with a **jar** suffix. EJB jars must have a **META-INF/ejb-jar.xml** descriptor and may have a **META-INF/jboss.xml** descriptor.
- **JARDeployer**: This subdeployer handles library JAR archives. The only restriction it places on an archive is that it cannot contain a **WEB-INF** directory.
- **RARDeployer**: This subdeployer handles JCA resource archives (RARs). It accepts deployment archives and directories whose name ends with a **rar** suffix. RARs must have a **META-INF/ra.xml** descriptor.
- **SARDeployer**: This subdeployer handles JBoss MBean service archives (SARs). It accepts deployment archives and directories whose name ends with a **sar** suffix, as well as standalone XML files that end with **service.xml**. SARs that are jars must have a **META-INF/jboss-service.xml** descriptor.
- **XSLSubDeployer**: This subdeployer deploys arbitrary XML files. JBoss uses the XSLSubDeployer to deploy **ds.xml** files and transform them into **service.xml** files for the **SARDeployer**. However, it is not limited to just this task.
- **HARDeployer**: This subdeployer deploys hibernate archives (HARs). It accepts deployment archives and directories whose name ends with a **har** suffix. HARs must have a **META-INF/hibernate-service.xml** descriptor.
- **AspectDeployer**: This subdeployer deploys AOP archives. It accepts deployment archives and directories whose name ends with an **aop** suffix as well as **aop.xml** files. AOP archives must have a **META-INF/jboss-aop.xml** descriptor.
- **ClientDeployer**: This subdeployer deploys J2EE application clients. It accepts deployment

archives and directories whose name ends with a **jar** suffix. J2EE clients must have a **META-INF/application-client.xml** descriptor and may have a **META-INF/jboss-client.xml** descriptor.

- **BeanShellSubDeployer**: This subdeployer deploys bean shell scripts as MBeans. It accepts files whose name ends with a **bsh** suffix.

The **MainDeployer**, **JARDeployer** and **SARDeployer** are hard coded deployers in the JBoss server core. All other deployers are MBean services that register themselves as deployers with the **MainDeployer** using the **addDeployer(SubDeployer)** operation.

The **MainDeployer** communicates information about the component to be deployed the **SubDeployer** using a **DeploymentInfo** object. The **DeploymentInfo** object is a data structure that encapsulates the complete state of a deployable component.

When the **MainDeployer** receives a deployment request, it iterates through its registered subdeployers and invokes the **accepts(DeploymentInfo)** method on the subdeployer. The first subdeployer to return true is chosen. The **MainDeployer** will delegate the init, create, start, stop and destroy deployment life cycle operations to the subdeployer.

5.5.1. Deployers and ClassLoaders

Deployers are the mechanism by which components are brought into a JBoss server. Deployers are also the creators of the majority of UCL instances, and the primary creator is the **MainDeployer**. The **MainDeployer** creates the UCL for a deployment early on during its init method. The UCL is created by calling the **DeploymentInfo.createClassLoaders()** method. Only the topmost **DeploymentInfo** will actually create a UCL. All subdeployments will add their class paths to their parent **DeploymentInfo** UCL. Every deployment does have a standalone **URLClassLoader** that uses the deployment URL as its path. This is used to localize the loading of resources such as deployment descriptors. [Figure 5.20, “An illustration of the class loaders involved with an EAR deployment”](#) provides an illustration of the interaction between Deployers, **DeploymentInfos** and class loaders.

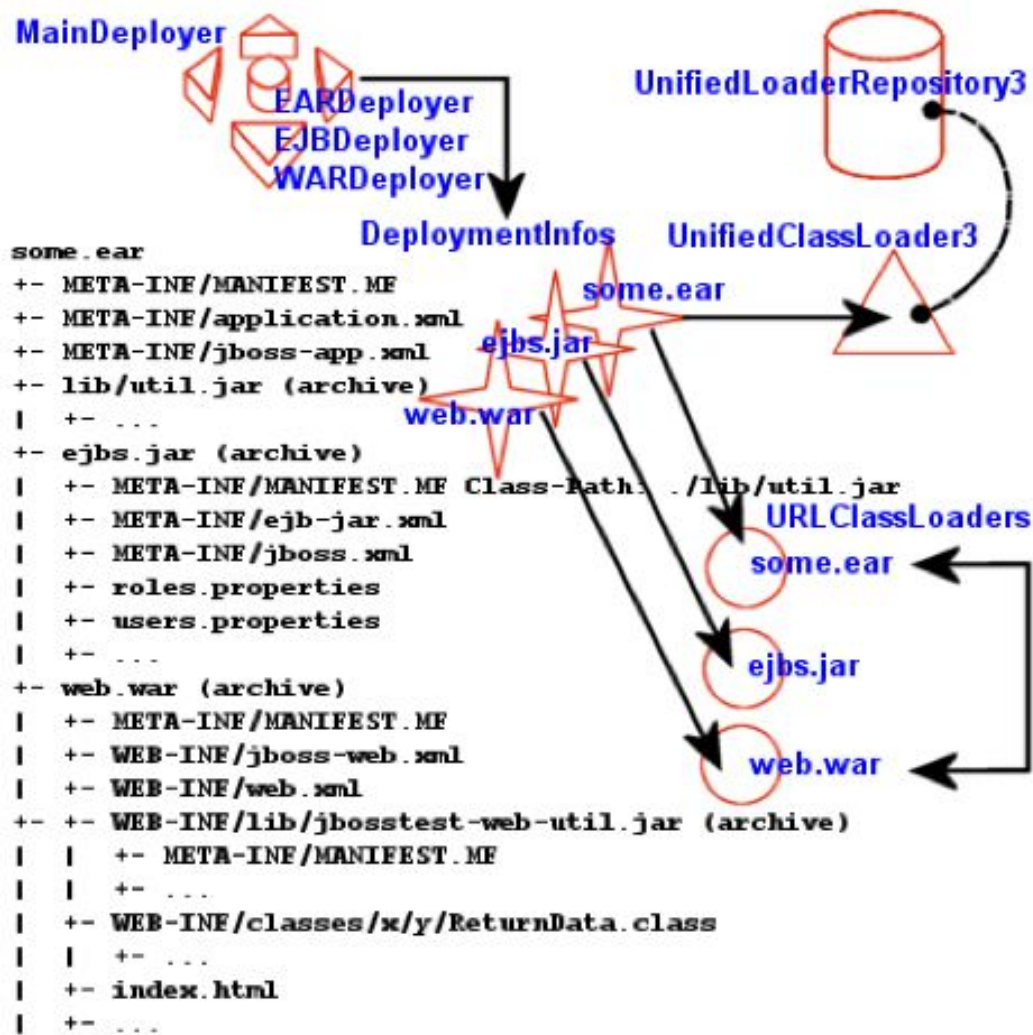


Figure 5.20. An illustration of the class loaders involved with an EAR deployment

The figure illustrates an EAR deployment with EJB and WAR subdeployments. The EJB deployment references the `lib/util.jar` utility jar via its manifest. The WAR includes classes in its `WEB-INF/classes` directory as well as the `WEB-INF/lib/jbosstest-web-util.jar`. Each deployment has a `DeploymentInfo` instance that has a `URLClassLoader` pointing to the deployment archive. The `DeploymentInfo` associated with `some.ear` is the only one to have a UCL created. The `ejbs.jar` and `web.war` `DeploymentInfos` add their deployment archive to the `some.ear` UCL classpath, and share this UCL as their deployment UCL. The `EJBDeployer` also adds any manifest jars to the EAR UCL.

The `WARDeployer` behaves differently than other deployers in that it only adds its WAR archive to the `DeploymentInfo` UCL classpath. The loading of classes from the WAR `WEB-INF/classes` and `WEB-INF/lib` locations is handled by the servlet container class loader. The servlet container class loaders delegate to the WAR `DeploymentInfo` UCL as their parent class loader, but the server container class loader is not part of the JBoss class loader repository. Therefore, classes inside of a WAR are not visible to other components. Classes that need to be shared between web application components and other components such as EJBs, and MBeans need to be loaded into the shared class loader repository either by including the classes into a SAR or EJB deployment, or by referencing a jar containing the shared classes through a manifest `Class-Path` entry. In the case of a SAR, the SAR classpath element in the service deployment serves the same purpose as a JAR manifest `Class-Path`.

5.6. REMOTE ACCESS TO SERVICES, DETACHED INVOKERS

In addition to the MBean services notion that allows for the ability to integrate arbitrary functionality, JBoss also has a detached invoker concept that allows MBean services to expose functional interfaces via arbitrary protocols for remote access by clients. The notion of a detached invoker is that remoting and the protocol by which a service is accessed is a functional aspect or service independent of the component. Thus, one can make a naming service available for use via RMI/JRMP, RMI/HTTP, RMI/SOAP, or any arbitrary custom transport.

Let's begin our discussion of the detached invoker architecture with an overview of the components involved. The main components in the detached invoker architecture are shown in [Figure 5.21](#), “The main components in the detached invoker architecture”.

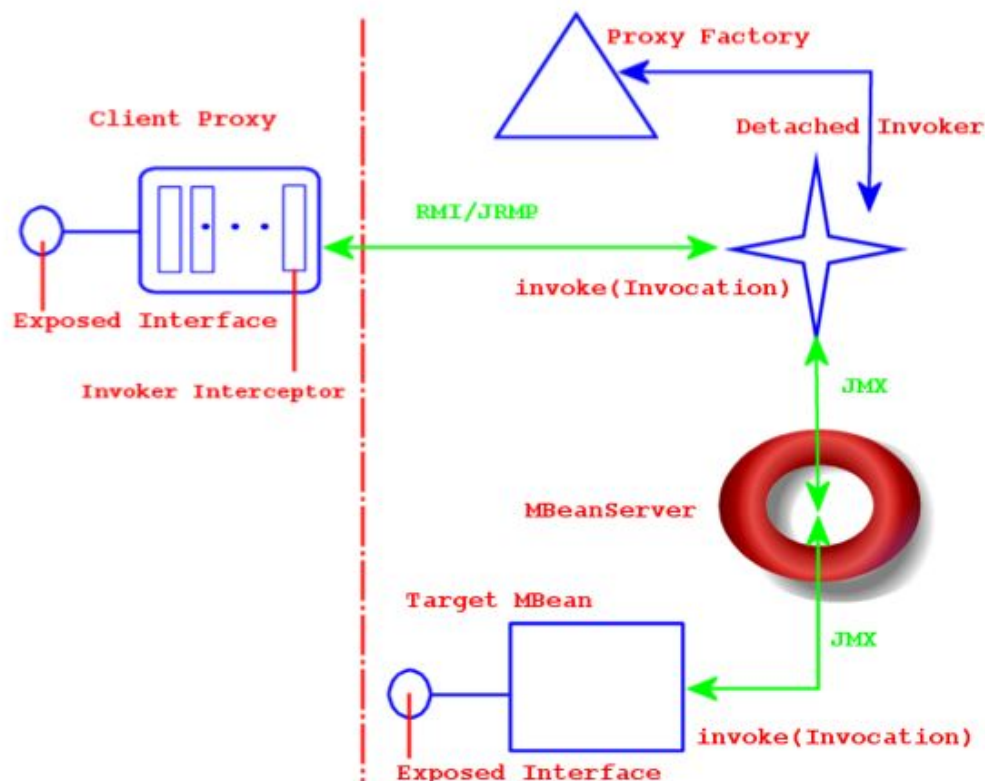


Figure 5.21. The main components in the detached invoker architecture

On the client side, there exists a client proxy which exposes the interface(s) of the MBean service. This is the same smart, compile-less dynamic proxy that we use for EJB home and remote interfaces. The only difference between the proxy for an arbitrary service and the EJB is the set of interfaces exposed as well as the client side interceptors found inside the proxy. The client interceptors are represented by the rectangles found inside of the client proxy. An interceptor is an assembly line type of pattern that allows for transformation of a method invocation and/or return values. A client obtains a proxy through some lookup mechanism, typically JNDI. Although RMI is indicated in [Figure 5.21, “The main components in the detached invoker architecture”](#), the only real requirement on the exposed interface and its types is that they are serializable between the client server over JNDI as well as the transport layer.

The choice of the transport layer is determined by the last interceptor in the client proxy, which is referred to as the *Invoker Interceptor* in [Figure 5.21](#), “[The main components in the detached invoker architecture](#)”. The invoker interceptor contains a reference to the transport specific stub of the server side *Detached Invoker* MBean service. The invoker interceptor also handles the optimization of calls that occur within the same VM as the target MBean. When the invoker interceptor detects that this is the case the call is passed to a call-by-reference invoker that simply passes the invocation along to the target MBean.

The detached invoker service is responsible for making a generic invoke operation available via the transport the detached invoker handles. The **Invoker** interface illustrates the generic invoke operation.

```
package org.jboss.invocation;

import java.rmi.Remote;
import org.jboss.proxy.Interceptor;
import org.jboss.util.id.GUID;

public interface Invoker
    extends Remote
{
    GUID ID = new GUID();

    String getServerHostName() throws Exception;

    Object invoke(Invocation invocation) throws Exception;
}
```

The **Invoker** interface extends **Remote** to be compatible with RMI, but this does not mean that an invoker must expose an RMI service stub. The detached invoker service simply acts as a transport gateway that accepts invocations represented as the **org.jboss.invocation.Invocation** object over its specific transport, unmarshalls the invocation, forwards the invocation onto the destination MBean service, represented by the *Target MBean* in [Figure 5.21, “The main components in the detached invoker architecture”](#), and marshalls the return value or exception resulting from the forwarded call back to the client.

The **Invocation** object is just a representation of a method invocation context. This includes the target MBean name, the method, the method arguments, a context of information associated with the proxy by the proxy factory, and an arbitrary map of data associated with the invocation by the client proxy interceptors.

The configuration of the client proxy is done by the server side proxy factory MBean service, indicated by the *Proxy Factory* component in [Figure 5.21, “The main components in the detached invoker architecture”](#). The proxy factory performs the following tasks:

- Create a dynamic proxy that implements the interface the target MBean wishes to expose.
- Associate the client proxy interceptors with the dynamic proxy handler.
- Associate the invocation context with the dynamic proxy. This includes the target MBean, detached invoker stub and the proxy JNDI name.
- Make the proxy available to clients by binding the proxy into JNDI.

The last component in [Figure 5.21, “The main components in the detached invoker architecture”](#) is the *Target MBean* service that wishes to expose an interface for invocations to remote clients. The steps required for an MBean service to be accessible through a given interface are:

- Define a JMX operation matching the signature: **public Object invoke(org.jboss.invocation.Invocation) throws Exception**
- Create a **HashMap<Long, Method>** mapping from the exposed interface **java.lang.reflect.Methods** to the long hash representation using the **org.jboss.invocation.MarshalledInvocation.calculateHash** method.

- Implement the **invoke(Invocation)** JMX operation and use the interface method hash mapping to transform from the long hash representation of the invoked method to the **java.lang.reflect.Method** of the exposed interface. Reflection is used to perform the actual invocation on the object associated with the MBean service that actually implements the exposed interface.

5.6.1. A Detached Invoker Example, the MBeanServer Invoker Adaptor Service

In the section on connecting to the JMX server we mentioned that there was a service that allows one to access the **javax.management.MBeanServer** via any protocol using an invoker service. In this section we present the **org.jboss.jmx.connector.invoker.InvokerAdaptorService** and its configuration for access via RMI/JRMP as an example of the steps required to provide remote access to an MBean service.

The **InvokerAdaptorService** is a simple MBean service that only exists to fulfill the target MBean role in the detached invoker pattern.

Example 5.16. The InvokerAdaptorService MBean

```
package org.jboss.jmx.connector.invoker;
public interface InvokerAdaptorServiceMBean
    extends org.jboss.system.ServiceMBean
{
    Class getExportedInterface();
    void setExportedInterface(Class exportedInterface);

    Object invoke(org.jboss.invocation.Invocation invocation)
        throws Exception;
}

package org.jboss.jmx.connector.invoker;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.UndeclaredThrowableException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;
import org.jboss.mx.server.ServerConstants;
import org.jboss.system.ServiceMBeanSupport;
import org.jboss.system.Registry;

public class InvokerAdaptorService
    extends ServiceMBeanSupport
    implements InvokerAdaptorServiceMBean, ServerConstants
{
    private static ObjectName mbeanRegistry;

    static {
```

```
        try {
            mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }

    private Map marshalledInvocationMapping = new HashMap();
    private Class exportedInterface;

    public Class getExportedInterface()
    {
        return exportedInterface;
    }

    public void setExportedInterface(Class exportedInterface)
    {
        this.exportedInterface = exportedInterface;
    }

    protected void startService()
        throws Exception
    {
        // Build the interface method map
        Method[] methods = exportedInterface.getMethods();
        HashMap tmpMap = new HashMap(methods.length);
        for (int m = 0; m < methods.length; m++) {
            Method method = methods[m];
            Long hash = new
Long(MarshalledInvocation.calculateHash(method));
            tmpMap.put(hash, method);
        }

        marshalledInvocationMapping =
Collections.unmodifiableMap(tmpMap);
        // Place our ObjectName hash into the Registry so invokers can
        // resolve it
        Registry.bind(new Integer(serviceName.hashCode()),
serviceName);
    }

    protected void stopService()
        throws Exception
    {
        Registry.unbind(new Integer(serviceName.hashCode()));
    }

    public Object invoke(Invocation invocation)
        throws Exception
    {
        // Make sure we have the correct classloader before
        unmarshalling
        Thread thread = Thread.currentThread();
        ClassLoader oldCL = thread.getContextClassLoader();
```

```

        // Get the MBean this operation applies to
        ClassLoader newCL = null;
        ObjectName objectName = (ObjectName)
            invocation.getValue("JMX_OBJECT_NAME");
        if (objectName != null) {
            // Obtain the ClassLoader associated with the MBean
deployment
            newCL = (ClassLoader)
                server.invoke(mbeanRegistry, "getValue",
                    new Object[] { objectName, CLASSLOADER
},
                    new String[] {
ObjectName.class.getName(),
                                "java.lang.String" });
        }

        if (newCL != null && newCL != oldCL) {
            thread.setContextClassLoader(newCL);
        }

        try {
            // Set the method hash to Method mapping
            if (invocation instanceof MarshalledInvocation) {
                MarshalledInvocation mi = (MarshalledInvocation)
invocation;
                mi.setMethodMap(marshalledInvocationMapping);
            }

            // Invoke the MBeanServer method via reflection
            Method method = invocation.getMethod();
            Object[] args = invocation.getArguments();
            Object value = null;
            try {
                String name = method.getName();
                Class[] sig = method.getParameterTypes();
                Method mbeanServerMethod =
                    MBeanServer.class.getMethod(name, sig);
                value = mbeanServerMethod.invoke(server, args);
            } catch (InvocationTargetException e) {
                Throwable t = e.getTargetException();
                if (t instanceof Exception) {
                    throw (Exception) t;
                } else {
                    throw new UndeclaredThrowableException(t,
method.toString());
                }
            }

            return value;
        } finally {
            if (newCL != null && newCL != oldCL) {
                thread.setContextClassLoader(oldCL);
            }
        }
    }
}

```

Let's go through the key details of this service. The **InvokerAdaptorServiceMBean** Standard MBean interface of the **InvokerAdaptorService** has a single **ExportedInterface** attribute and a single **invoke(Invocation)** operation. The **ExportedInterface** attribute allows customization of the type of interface the service exposes to clients. This has to be compatible with the **MBeanServer** class in terms of method name and signature. The **invoke(Invocation)** operation is the required entry point that target MBean services must expose to participate in the detached invoker pattern. This operation is invoked by the detached invoker services that have been configured to provide access to the **InvokerAdaptorService**.

Lines 54-64 of the **InvokerAdaptorService** build the **HashMap<Long, Method>** of the **ExportedInterface** Class using the **org.jboss.invocation.MarshalledInvocation.calculateHash(Method)** utility method. Because **java.lang.reflect.Method** instances are not serializable, a **MarshalledInvocation** version of the non-serializable **Invocation** class is used to marshall the invocation between the client and server. The **MarshalledInvocation** replaces the **Method** instances with their corresponding hash representation. On the server side, the **MarshalledInvocation** must be told what the hash to **Method** mapping is.

Line 64 creates a mapping between the **InvokerAdaptorService** service name and its hash code representation. This is used by detached invokers to determine what the target MBean **ObjectName** of an **Invocation** is. When the target MBean name is store in the **Invocation**, its store as its **hashCode** because **ObjectNames** are relatively expensive objects to create. The **org.jboss.system.Registry** is a global map like construct that invokers use to store the hash code to **ObjectName** mappings in.

Lines 77-93 obtain the name of the MBean on which the MBeanServer operation is being performed and lookup the class loader associated with the MBean's SAR deployment. This information is available via the **org.jboss.mx.server.registry.BasicMBeanRegistry**, a JBoss JMX implementation specific class. It is generally necessary for an MBean to establish the correct class loading context because the detached invoker protocol layer may not have access to the class loaders needed to unmarshall the types associated with an invocation.

Lines 101-105 install the **ExposedInterface** class method hash to method mapping if the invocation argument is of type **MarshalledInvocation**. The method mapping calculated previously at lines 54-62 is used here.

Lines 107-114 perform a second mapping from the **ExposedInterface** Method to the matching method of the MBeanServer class. The **InvokerServiceAdaptor** decouples the **ExposedInterface** from the MBeanServer class in that it allows an arbitrary interface. This is needed on one hand because the standard **java.lang.reflect.Proxy** class can only proxy interfaces. It also allows one to only expose a subset of the MBeanServer methods and add transport specific exceptions like **java.rmi.RemoteException** to the **ExposedInterface** method signatures.

Line 115 dispatches the MBeanServer method invocation to the MBeanServer instance to which the **InvokerAdaptorService** was deployed. The server instance variable is inherited from the **ServiceMBeanSupport** superclass.

Lines 117-124 handle any exceptions coming from the reflective invocation including the unwrapping of any declared exception thrown by the invocation.

Line 126 is the return of the successful MBeanServer method invocation result.

Note that the **InvokerAdapterService** MBean does not deal directly with any transport specific details. There is the calculation of the method hash to Method mapping, but this is a transport independent detail.

Now let's take a look at how the **InvokerAdapterService** may be used to expose the same **org.jboss.jmx.adaptor.rmi.RMIAdaptor** interface via RMI/JRMP as seen in Connecting to JMX Using RMI. We will start by presenting the proxy factory and **InvokerAdapterService** configurations found in the default setup in the **jmx-invoker-adaptor-server.sar** deployment. [Example 5.17](#), “The default **jmx-invoker-adaptor-server.sar** **jboss-service.xml** deployment descriptor” shows the **jboss-service.xml** descriptor for this deployment.

Example 5.17. The default **jmx-invoker-adaptor-server.sar** **jboss-service.xml** deployment descriptor

```
<server>
  <!-- The JRMP invoker proxy configuration for the
  InvokerAdapterService -->
  <mbean code="org.jboss.invocation.jrmp.server.JRMPProxyFactory"

  name="jboss.jmx:type=adaptor,name=Invoker,protocol=jrmp,service=proxyFac
  tory">
    <!-- Use the standard JRMPInvoker from conf/jboss-service.xml --
  >
    <attribute
  name="InvokerName">jboss:service=invoker,type=jrmp</attribute>
    <!-- The target MBean is the InvokerAdapterService configured
  below -->
    <attribute
  name="TargetName">jboss.jmx:type=adaptor,name=Invoker</attribute>
    <!-- Where to bind the RMIAdaptor proxy -->
    <attribute name="JndiName">jmx/invoker/RMIAdaptor</attribute>
    <!-- The RMI compabitle MBeanServer interface -->
    <attribute
  name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute
  >
    <attribute name="ClientInterceptors">
      <interceptors>

<interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
      </interceptors>

      org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
      </interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
      </interceptors>
    </attribute>
    <depends>jboss:service=invoker,type=jrmp</depends>
  </mbean>
  <!-- This is the service that handles the RMIAdaptor invocations by
  routing
  them to the MBeanServer the service is deployed under. -->
  <mbean code="org.jboss.jmx.connector.invoker.InvokerAdapterService"
    name="jboss.jmx:type=adaptor,name=Invoker">
    <attribute
```

```

name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute
>
    </mbean>
</server>

```

The first MBean, **org.jboss.invocation.jrmp.server.JRMPProxyFactory**, is the proxy factory MBean service that creates proxies for the RMI/JRMP protocol. The configuration of this service as shown in [Example 5.17](#), “[The default jmx-invoker-adaptor-server.sar jboss-service.xml deployment descriptor](#)” states that the JRMPInvoker will be used as the detached invoker, the **InvokerAdaptorService** is the target mbean to which requests will be forwarded, that the proxy will expose the **RMIAdaptor** interface, the proxy will be bound into JNDI under the name **jmx/invoker/RMIAdaptor**, and the proxy will contain 3 interceptors: **ClientMethodInterceptor**, **InvokerAdaptorClientInterceptor**, **InvokerInterceptor**. The configuration of the **InvokerAdaptorService** simply sets the RMIAdaptor interface that the service is exposing.

The last piece of the configuration for exposing the **InvokerAdaptorService** via RMI/JRMP is the detached invoker. The detached invoker we will use is the standard RMI/JRMP invoker used by the EJB containers for home and remote invocations, and this is the **org.jboss.invocation.jrmp.server.JRMPInvoker** MBean service configured in the **conf/jboss-service.xml** descriptor. That we can use the same service instance emphasizes the detached nature of the invokers. The JRMPInvoker simply acts as the RMI/JRMP endpoint for all RMI/JRMP proxies regardless of the interface(s) the proxies expose or the service the proxies utilize.

5.6.2. Detached Invoker Reference

5.6.2.1. The JRMPInvoker - RMI/JRMP Transport

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

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

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

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

- **SecurityDomain**: specifies the JNDI name of an **org.jboss.security.SecurityDomain** interface implementation to associate with the **RMIServerSocketFactory** implementation. The value will be passed to the **RMIServerSocketFactory** using reflection to locate a method with a signature of **public void setSecurityDomain(org.jboss.security.SecurityDomain d)**. If no such method exists the **SecurityDomain** value will be ignored.

5.6.2.2. The PooledInvoker - RMI/Socket Transport

The **org.jboss.invocation.pooled.server.PooledInvoker** is an MBean service that provides RMI over a custom socket transport implementation of the **Invoker** interface. The **PooledInvoker** exports itself as an RMI server so that when it is used as the **Invoker** in a remote client, the **PooledInvoker** stub is sent to the client instead and invocations use the custom socket protocol.

The **PooledInvoker** MBean supports a number of attribute to configure the socket transport layer. Its configurable attributes are:

- **NumAcceptThreads**: The number of threads that exist for accepting client connections. The default is 1.
- **MaxPoolSize**: The number of server threads for processing client. The default is 300.
- **SocketTimeout**: The socket timeout value passed to the **Socket.setSoTimeout()** method. The default is 60000.
- **ServerBindPort**: The port used for the server socket. A value of 0 indicates that an anonymous port should be chosen.
- **ClientConnectAddress**: The address that the client passes to the **Socket(addr, port)** constructor. This defaults to the server **InetAddress.getLocalHost()** value.
- **ClientConnectPort**: The port that the client passes to the **Socket(addr, port)** constructor. The default is the port of the server listening socket.
- **ClientMaxPoolSize**: The client side maximum number of threads. The default is 300.
- **Backlog**: The backlog associated with the server accept socket. The default is 200.
- **EnableTcpNoDelay**: A boolean flag indicating if client sockets will enable the **TcpNoDelay** flag on the socket. The default is false.
- **ServerBindAddress**: The address on which the server binds its listening socket. The default is an empty value which indicates the server should be bound on all interfaces.
- **TransactionManagerService**: The JMX ObjectName of the JTA transaction manager service.

5.6.2.3. The IIOPInvoker - RMI/IIOP Transport

The **org.jboss.invocation.iiop.IIOPInvoker** class is an MBean service that provides the RMI/IIOP implementation of the **Invoker** interface. The **IIOPInvoker** routes IIOP requests to CORBA servants. This is used by the **org.jboss.proxy.ejb.IORFactory** proxy factory to create RMI/IIOP

proxies. However, rather than creating Java proxies (as the JRMP proxy factory does), this factory creates CORBA IORs. An **IORFactory** is associated to a given enterprise bean. It registers with the IIOPI invoker two CORBA servants: an **EjbHomeCorbaServant** for the bean's **EJBHome** and an **EjbObjectCorbaServant** for the bean's **EJBObjects**.

The IIOPIInvoker MBean has no configurable properties, since all properties are configured from the `conf/jacorb.properties` property file used by the JacORB CORBA service.

5.6.2.4. The JRMPProxyFactory Service - Building Dynamic JRMP Proxies

The `org.jboss.invocation.jrmp.server.JRMPProxyFactory` MBean service is a proxy factory that can expose any interface with RMI compatible semantics for access to remote clients using JRMP as the transport.

The JRMPProxyFactory supports the following attributes:

- **InvokerName**: The server side JRMPInvoker MBean service JMX ObjectName string that will handle the RMI/JRMP transport.
- **TargetName**: The server side MBean that exposes the **invoke(Invocation)** JMX operation for the exported interface. This is used as the destination service for any invocations done through the proxy.
- **JndiName**: The JNDI name under which the proxy will be bound.
- **ExportedInterface**: The fully qualified class name of the interface that the proxy implements. This is the typed view of the proxy that the client uses for invocations.
- **ClientInterceptors**: An XML fragment of interceptors/interceptor elements with each interceptor element body specifying the fully qualified class name of an `org.jboss.proxy.Interceptor` implementation to include in the proxy interceptor stack. The ordering of the interceptors/interceptor elements defines the order of the interceptors.

5.6.2.5. The HttpInvoker - RMI/HTTP Transport

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

The HttpInvoker supports the following attributes:

- **InvokerURL**: This is either the http URL to the **InvokerServlet** mapping, or the name of a system property that will be resolved inside the client VM to obtain the http URL to the **InvokerServlet**.
- **InvokerURLPrefix**: If there is no **invokerURL** set, then one will be constructed via the concatenation of **invokerURLPrefix** + the local host + **invokerURLSuffix**. The default prefix is `http://`.

- **InvokerURLSuffix**: If there is no **invokerURL** set, then one will be constructed via the concatenation of **invokerURLPrefix** + the local host + **invokerURLSuffix**. The default suffix is **:8080/invoker/JMXInvokerServlet**.
- **UseHostName**: A boolean flag if the **InetAddress.getHostName()** or **getHostAddress()** method should be used as the host component of **invokerURLPrefix** + host + **invokerURLSuffix**. If true **getHostName()** is used, otherwise **getHostAddress()** is used.

5.6.2.6. The HA JRMPInvoker - Clustered RMI/JRMP Transport

The **org.jboss.proxy.generic.ProxyFactoryHA** service is an extension of the **ProxyFactoryHA** that is a cluster aware factory. The **ProxyFactoryHA** fully supports all of the attributes of the **JRMPProxyFactory**. This means that customized bindings of the port, interface and socket transport are available to clustered RMI/JRMP as well. In addition, the following cluster specific attributes are supported:

- **PartitionObjectName**: The JMX **ObjectName** of the cluster service to which the proxy is to be associated with.
- **LoadBalancePolicy**: The class name of the **org.jboss.ha.framework.interfaces.LoadBalancePolicy** interface implementation to associate with the proxy.

5.6.2.7. The HA HttpInvoker - Clustered RMI/HTTP Transport

The RMI/HTTP layer allows for software load balancing of the invocations in a clustered environment. The HA capable extension of the HTTP invoker borrows much of its functionality from the HA-RMI/JRMP clustering. To enable HA-RMI/HTTP you need to configure the invokers for the EJB container. This is done through either a **jboss.xml** descriptor, or the **standardjboss.xml** descriptor.

5.6.2.8. HttpProxyFactory - Building Dynamic HTTP Proxies

The **org.jboss.invocation.http.server.HttpProxyFactory** MBean service is a proxy factory that can expose any interface with RMI compatible semantics for access to remote clients using HTTP as the transport.

The **HttpProxyFactory** supports the following attributes:

- **InvokerName**: The server side MBean that exposes the invoke operation for the exported interface. The name is embedded into the **HttpInvokerProxy** context as the target to which the invocation should be forwarded by the **HttpInvoker**.
- **JndiName**: The JNDI name under which the **HttpInvokerProxy** will be bound. This is the name clients lookup to obtain the dynamic proxy that exposes the service interfaces and marshalls invocations over HTTP. This may be specified as an empty value to indicate that the proxy should not be bound into JNDI.
- **InvokerURL**: This is either the http URL to the **InvokerServlet** mapping, or the name of a system property that will be resolved inside the client VM to obtain the http URL to the **InvokerServlet**.

- **InvokerURLPrefix**: If there is no **invokerURL** set, then one will be constructed via the concatenation of **invokerURLPrefix** + the local host + **invokerURLSuffix**. The default prefix is **http://**.
- **InvokerURLSuffix**: If there is no **invokerURL** set, then one will be constructed via the concatenation of **invokerURLPrefix** + the local host + **invokerURLSuffix**. The default suffix is **:8080/invoker/JMXInvokerServlet**.
- **UseHostName**: A boolean flag indicating if the **InetAddress.getHostName()** or **getHostAddress()** method should be used as the host component of **invokerURLPrefix** + host + **invokerURLSuffix**. If true **getHostName()** is used, otherwise **getHostAddress()** is used.
- **ExportedInterface**: The name of the RMI compatible interface that the **HttpInvokerProxy** implements.

5.6.2.9. Steps to Expose Any RMI Interface via HTTP

Using the **HttpProxyFactory** MBean and JMX, you can expose any interface for access using HTTP as the transport. The interface to expose does not have to be an RMI interface, but it does have to be compatible with RMI in that all method parameters and return values are serializable. There is also no support for converting RMI interfaces used as method parameters or return values into their stubs.

The three steps to making your object invocable via HTTP are:

- Create a mapping of longs to the RMI interface methods using the **MarshaledInvocation.calculateHash** method. Here for example, is the procedure for an RMI **SRPRemoteServerInterface** interface:

```
import java.lang.reflect.Method;
import java.util.HashMap;
import org.jboss.invocation.MarshaledInvocation;

HashMap marshalledInvocationMapping = new HashMap();

// Build the Naming interface method map
Method[] methods = SRPRemoteServerInterface.class.getMethods();
for(int m = 0; m < methods.length; m++) {
    Method method = methods[m];
    Long hash = new
Long(MarshaledInvocation.calculateHash(method));
    marshalledInvocationMapping.put(hash, method);
}
```

- Either create or extend an existing MBean to support an invoke operation. Its signature is **Object invoke(Invocation invocation) throws Exception**, and the steps it performs are as shown here for the **SRPRemoteServerInterface** interface. Note that this uses the **marshalledInvocationMapping** from step 1 to map from the **Long** method hashes in the **MarshaledInvocation** to the **Method** for the interface.

```
import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshaledInvocation;

public Object invoke(Invocation invocation)
```

```

        throws Exception
    {
        SRPRemoteServerInterface theServer =
<the_actual_rmi_server_object>;
        // Set the method hash to Method mapping
        if (invocation instanceof MarshalledInvocation) {
            MarshalledInvocation mi = (MarshalledInvocation) invocation;
            mi.setMethodMap(marshalledInvocationMapping);
        }

        // Invoke the Naming method via reflection
        Method method = invocation.getMethod();
        Object[] args = invocation.getArguments();
        Object value = null;
        try {
            value = method.invoke(theServer, args);
        } catch (InvocationTargetException e) {
            Throwable t = e.getTargetException();
            if (t instanceof Exception) {
                throw (Exception) e;
            } else {
                throw new UndeclaredThrowableException(t,
method.toString());
            }
        }

        return value;
    }
}

```

- Create a configuration of the **HttpProxyFactory** MBean to make the RMI/HTTP proxy available through JNDI. For example:

```

<!-- Expose the SRP service interface via HTTP -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
    name="jboss.security.tests:service=SRP/HTTP">
    <attribute
name="InvokerURL">http://localhost:8080/invoker/JMXInvokerServlet</a
ttribute>
    <attribute
name="InvokerName">jboss.security.tests:service=SRPService</attribut
e>
    <attribute name="ExportedInterface">
        org.jboss.security.srp.SRPRemoteServerInterface
    </attribute>
    <attribute name="JndiName">srp-test-
http/SRPServiceInterface</attribute>
</mbean>

```

Any client may now lookup the RMI interface from JNDI using the name specified in the **HttpProxyFactory** (e.g., **srp-test-http/SRPServiceInterface**) and use the obtain proxy in exactly the same manner as the RMI/JRMP version.

CHAPTER 6. NAMING ON JBOSS

The JNDI Naming Service

The naming service plays a key role in enterprise Java applications, providing the core infrastructure that is used to locate objects or services in an application server. It is also the mechanism that clients external to the application server use to locate services inside the application server. Application code, whether it is internal or external to the JBoss instance, need only know that it needs to talk to the a message queue named **queue/IncomingOrders** and would not need to worry about any of the details of how the queue is configured. In a clustered environment, naming services are even more valuable. A client of a service would desire to look up the **ProductCatalog** session bean from the cluster without worrying which machine the service is residing. Whether it is a big clustered service, a local resource or just a simple application component that is needed, the JNDI naming service provides the glue that lets code find the objects in the system by name.

6.1. AN OVERVIEW OF JNDI

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

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

6.1.1. Names

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

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

The **javax.naming.Name** interface represents a generic name as an ordered sequence of components. It can be a composite name (one that spans multiple namespaces), or a compound name (one that is used within a single hierarchical naming system). The components of a name are numbered.

The indexes of a name with N components range from 0 up to, but not including, N. The most significant component is at index 0. An empty name has no components.

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

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

A compound name is derived from a hierarchical namespace. Each component in a compound name is an atomic name, meaning a string that cannot be parsed into smaller components. A file pathname in the UNIX file system is an example of a compound name. **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.

6.1.2. 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.

6.1.2.1. 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.

- The first occurrence of the property from the constructor's environment parameter and (for appropriate properties) the applet parameters and system properties.
- All **java.naming.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, which allows your code to externalize the JNDI provider specific information so that changing JNDI providers will not require changes to your code or recompilation.

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.

[Example 6.1, “A sample jndi.properties file”](#) 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.

Example 6.1. A sample jndi.properties file

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

6.2. 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 6.1, “Key components in the JBossNS architecture.”](#) illustrates some of the key classes in the JBossNS implementation and their relationships.

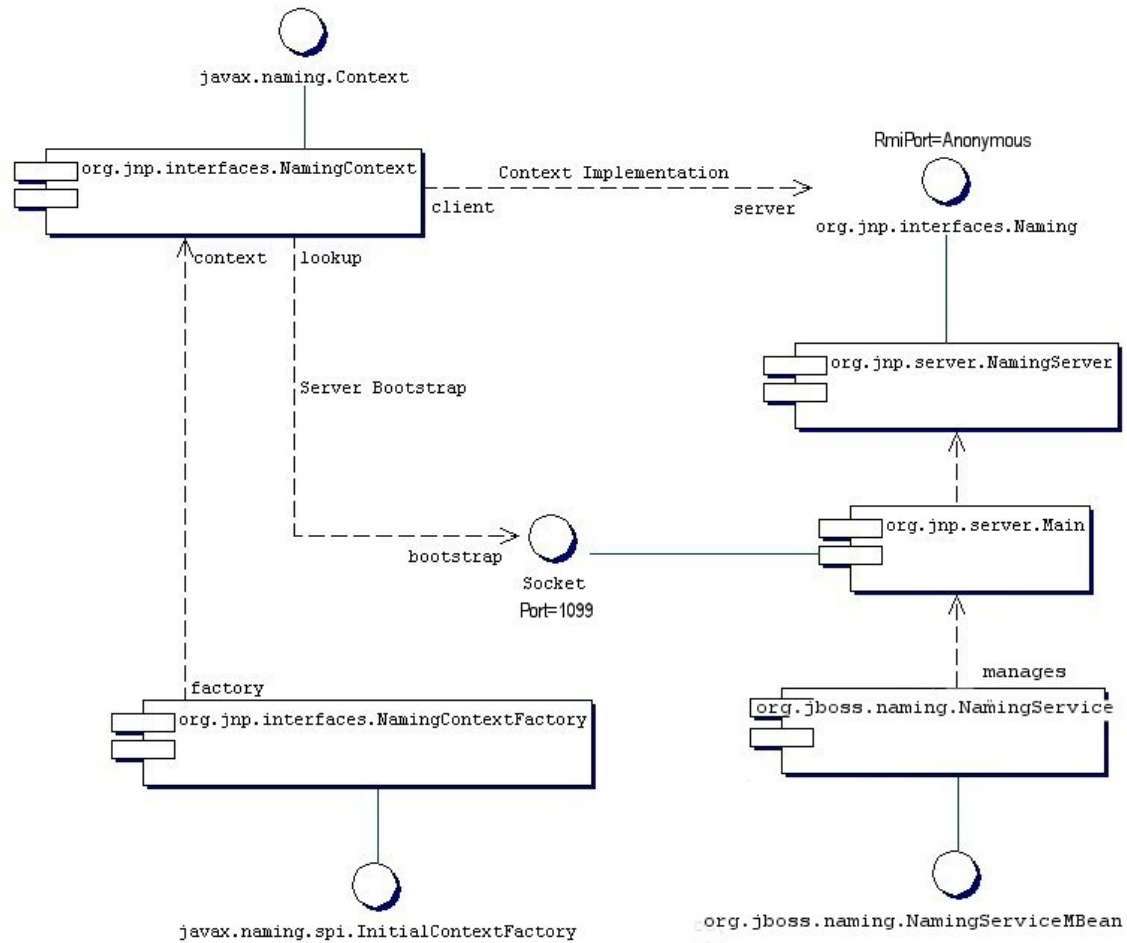


Figure 6.1. Key components in the JBossNS architecture.

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.
- **RmiBindAddress**: The specific address the RMI server portion of 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. If this is not specified and the **BindAddress** is, the **RmiBindAddress** defaults to the **BindAddress** value.
- **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.
- **JNPServerSocketFactory**: An optional custom `javax.net.ServerSocketFactory` implementation class name. This is the factory for the **ServerSocket** used to bootstrap the download of the JBossNS **Naming** interface. If not specified the `javax.net.ServerSocketFactory.getDefault()` method value is used.

The **NamingService** also creates the `java:comp` context such that access to this context is isolated based on the context class loader of the thread that accesses the `java:comp` context. This provides the application component private ENC that is required by the J2EE specs. This segregation 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 class loader instance, one is created and associated with that class loader in the **ENCFactory** map. Thus, correct isolation of an application component's ENC relies on each component receiving a unique **ClassLoader** that is associated with the component threads of execution.

The **NamingService** delegates its functionality to an `org.jnp.server.Main` MBean. The reason for the duplicate MBeans is because JBossNS started out as a stand-alone JNDI implementation, and can still be run as such. The **NamingService** MBean embeds the **Main** instance into the JBoss server so that usage of JNDI with the same VM as the JBoss server does not incur any socket overhead. The configurable attributes of the **NamingService** are really the configurable attributes of the JBossNS **Main** MBean. The setting of any attributes on the **NamingService** MBean simply set the corresponding attributes on the **Main** MBean the **NamingService** contains. When the **NamingService** is started, it starts the contained **Main** MBean to activate the JNDI naming service.

In addition, the **NamingService** exposes the **Naming** interface operations through a JMX detyped invoke operation. This allows the naming service to be accessed via JMX adaptors for arbitrary protocols. We will look at an example of how HTTP can be used to access the naming service using the invoke operation later in this chapter.

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

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

- Instantiates an `org.jnp.naming.NamingService` instance and sets this as the local VM server instance. This is used by any `org.jnp.interfaces.NamingContext` instances that are created within the JBoss server VM to avoid RMI calls over TCP/IP.
- Exports the **NamingServer** instance's `org.jnp.naming.interfaces.Naming` RMI interface using the configured **RmiPort**, **ClientSocketFactory**, **ServerSocketFactory** attributes.
- Creates a socket that listens on the interface given by the **BindAddress** and **Port** attributes.
- Spawns a thread to accept connections on the socket.

6.3. THE NAMING INITIALCONTEXT FACTORIES

The JBoss JNDI provider currently supports several different **InitialContext** factory implementations.

6.3.1. The standard naming context factory

The most commonly used factory is the **org.jnp.interfaces.NamingContextFactory** implementation. Its properties include:

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

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

6.3.2. The org.jboss.naming.NamingContextFactory

This version of the **InitialContextFactory** implementation is a simple extension of the jnp version which differs from the jnp version in that it stores the last configuration passed to its **InitialContextFactory.getInitialContext(Hashtable env)** method in a public thread local variable. This is used by EJB handles and other JNDI sensitive objects like the **UserTransaction** factory to keep track of the JNDI context that was in effect when they were created. If you want this environment to be bound to the object even after its serialized across vm boundaries, then you should use the **org.jboss.naming.NamingContextFactory**. If you want the environment that is defined in the current VM **jndi.properties** or system properties, then you should use the **org.jnp.interfaces.NamingContextFactory** version.

6.3.3. Naming Discovery in Clustered Environments

When running in a clustered JBoss environment, you can choose not to specify a **Context.PROVIDER_URL** value and let the client query the network for available naming services. This only works with JBoss servers running with the **all** configuration, or an equivalent configuration that has **org.jboss.ha.framework.server.ClusterPartition** and **org.jboss.ha.jndi.HANamingService** services deployed. The discovery process consists of sending a multicast request packet to the discovery address/port and waiting for any node to respond. The response is a HA-RMI version of the **Naming** interface. The following **InitialContext** properties affect the discovery configuration:

- **jnp.partitionName**: The cluster partition name discovery should be restricted to. If you are

running in an environment with multiple clusters, you may want to restrict the naming discovery to a particular cluster. There is no default value, meaning that any cluster response will be accepted.

- **jnp.discoveryGroup**: The multicast IP/address to which the discovery query is sent. The default is 230.0.0.4.
- **jnp.discoveryPort**: The port to which the discovery query is sent. The default is 1102.
- **jnp.discoveryTimeout**: The time in milliseconds to wait for a discovery query response. The default value is 5000 (5 seconds).
- **jnp.disableDiscovery**: A flag indicating if the discovery process should be avoided. Discovery occurs when either no **Context.PROVIDER_URL** is specified, or no valid naming service could be located among the URLs specified. If the **jnp.disableDiscovery** flag is true, then discovery will not be attempted.

6.3.4. The HTTP InitialContext Factory Implementation

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

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

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory, which must be **org.jboss.naming.HttpNamingContextFactory**.
- **java.naming.provider.url** (or **Context.PROVIDER_URL**): This must be set to the HTTP URL of the JNDI factory. The full HTTP URL would be the public URL of the JBoss servlet container plus **/invoker/JNDIFactory**. Examples include:
 - **http://www.jboss.org:8080/invoker/JNDIFactory**
 - **http://www.jboss.org/invoker/JNDIFactory**
 - **https://www.jboss.org/invoker/JNDIFactory**

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

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

The JNDI **Context** implementation returned by the **HttpNamingContextFactory** is a proxy that delegates invocations made on it to a bridge servlet which forwards the invocation to the **NamingService** through the JMX bus and marshalls the reply back over HTTP. The proxy needs to know what the URL of the bridge servlet is in order to operate. This value may have been bound on the server side if the JBoss web server has a well known public interface. If the JBoss web server is sitting

behind one or more firewalls or proxies, the proxy cannot know what URL is required. In this case, the proxy will be associated with a system property value that must be set in the client VM. For more information on the operation of JNDI over HTTP see [Section 6.4.1, “Accessing JNDI over HTTP”](#).

6.3.5. The Login InitialContext Factory Implementation

JAAS is the preferred method for authenticating a remote client to JBoss. However, for simplicity and to ease the migration from other application server environment that do not use JAAS, JBoss allows you the security credentials to be passed through the **InitialContext**. JAAS is still used under the covers, but there is no manifest use of the JAAS interfaces in the client application.

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

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory, which must be **org.jboss.security.jndi.LoginInitialContextFactory**.
- **java.naming.provider.url**: This must be set to a **NamingContextFactory** provider URL. The **LoginInitialContext** is really just a wrapper around the **NamingContextFactory** that adds a JAAS login to the existing **NamingContextFactory** behavior.
- **java.naming.factory.url.pkgs**: For all JBoss JNDI provider this must be **org.jboss.naming:org.jnp.interfaces**. This property is essential for locating the **jnp:** and **java:** URL context factories of the JBoss JNDI provider.
- **java.naming.security.principal** (or **Context.SECURITY_PRINCIPAL**): The principal to authenticate. This may be either a **java.security.Principal** implementation or a string representing the name of a principal.
- **java.naming.security.credentials** (or **Context.SECURITY_CREDENTIALS**), The credentials that should be used to authenticate the principal, e.g., password, session key, etc.
- **java.naming.security.protocol**: (**Context.SECURITY_PROTOCOL**) This gives the name of the JAAS login module to use for the authentication of the principal and credentials.

6.3.6. The ORBInitialContextFactory

When using Sun's CosNaming it is necessary to use a different naming context factory from the default. CosNaming looks for the ORB in JNDI instead of using the the ORB configured in **deploy/iiop-service.xml?**. It is necessary to set the global context factory to **org.jboss.iiop.naming.ORBInitialContextFactory**, which sets the ORB to JBoss's ORB. This is done in the **conf/jndi.properties** file:

```
# DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING
#
java.naming.factory.initial=org.jboss.iiop.naming.ORBInitialContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

It is also necessary to use **ORBInitialContextFactory** when using CosNaming in an application client.

6.4. JNDI OVER HTTP

In addition to the legacy RMI/JRMP with a socket bootstrap protocol, JBoss provides support for accessing its JNDI naming service over HTTP.

6.4.1. Accessing JNDI over HTTP

This capability is provided by `http-invoker.sar`. The structure of the `http-invoker.sar` is:

```

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

```

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

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

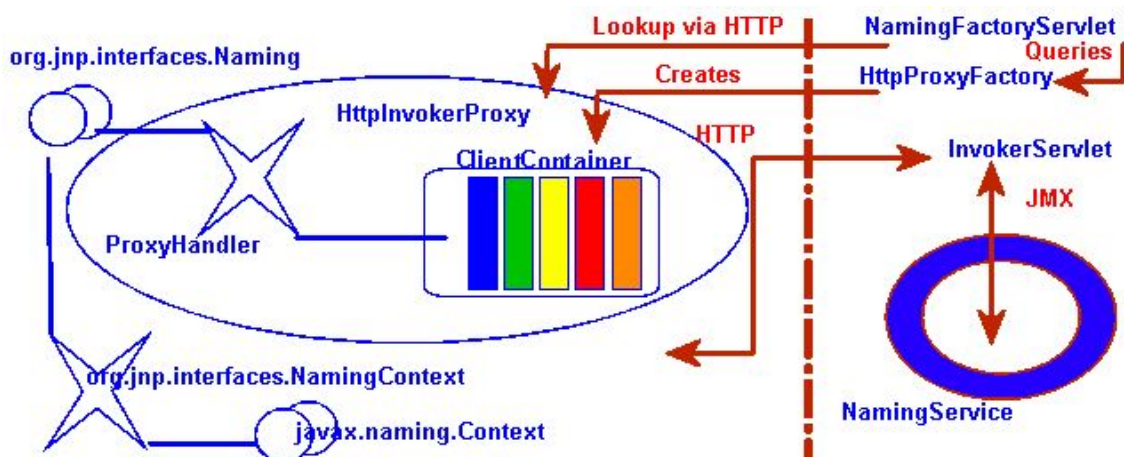


Figure 6.2. The HTTP invoker proxy/server structure for a JNDI Context

Before looking at the configurations let's look at the operation of the `http-invoker` services. Figure 6.2, "The HTTP invoker proxy/server structure for a JNDI Context" shows a logical view of the structure of a JBoss JNDI proxy and its relationship to the JBoss server side components of the `http-invoker`. The proxy is obtained from the `NamingFactoryServlet` using an `InitialContext` with the

`Context.INITIAL_CONTEXT_FACTORY` property set to `org.jboss.naming.HttpNamingContextFactory`, and the `Context.PROVIDER_URL` property set to the HTTP URL of the `NamingFactoryServlet`. The resulting proxy is embedded in an `org.jnp.interfaces.NamingContext` instance that provides the `Context` interface implementation.

The proxy is an instance of `org.jboss.invocation.http.interfaces.HttpInvokerProxy`, and implements the `org.jnp.interfaces.Naming` interface. Internally the `HttpInvokerProxy` contains an invoker that marshalls the `Naming` interface method invocations to the `InvokerServlet` via HTTP posts. The `InvokerServlet` translates these posts into JMX invocations to the `NamingService`, and returns the invocation response back to the proxy in the HTTP post response.

There are several configuration values that need to be set to tie all of these components together and [Figure 6.3, “The relationship between configuration files and JNDI/HTTP component”](#) illustrates the relationship between configuration files and the corresponding components.

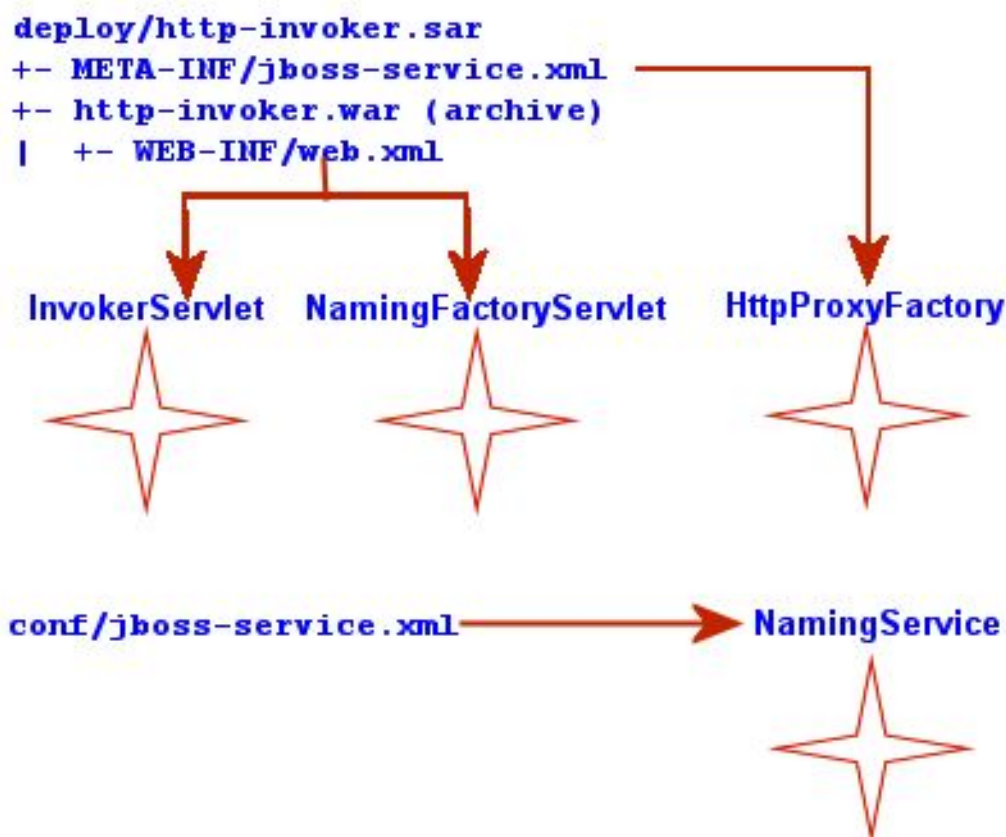


Figure 6.3. The relationship between configuration files and JNDI/HTTP component

The `http-invoker.sar/META-INF/jboss-service.xml` descriptor defines the `HttpProxyFactory` that creates the `HttpInvokerProxy` for the `NamingService`. The attributes that need to be configured for the `HttpProxyFactory` include:

- **InvokerName:** The JMX `ObjectName` of the `NamingService` defined in the `conf/jboss-service.xml` descriptor. The standard setting used in the JBoss distributions is `jboss:service=Naming`.
- **InvokerURL** or **InvokerURLPrefix** + **InvokerURLSuffix** + **UseHostName**. You can specify the full HTTP URL to the `InvokerServlet` using the `InvokerURL` attribute, or you can specify the hostname independent parts of the URL and have the `HttpProxyFactory` fill them in. An

example **InvokerURL** value would be

http://jboss-host1.dot.com:8080/invoker/JMXInvokerServlet. This can be broken down into:

- **InvokerURLPrefix**: the URL prefix prior to the hostname. Typically this will be **http://** or **https://** if SSL is to be used.
- **InvokerURLSuffix**: the URL suffix after the hostname. This will include the port number of the web server as well as the deployed path to the **InvokerServlet**. For the example **InvokerURL** value the **InvokerURLSuffix** would be **:8080/invoker/JMXInvokerServlet** without the quotes. The port number is determined by the web container service settings. The path to the **InvokerServlet** is specified in the **http-invoker.sar/invoker.war/WEB-INF/web.xml** descriptor.
- **UseHostName**: a flag indicating if the hostname should be used in place of the host IP address when building the hostname portion of the full **InvokerURL**. If true, **InetAddress.getLocalHost().getHostName** method will be used. Otherwise, the **InetAddress.getLocalHost().getHostAddress()** method is used.
- **ExportedInterface**: The **org.jnp.interfaces.Naming** interface the proxy will expose to clients. The actual client of this proxy is the JBoss JNDI implementation **NamingContext** class, which JNDI client obtain from **InitialContext** lookups when using the JBoss JNDI provider.
- **JndiName**: The name in JNDI under which the proxy is bound. This needs to be set to a blank/empty string to indicate the interface should not be bound into JNDI. We can't use the JNDI to bootstrap itself. This is the role of the **NamingFactoryServlet**.

The **http-invoker.sar/invoker.war/WEB-INF/web.xml** descriptor defines the mappings of the **NamingFactoryServlet** and **InvokerServlet** along with their initialization parameters. The configuration of the **NamingFactoryServlet** relevant to JNDI/HTTP is the **JNDIFactory** entry which defines:

- A **namingProxyMBean** initialization parameter that maps to the **HttpProxyFactory** MBean name. This is used by the **NamingFactoryServlet** to obtain the **Naming** proxy which it will return in response to HTTP posts. For the default **http-invoker.sar/META-INF/jboss-service.xml** settings the name **jboss:service=invoker, type=http, target=Naming**.
- A proxy initialization parameter that defines the name of the **namingProxyMBean** attribute to query for the Naming proxy value. This defaults to an attribute name of **Proxy**.
- The servlet mapping for the **JNDIFactory** configuration. The default setting for the unsecured mapping is **/JNDIFactory/***. This is relative to the context root of the **http-invoker.sar/invoker.war**, which by default is the WAR name minus the **.war** suffix.

The configuration of the **InvokerServlet** relevant to JNDI/HTTP is the **JMXInvokerServlet** which defines:

- The servlet mapping of the **InvokerServlet**. The default setting for the unsecured mapping is **/JMXInvokerServlet/***. This is relative to the context root of the **http-invoker.sar/invoker.war**, which by default is the WAR name minus the **.war** suffix.

6.4.2. Accessing JNDI over HTTPS

To be able to access JNDI over HTTP/SSL you need to enable an SSL connector on the web container.

The details of this are covered in the Integrating Servlet Containers for Tomcat. We will demonstrate the use of HTTPS with a simple example client that uses an HTTPS URL as the JNDI provider URL. We will provide an SSL connector configuration for the example, so unless you are interested in the details of the SSL connector setup, the example is self contained.

We also provide a configuration of the **HttpProxyFactory** setup to use an HTTPS URL. The following example shows the section of the **http-invoker.sarjboss-service.xml** descriptor that the example installs to provide this configuration. All that has changed relative to the standard HTTP configuration are the **InvokerURLPrefix** and **InvokerURLSuffix** attributes, which setup an HTTPS URL using the 8443 port.

```
<!-- Expose the Naming service interface via HTTPS -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
      name="jboss:service=invoker,type=https,target=Naming">
  <!-- The Naming service we are proxying -->
  <attribute name="InvokerName">jboss:service=Naming</attribute>
  <!-- Compose the invoker URL from the cluster node address -->
  <attribute name="InvokerURLPrefix">https://</attribute>
  <attribute name="InvokerURLSuffix">:8443/invoker/JMXInvokerServlet
</attribute>
  <attribute name="UseHostName">true</attribute>
  <attribute name="ExportedInterface">org.jnp.interfaces.Naming
</attribute>
  <attribute name="JndiName"/>
  <attribute name="ClientInterceptors">
    <interceptors>
      <interceptor>org.jboss.proxy.ClientMethodInterceptor
</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor
</interceptor>

<interceptor>org.jboss.naming.interceptors.ExceptionInterceptor
</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor
</interceptor>
    </interceptors>
  </attribute>
</mbean>
```

At a minimum, a JNDI client using HTTPS requires setting up a HTTPS URL protocol handler. We will be using the Java Secure Socket Extension (JSSE) for HTTPS. The JSSE documentation does a good job of describing what is necessary to use HTTPS, and the following steps were needed to configure the example client shown in [Example 6.2, "A JNDI client that uses HTTPS as the transport"](#):

- A protocol handler for HTTPS URLs must be made available to Java. The JSSE release includes an HTTPS handler in the **com.sun.net.ssl.internal.www.protocol** package. To enable the use of HTTPS URLs you include this package in the standard URL protocol handler search property, **java.protocol.handler.pkgs**. We set the **java.protocol.handler.pkgs** property in the Ant script.
- The JSSE security provider must be installed in order for SSL to work. This can be done either by installing the JSSE jars as an extension package, or programmatically. We use the programatic approach in the example since this is less intrusive. Line 18 of the **ExClient** code demonstrates how this is done.
- The JNDI provider URL must use HTTPS as the protocol. Lines 24-25 of the **ExClient** code

specify an HTTP/SSL connection to the localhost on port 8443. The hostname and port are defined by the web container SSL connector.

- The validation of the HTTPS URL hostname against the server certificate must be disabled. By default, the JSSE HTTPS protocol handler employs a strict validation of the hostname portion of the HTTPS URL against the common name of the server certificate. This is the same check done by web browsers when you connect to secured web site. We are using a self-signed server certificate that uses a common name of "**Chapter 8 SSL Example**" rather than a particular hostname, and this is likely to be common in development environments or intranets. The JBoss **HttpInvokerProxy** will override the default hostname checking if a **org.jboss.security.ignoreHttpsHost** system property exists and has a value of true. We set the **org.jboss.security.ignoreHttpsHost** property to true in the Ant script.

Example 6.2. A JNDI client that uses HTTPS as the transport

```
package org.jboss.chap3.ex1;

import java.security.Security;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Properties env = new Properties();
        env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                        "org.jboss.naming.HttpNamingContextFactory");
        env.setProperty(Context.PROVIDER_URL,
                        "https://localhost:8443/invoker/JNDIFactorySSL");

        Context ctx = new InitialContext(env);
        System.out.println("Created InitialContext, env=" + env);

        Object data = ctx.lookup("jmx/invoker/RMIAdaptor");
        System.out.println("lookup(jmx/invoker/RMIAdaptor): " + data);
    }
}
```

To test the client, first build the chapter 3 example to create the **chap3** configuration fileset.

```
[examples]$ ant -Dchap=naming config
```

Next, start the JBoss server using the **naming** configuration fileset:

```
[bin]$ sh run.sh -c naming
```

And finally, run the **ExClient** using:

```
[examples]$ ant -Dchap=naming -Dex=1 run-example
...
```

```
run-example1:
```

```
[java] Created InitialContext, env={java.naming. \
provider.url=https://localhost:8443/invoker/JNDIFactorySSL, java.naming. \
factory.initial=org.jboss.naming.HttpNamingContextFactory}
    [java] lookup(jmx/invoker/RMIAdaptor): org.jboss.invocation.jrmp. \
    interfaces.JRMPInvokerP
roxy@cac3fa
```

6.4.3. Securing Access to JNDI over HTTP

One benefit to accessing JNDI over HTTP is that it is easy to secure access to the JNDI **InitialContext** factory as well as the naming operations using standard web declarative security. This is possible because the server side handling of the JNDI/HTTP transport is implemented with two servlets. These servlets are included in the **http-invoker.sar/invoker.war** directory found in the **default** and **all** configuration deploy directories as shown previously. To enable secured access to JNDI you need to edit the **invoker.war/WEB-INF/web.xml** descriptor and remove all unsecured servlet mappings. For example, the **web.xml** descriptor shown in [Example 6.3, “An example web.xml descriptor for secured access to the JNDI servlets”](#) only allows access to the **invoker.war** servlets if the user has been authenticated and has a role of **HttpInvoker**.

Example 6.3. An example web.xml descriptor for secured access to the JNDI servlets

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <!-- ### Servlets -->
  <servlet>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <servlet-class>
      org.jboss.invocation.http.servlet.InvokerServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
    <servlet-name>JNDIFactory</servlet-name>
    <servlet-class>
      org.jboss.invocation.http.servlet.NamingFactoryServlet
    </servlet-class>
    <init-param>
      <param-name>namingProxyMBean</param-name>
      <param-
value>jboss:service=invoker,type=http,target=Naming</param-value>
    </init-param>
    <init-param>
      <param-name>proxyAttribute</param-name>
      <param-value>Proxy</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
  <!-- ### Servlet Mappings -->
  <servlet-mapping>
    <servlet-name>JNDIFactory</servlet-name>
    <url-pattern>/restricted/JNDIFactory/*</url-pattern>
```

```

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

```

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

```

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

```

The **security-domain** element defines the name of the security domain that will be used for the JAAS login module configuration used for authentication and authorization. See [Section 10.1.6, “Enabling Declarative Security in JBoss”](#) for additional details on the meaning and configuration of the security domain name.

6.4.4. Securing Access to JNDI with a Read-Only Unsecured Context

Another feature available for the JNDI/HTTP naming service is the ability to define a context that can be accessed by unauthenticated users in read-only mode. This can be important for services used by the authentication layer. For example, the **SRPLoginModule** needs to lookup the SRP server interface used to perform authentication. We'll now walk through how read-only JNDI works in JBoss.

First, the **ReadOnlyJNDIFactory** is declared in **invoker.sar/WEB-INF/web.xml**. It will be mapped to **/invoker/ReadOnlyJNDIFactory**.

```

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

```

```

        <description>A servlet that exposes the JBoss JNDI Naming service stub
            through http, but only for a single read-only context. The
return content
            is serialized MarshalledValue containing the
org.jnp.interfaces.Naming
            stub.
        </description>
        <servlet-
class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-
class>
        <init-param>
            <param-name>namingProxyMBean</param-name>
            <param-
value>jboss:service=invoker, type=http, target=Naming, readonly=true</param-
value>
        </init-param>
        <init-param>
            <param-name>proxyAttribute</param-name>
            <param-value>Proxy</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <!-- ... -->

    <servlet-mapping>
        <servlet-name>ReadOnlyJNDIFactory</servlet-name>
        <url-pattern>/ReadOnlyJNDIFactory/*</url-pattern>
    </servlet-mapping>

```

The factory only provides a JNDI stub which needs to be connected to an invoker. Here the invoker is **jboss:service=invoker, type=http, target=Naming, readonly=true**. This invoker is declared in the **http-invoker.sar/META-INF/jboss-service.xml** file.

```

    <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
        name="jboss:service=invoker, type=http, target=Naming, readonly=true">
        <attribute name="InvokerName">jboss:service=Naming</attribute>
        <attribute name="InvokerURLPrefix">http://</attribute>
        <attribute
name="InvokerURLSuffix">:8080/invoker/readonly/JMXInvokerServlet</attribut
e>
        <attribute name="UseHostName">true</attribute>
        <attribute
name="ExportedInterface">org.jnp.interfaces.Naming</attribute>
        <attribute name="JndiName"></attribute>
        <attribute name="ClientInterceptors">
            <interceptors>

<interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.naming.interceptors.ExceptionInterceptor</intercept
or>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>

```

```

        </interceptors>
    </attribute>
</mbean>

```

The proxy on the client side needs to talk back to a specific invoker servlet on the server side. The configuration here has the actual invocations going to **/invoker/readonly/JMXInvokerServlet**. This is actually the standard **JMXInvokerServlet** with a read-only filter attached.

```

    <filter>
        <filter-name>ReadOnlyAccessFilter</filter-name>
        <filter-
class>org.jboss.invocation.http.servlet.ReadOnlyAccessFilter</filter-
class>
        <init-param>
            <param-name>readOnlyContext</param-name>
            <param-value>readonly</param-value>
            <description>The top level JNDI context the filter will
enforce
                        read-only access on. If specified only Context.lookup
operations
                        will be allowed on this context. Another other operations
or
                        lookups on any other context will fail. Do not associate
this
                        filter with the JMXInvokerServlets if you want
unrestricted
                        access. </description>
        </init-param>
        <init-param>
            <param-name>invokerName</param-name>
            <param-value>jboss:service=Naming</param-value>
            <description>The JMX ObjectName of the naming service mbean
</description>
        </init-param>
    </filter>

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

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

```

The **readOnlyContext** parameter is set to **readonly** which means that when you access JBoss through the **ReadOnlyJNDIFactory**, you will only be able to access data in the **readonly** context. Here is a code fragment that illustrates the usage:

```

Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
    "http://localhost:8080/invoker/ReadOnlyJNDIFactory");

Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly/data");

```

Attempts to look up any objects outside of the readonly context will fail. Note that JBoss doesn't ship with any data in the **readonly** context, so the readonly context won't be bound usable unless you create it.

6.5. ADDITIONAL NAMING MBEANS

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

6.5.1. JNDI Binding Manager

The JNDI binding manager service allows you to quickly bind objects into JNDI for use by application code. The MBean class for the binding service is **org.jboss.naming.JNDIBindingServiceMgr**. It has a single attribute, **BindingsConfig**, which accepts an XML document that conforms to the **jndi-binding-service_1_0.xsd** schema. The content of the **BindingsConfig** attribute is unmarshalled using the JBossXB framework. The following is an MBean definition that shows the most basic form usage of the JNDI binding manager service.

```

<mbean code="org.jboss.naming.JNDIBindingServiceMgr"
      name="jboss.tests:name=example1">
  <attribute name="BindingsConfig" serialDataType="jbx">
    <jndi:bindings xmlns:xs="http://www.w3.org/2001/XMLSchema-
instance"
                  xmlns:jndi="urn:jboss:jndi-binding-service:1.0"
                  xs:schemaLocation="urn:jboss:jndi-binding-service
\
      resource:jndi-binding-service_1_0.xsd">
      <jndi:binding name="bindexample/message">
        <jndi:value trim="true">
          Hello, JNDI!
        </jndi:value>
      </jndi:binding>
    </jndi:bindings>
  </attribute>
</mbean>

```

This binds the text string **"Hello, JNDI!"** under the JNDI name **bindexample/message**. An application would look up the value just as it would for any other JNDI value. The **trim** attribute specifies that leading and trailing whitespace should be ignored. The use of the attribute here is purely for illustrative purposes as the default value is true.

```

InitialContext ctx = new InitialContext();
String          text = (String) ctx.lookup("bindexample/message");

```

String values themselves are not that interesting. If a JavaBeans property editor is available, the desired class name can be specified using the **type** attribute

```
<jndi:binding name="urls/jboss-home">
  <jndi:value type="java.net.URL">http://www.jboss.org</jndi:value>
</jndi:binding>
```

The **editor** attribute can be used to specify a particular property editor to use.

```
<jndi:binding name="hosts/localhost">
  <jndi:value editor="org.jboss.util.propertyeditor.InetAddressEditor">
    127.0.0.1
  </jndi:value>
</jndi:binding>
```

For more complicated structures, any JBossXB-ready schema may be used. The following example shows how a **java.util.Properties** object would be mapped.

```
<jndi:binding name="maps/testProps">
  <java:properties xmlns:java="urn:jboss:java-properties"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="urn:jboss:java-properties \
resource:java-properties_1_0.xsd">
    <java:property>
      <java:key>key1</java:key>
      <java:value>value1</java:value>
    </java:property>
    <java:property>
      <java:key>key2</java:key>
      <java:value>value2</java:value>
    </java:property>
  </java:properties>
</jndi:binding>
```

For more information on JBossXB, see the [JBossXB wiki page](#).

6.5.2. 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-service.xml** 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 a dot (.), the name is relative to the context in which the link is bound.

The following example provides a mapping of the JNDI name **QueueConnectionFactory** to the name **ConnectionFactory**.

```
<mbean code="org.jboss.naming.NamingAlias"
```

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

6.5.3. 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-service.xml** 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 can 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**: The **Properties** attribute contains the JNDI properties for the external **InitialContext**. The input should be the text equivalent to what would go into a **jndi.properties** file.
- **PropertiesURL**: This set the **jndi.properties** information for the external **InitialContext** from an external properties file. 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 MBean definition below shows a binding to an external LDAP context into the JBoss JNDI namespace under the name **external/ldap/jboss**.

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"

name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/jboss">
  <attribute name="JndiName">external/ldap/jboss</attribute>
  <attribute name="Properties">
    java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
    java.naming.provider.url=ldap://ldaphost.jboss.org:389/o=jboss.org
    java.naming.security.principal=cn=Directory Manager
    java.naming.security.authentication=simple
    java.naming.security.credentials=secret
  </attribute>
  <attribute name="InitialContext"> javax.naming.ldap.InitialLdapContext
</attribute>
  <attribute name="RemoteAccess">true</attribute>
</mbean>
```

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

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

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

```
<!-- Bind the /usr/local file system directory -->
<mbean code="org.jboss.naming.ExternalContext"

name="jboss.jndi:service=ExternalContext,jndiName=external/fs/usr/local">
  <attribute name="JndiName">external/fs/usr/local</attribute>
  <attribute name="Properties">

    java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
      java.naming.provider.url=file:///usr/local
    </attribute>
    <attribute
name="InitialContext">javax.naming.IntialContext</attribute>
  </mbean>
```

This configuration describes binding a local file system directory **/usr/local** into the JBoss JNDI namespace under the name **external/fs/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");
```

Note that the use the Sun JNDI service providers, which must be downloaded from <http://java.sun.com/products/jndi/serviceproviders.html>. The provider JARs should be placed in the server configuration **lib** directory.

6.5.4. 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. To view the JBoss JNDI namespace using the JNDIView MBean, you connect to the JMX Agent View using the http interface. The default settings put this at **http://localhost:8080/jmx-console/**. On this page you will see a section that lists the registered MBeans sorted by domain. It should look something like that shown in [Figure 6.4, “The JMX Console view of the configured JBoss MBeans”](#).

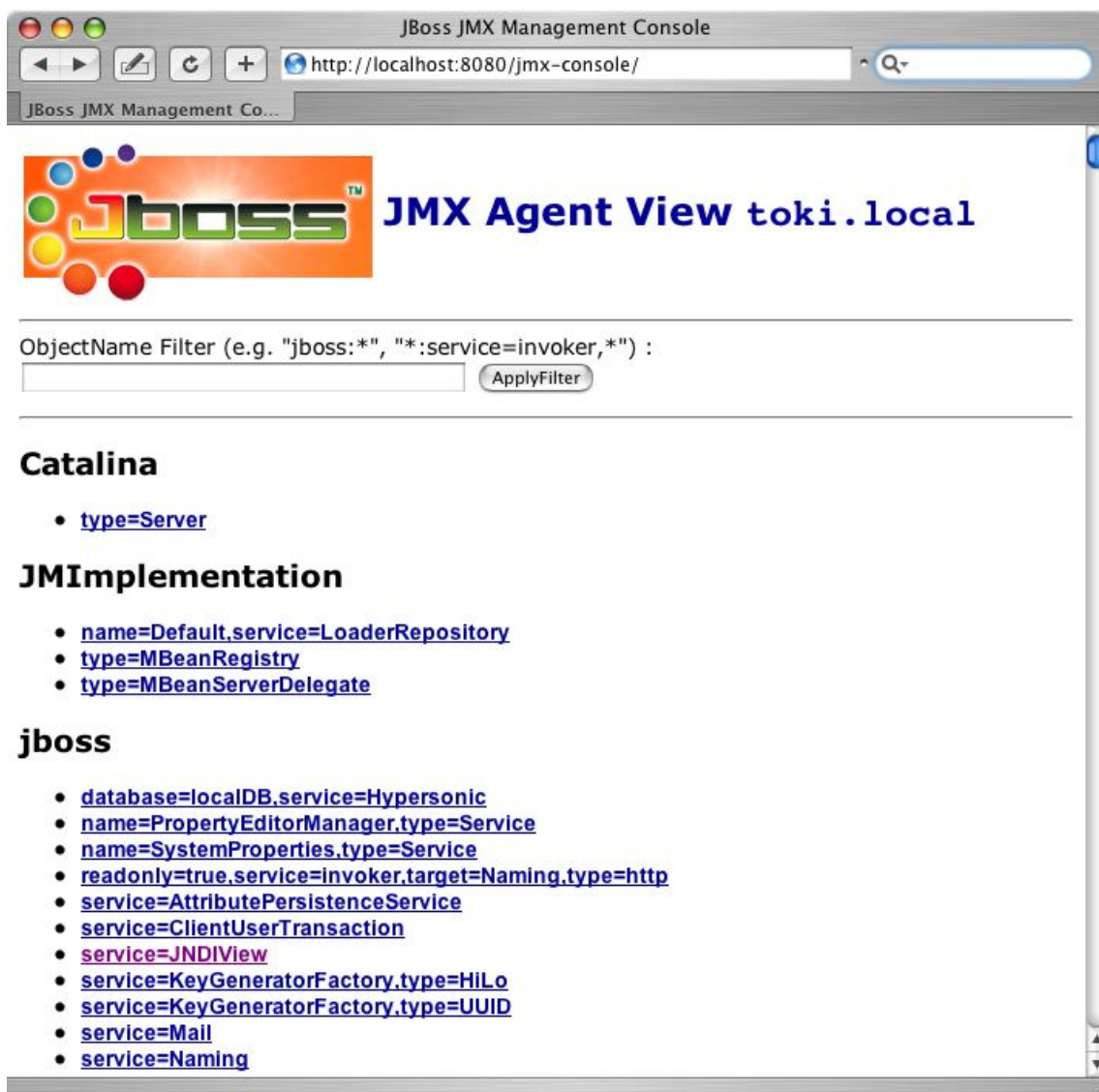


Figure 6.4. The JMX Console view of the configured JBoss MBeans

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 6.5, “The JMX Console view of the JNDIView MBean”](#).

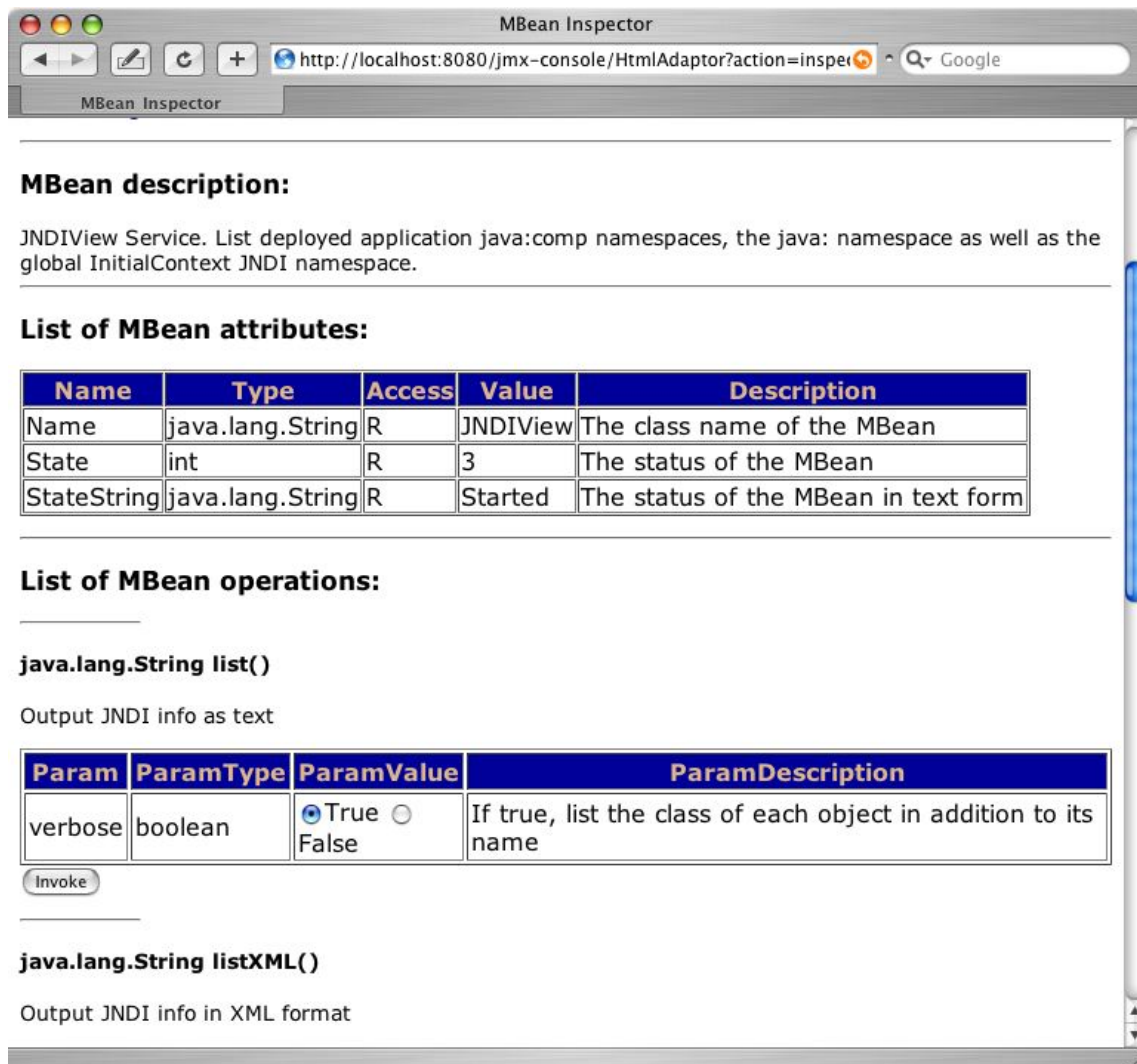


Figure 6.5. The JMX Console view of the JNDIView MBean

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 produces the view shown in [Figure 6.6, “The JMX Console view of the JNDIView list operation output”](#).

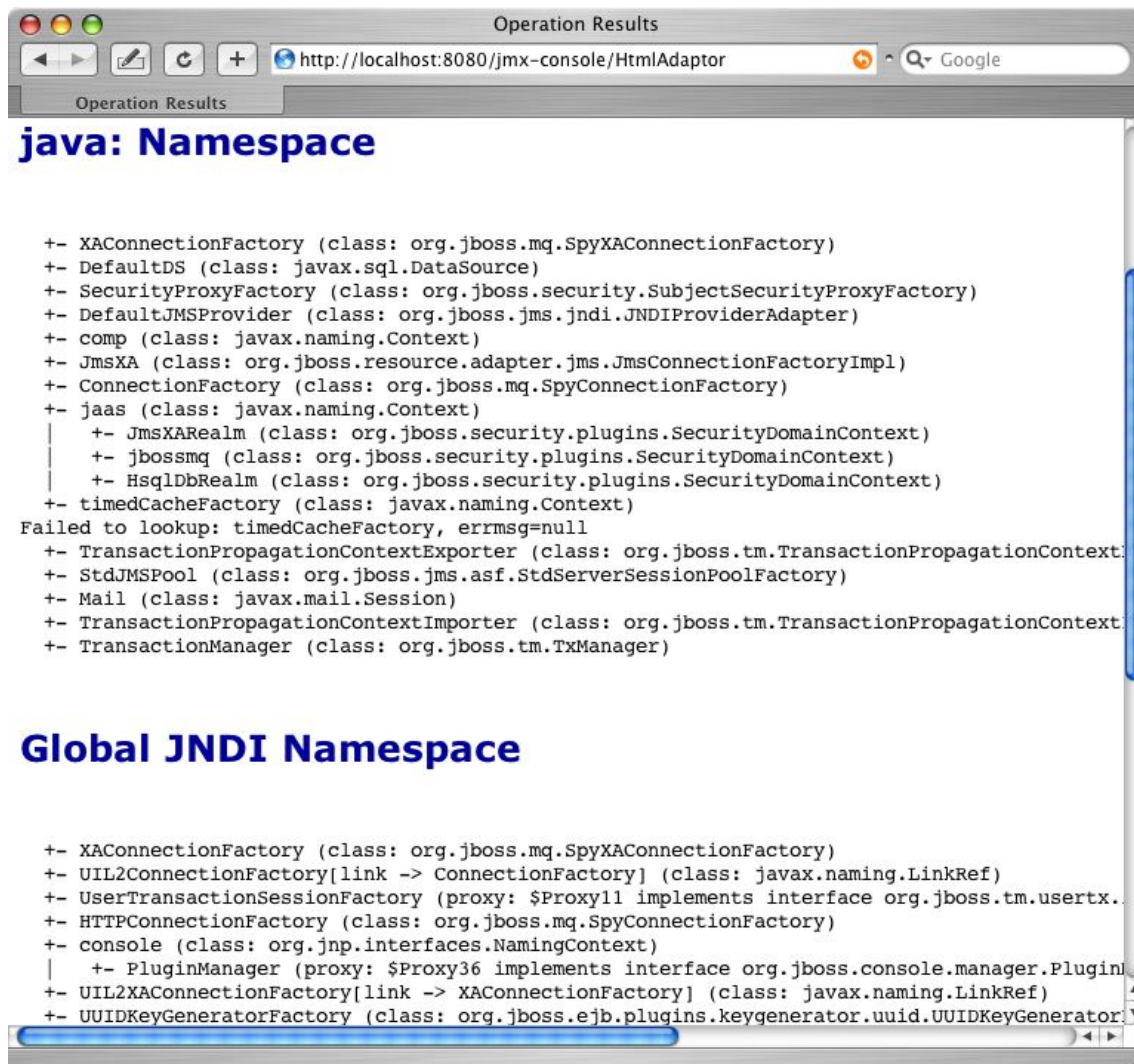


Figure 6.6. The JMX Console view of the JNDIView list operation output

6.6. 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 referred to as the ENC, the enterprise naming context. 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.

- 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.
- 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.
- The component deployer utilizes the container tools to ready a component for final deployment.
- 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 1.4 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. [Example 6.4, "ENC access sample code"](#) illustrates the prototypical lines of code a component uses to access its ENC.

Example 6.4. 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 vice 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. Components **A** and **B**, for example, 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 JBoss: 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 that particular context. Subcontexts and object bindings directly under **java:** are only visible within the JBoss server virtual machine and not to remote clients. 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 6.2, "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 server where the associated database pool resides. On the other hand, an EJB home interface would be bound to a globally visible name that should be accessible by remote client.

6.6.1. 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 deploymentdescriptor 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.

6.6.1.1. 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 [Example 6.5, “An example ejb-jar.xml env-entry fragment”](#). There is no JBoss specific deployment descriptor element because an **env-entry** is a complete name and value specification. [Example 6.6, “ENC env-entry access code fragment”](#) shows a sample code fragment for accessing the **maxExemptions** and **taxRate** env-entry values declared in the deployment descriptor.

Example 6.5. An example ejb-jar.xml env-entry fragment

```
<!-- ... -->
<session>
  <ejb-name>ASessionBean</ejb-name>
```

```

<!-- ... -->
<env-entry>
  <description>The maximum number of tax exemptions allowed
</description>
  <env-entry-name>maxExemptions</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>15</env-entry-value>
</env-entry>
<env-entry>
  <description>The tax rate </description>
  <env-entry-name>taxRate</env-entry-name>
  <env-entry-type>java.lang.Float</env-entry-type>
  <env-entry-value>0.23</env-entry-value>
</env-entry>
</session>
<!-- ... -->

```

Example 6.6. 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");

```

6.6.1.2. 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 same EJB JAR 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. [Example 6.7, “An example ejb-jar.xml ejb-ref descriptor fragment”](#) 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 [Example 6.7, “An example ejb-jar.xml ejb-ref descriptor fragment”](#) is given in [Example 6.8, “ENC ejb-ref access code fragment”](#).

Example 6.7. An example ejb-jar.xml ejb-ref descriptor fragment

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

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

    <!--...-->

```

Example 6.8. 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");

```

6.6.1.3. EJB References with `jboss.xml` and `jboss-web.xml`

The JBoss specific `jboss.xml` EJB deployment descriptor affects EJB references in two ways. First, the `jndi-name` child element of the `session` and `entity` elements allows the user to specify the deployment JNDI name for the EJB home interface. In the absence of a `jboss.xml` specification of the `jndi-name` for an EJB, the home interface is bound under the `ejb-jar.xml` `ejb-name` value. For example, the session EJB with the `ejb-name` of `ShoppingCartBean` in [Example 6.7, “An example ejb-jar.xml ejb-ref descriptor fragment”](#) would have its home interface bound under the JNDI name `ShoppingCartBean` in the absence of a `jboss.xml` `jndi-name` specification.

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

The `jboss-web.xml` descriptor is used only to set the destination to which a Web application 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 `jndi-name` element that specifies the JNDI name of the EJB home interface in the deployment environment

[Example 6.9, “An example jboss.xml ejb-ref fragment”](#) 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`

Example 6.9. An example `jboss.xml` ejb-ref fragment

```

<!-- ... -->
<session>
    <ejb-name>ProductBeanUser</ejb-name>
    <ejb-ref>
        <ejb-ref-name>ejb/ProductHome</ejb-ref-name>

```

```

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

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

```

6.6.1.4. EJB Local References

EJB 2.0 added 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. [Example 6.10, “An example ejb-jar.xml ejb-local-ref descriptor fragment”](#) 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 [Example 6.10](#), “An example ejb-jar.xml ejb-local-ref descriptor fragment” is given in [Example 6.11](#), “ENC ejb-local-ref access code fragment”.

Example 6.10. An example ejb-jar.xml ejb-local-ref descriptor fragment

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

Example 6.11. 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");
```

6.6.1.5. 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 optional **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.

[Example 6.12, “A web.xml resource-ref descriptor fragment”](#) shows an example **web.xml** descriptor fragment that illustrates the **resource-ref** element usage. [Example 6.13, “ENC resource-ref access sample code fragment”](#) provides a code fragment that an application component would use to access the **DefaultMail** resource declared by the **resource-ref**.

Example 6.12. A web.xml resource-ref descriptor fragment

```
<web>
  <!-- ... -->
  <servlet>
    <servlet-name>AServlet</servlet-name>
    <!-- ... -->
  </servlet>
  <!-- ... -->
  <!-- JDBC DataSources (java:comp/env/jdbc) -->
  <resource-ref>
    <description>The default DS</description>
```

```

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

```

Example 6.13. 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");

```

6.6.1.6. 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
- A **res-url** element that specifies the URL string in the case of a **resource-ref** of type `java.net.URL`

Example 6.14, “A sample `jboss-web.xml` resource-ref descriptor fragment” provides a sample `jboss-web.xml` descriptor fragment that shows sample mappings of the **resource-ref** elements given in Example 6.12, “A `web.xml` resource-ref descriptor fragment”.

Example 6.14. A sample `jboss-web.xml` resource-ref descriptor fragment

```

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

```

6.6.1.7. Resource Environment References

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

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

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

[Example 6.15, “An example ejb-jar.xml resource-env-ref fragment”](#) provides an example **resource-env-ref** element declaration by a session bean. [Example 6.16, “ENC resource-env-ref access code fragment”](#) gives a code fragment that illustrates how to look up the **StockInfo** queue declared by the **resource-env-ref**.

Example 6.15. 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>

```

Example 6.16. 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");

```

6.6.1.8. 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

Example 6.17, “A sample jboss.xml resource-env-ref descriptor fragment” provides a sample **jboss.xml** descriptor fragment that shows a sample mapping for the **StockInforesource-env-ref**.

Example 6.17. A sample jboss.xml resource-env-ref descriptor fragment

```

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

```

CHAPTER 7. CONNECTORS ON JBOSS

The JCA Configuration and Architecture

This chapter discusses the JBoss server implementation of the J2EE Connector Architecture (JCA). JCA is a resource manager integration API whose goal is to standardize access to non-relational resources in the same way the JDBC API standardized access to relational data. The purpose of this chapter is to introduce the utility of the JCA APIs and then describe the architecture of JCA in JBoss

7.1. JCA OVERVIEW

J2EE 1.4 contains a connector architecture (JCA) specification that allows for the integration of transacted and secure resource adaptors into a J2EE application server environment. The JCA specification describes the notion of such resource managers as Enterprise Information Systems (EIS). Examples of EIS systems include enterprise resource planning packages, mainframe transaction processing, non-Java legacy applications, etc.

The reason for focusing on EIS is primarily because the notions of transactions, security, and scalability are requirements in enterprise software systems. However, the JCA is applicable to any resource that needs to integrate into JBoss in a secure, scalable and transacted manner. In this introduction we will focus on resource adapters as a generic notion rather than something specific to the EIS environment.

The connector architecture defines a standard SPI (Service Provider Interface) for integrating the transaction, security and connection management facilities of an application server with those of a resource manager. The SPI defines the system level contract between the resource adaptor and the application server.

The connector architecture also defines a Common Client Interface (CCI) for accessing resources. The CCI is targeted at EIS development tools and other sophisticated users of integrated resources. The CCI provides a way to minimize the EIS specific code required by such tools. Typically J2EE developers will access a resource using such a tool, or a resource specific interface rather than using CCI directly. The reason is that the CCI is not a type specific API. To be used effectively it must be used in conjunction with metadata that describes how to map from the generic CCI API to the resource manager specific data types used internally by the resource manager.

The purpose of the connector architecture is to enable a resource vendor to provide a standard adaptor for its product. A resource adaptor is a system-level software driver that is used by a Java application to connect to resource. The resource adaptor plugs into an application server and provides connectivity between the resource manager, the application server, and the enterprise application. A resource vendor need only implement a JCA compliant adaptor once to allow use of the resource manager in any JCA capable application server.

An application server vendor extends its architecture once to support the connector architecture and is then assured of seamless connectivity to multiple resource managers. Likewise, a resource manager vendor provides one standard resource adaptor and it has the capability to plug in to any application server that supports the connector architecture.

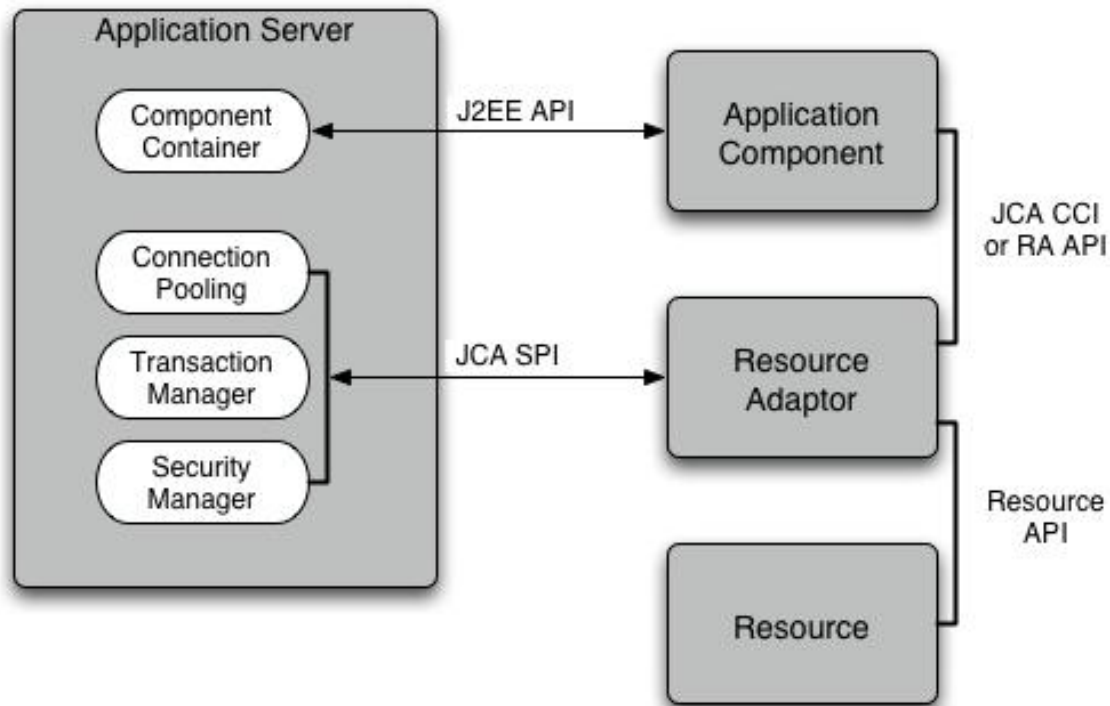


Figure 7.1. The relationship between a J2EE application server and a JCA resource adaptor

Figure 7.1, “The relationship between a J2EE application server and a JCA resource adaptor” 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 three-part system contract.

- **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 7.2, “The JCA 1.0 specification class diagram for the connection management architecture.” (from the JCA 1.5 specification) illustrates the relationship between the JCA architecture participants in terms of how they relate to the JCA SPI, CCI and JTA packages.

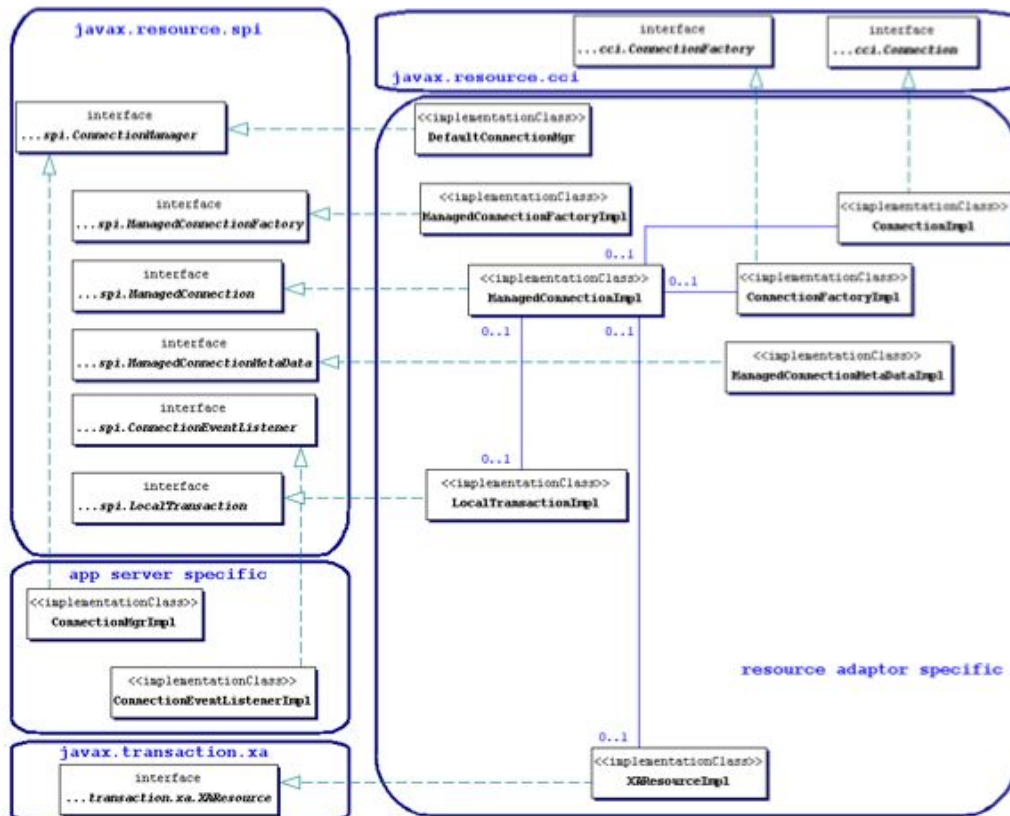


Figure 7.2. The JCA 1.0 specification class diagram for the connection management architecture.

The JBossCX architecture provides the implementation of the application server specific classes. Figure 7.2, “The JCA 1.0 specification class diagram for the connection management architecture.” shows that this comes down to the implementation of the `javax.resource.spi.ConnectionManager` and `javax.resource.spi.ConnectionEventListener` interfaces. The key aspects of this implementation are discussed in the following section on the JBossCX architecture.

7.2. AN OVERVIEW OF THE JBOSSCX ARCHITECTURE

The JBossCX framework provides the application server architecture extension required for the use of JCA resource adaptors. This is primarily a connection pooling and management extension along with a number of MBeans for loading resource adaptors into the JBoss server.

There are three coupled MBeans that make up a RAR deployment. These are the `org.jboss.resource.deployment.RARDeployment`, `org.jboss.resource.connectionmanager.RARDeployment`, and `org.jboss.resource.connectionmanager.BaseConnectionManager2`. The `org.jboss.resource.deployment.RARDeployment` is simply an encapsulation of the metadata of a RAR `META-INF/ra.xml` descriptor. It exposes this information as a DynamicMBean simply to make it available to the `org.jboss.resource.connectionmanager.RARDeployment` MBean.

The RARDeployer service handles the deployment of archives files containing resource adaptors (RARs). It creates the `org.jboss.resource.deployment.RARDeployment` MBeans when a RAR file is deployed. Deploying the RAR file is the first step in making the resource adaptor available to application components. For each deployed RAR, one or more connection factories must be configured and bound into JNDI. This task performed using a JBoss service descriptor that sets up a `org.jboss.resource.connectionmanager.BaseConnectionManager2` MBean implementation with a `org.jboss.resource.connectionmgr.RARDeployment` dependent.

7.2.1. BaseConnectionManager2 MBean

The `org.jboss.resource.connectionmanager.BaseConnectionManager2` MBean is a base class for the various types of connection managers required by the JCA spec. Subclasses include `NoTxConnectionManager`, `LocalTxConnectionManager` and `XATxConnectionManager`. These correspond to resource adaptors that support no transactions, local transaction and XA transaction respectively. You choose which subclass to use based on the type of transaction semantics you want, provided the JCA resource adaptor supports the corresponding transaction capability.

The common attributes supported by the `BaseConnectionManager2` MBean are:

- **ManagedConnectionPool:** This specifies the `ObjectName` of the MBean representing the pool for this connection manager. The MBean must have an `ManagedConnectionPool` attribute that is an implementation of the `org.jboss.resource.connectionmanager.ManagedConnectionPool` interface. Normally it will be an embedded MBean in a `depends` tag rather than an `ObjectName` reference to an existing MBean. The default MBean for use is the `org.jboss.resource.connectionmanager.JBossManagedConnectionPool`. Its configurable attributes are discussed below.
- **CachedConnectionManager:** This specifies the `ObjectName` of the `CachedConnectionManager` MBean implementation used by the connection manager. Normally this is specified using a `depends` tag with the `ObjectName` of the unique `CachedConnectionManager` for the server. The name `jboss.jca:service=CachedConnectionManager` is the standard setting to use.
- **SecurityDomainJndiName:** This specifies the JNDI name of the security domain to use for authentication and authorization of resource connections. This is typically of the form `java:/jaas/<domain>` where the `<domain>` value is the name of an entry in the `conf/login-config.xml` JAAS login module configuration file. This defines which JAAS login modules execute to perform authentication.
- **JaasSecurityManagerService:** This is the `ObjectName` of the security manager service. This should be set to the security manager MBean name as defined in the `conf/jboss-service.xml` descriptor, and currently this is `jboss.security:service=JaasSecurityManager`. This attribute will likely be removed in the future.

7.2.2. RARDeployment MBean

The `org.jboss.resource.connectionmanager.RARDeployment` MBean manages configuration and instantiation `ManagedConnectionFactory` instance. It does this using the resource adaptor metadata settings from the RAR `META-INF/ra.xml` descriptor along with the `RARDeployment` attributes. The configurable attributes are:

- **OldRarDeployment:** This is the `ObjectName` of the `org.jboss.resource.RarDeployment` MBean that contains the resource adaptor metadata. The form of this name is `jboss.jca:service=RARDeployment,name=<ra-display-name>` where the `<ra-display-name>` is the `ra.xml` descriptor `display-name` attribute value. The `RARDeployer` creates this when it deploys a RAR file. This attribute will likely be removed in the future.
- **ManagedConnectionFactoryProperties:** This is a collection of (name, type, value) triples that define attributes of the `ManagedConnectionFactory` instance. Therefore, the names of the attributes depend on the resource adaptor `ManagedConnectionFactory` instance. The following example shows the structure of the content of this attribute.

```

<properties>
  <config-property>
    <config-property-name>Attr0Name</config-property-name>
    <config-property-type>Attr0Type</config-property-type>
    <config-property-value>Attr0Value</config-property-value>
  </config-property>
  <config-property>
    <config-property-name>Attr1Name</config-property-name>
    <config-property-type>Attr2Type</config-property-type>
    <config-property-value>Attr2Value</config-property-value>
  </config-property>
  ...
</properties>

```

AttrXName is the Xth attribute name, **AttrXType** is the fully qualified Java type of the attribute, and **AttrXValue** is the string representation of the value. The conversion from string to **AttrXType** is done using the `java.beans.PropertyEditor` class for the **AttrXType**.

- **JndiName**: This is the JNDI name under which the resource adaptor will be made available. Clients of the resource adaptor use this name to obtain either the `javax.resource.cci.ConnectionFactory` or resource adaptor specific connection factory. The full JNDI name will be `java:/<JndiName>` meaning that the **JndiName** attribute value will be prefixed with `java:/`. This prevents use of the connection factory outside of the JBoss server VM. In the future this restriction may be configurable.

7.2.3. JBossManagedConnectionPool MBean

The `org.jboss.resource.connectionmanager.JBossManagedConnectionPool` MBean is a connection pooling MBean. It is typically used as the embedded MBean value of the `BaseConnectionManager2ManagedConnectionPool` attribute. When you setup a connection manager MBean you typically embed the pool configuration in the connection manager descriptor. The configurable attributes of the `JBossManagedConnectionPool` are:

- **ManagedConnectionFactoryName**: This specifies the **ObjectName** of the MBean that creates `javax.resource.spi.ManagedConnectionFactory` instances. Normally this is configured as an embedded MBean in a depends element rather than a separate MBean reference using the `RARDeployment` MBean. The MBean must provide an appropriate `startManagedConnectionFactory` operation.
- **MinSize**: This attribute indicates the minimum number of connections this pool should hold. These are not created until a **Subject** is known from a request for a connection. **MinSize** connections will be created for each sub-pool.
- **MaxSize**: This attribute indicates the maximum number of connections for a pool. No more than **MaxSize** connections will be created in each sub-pool.
- **BlockingTimeoutMillis**: This attribute indicates the maximum time to block while waiting for a connection before throwing an exception. Note that this blocks only while waiting for a permit for a connection, and will never throw an exception if creating a new connection takes an inordinately long time.
- **IdleTimeoutMinutes**: This attribute indicates the maximum time a connection may be idle before being closed. The actual maximum time depends also on the idle remover thread scan time, which is 1/2 the smallest idle timeout of any pool.

- **NoTxSeparatePools**: Setting this to true doubles the available pools. One pool is for connections used outside a transaction the other inside a transaction. The actual pools are lazily constructed on first use. This is only relevant when setting the pool parameters associated with the **LocalTxConnectionManager** and **XATxConnectionManager**. Its use case is for Oracle (and possibly other vendors) XA implementations that don't like using an XA connection with and without a JTA transaction.
- **Criteria**: This attribute indicates if the JAAS **javax.security.auth.Subject** from security domain associated with the connection, or app supplied parameters (such as from **getConnection(user, pw)**) are used to distinguish connections in the pool. The allowed values are:
 - **ByContainer**: use **Subject**
 - **ByApplication**: use application supplied parameters only
 - **ByContainerAndApplication**: use both
 - **ByNothing**: all connections are equivalent, usually if adapter supports reauthentication

7.2.4. CachedConnectionManager MBean

The **org.jboss.resource.connectionmanager.CachedConnectionManager** MBean manages associations between meta-aware objects (those accessed through interceptor chains) and connection handles, as well as between user transactions and connection handles. Normally there should only be one such MBean, and this is configured in the core **jboss-service.xml** descriptor. It is used by **CachedConnectionInterceptor**, JTA **UserTransaction** implementation and all **BaseConnectionManager2** instances. The configurable attributes of the **CachedConnectionManager** MBean are:

- **SpecCompliant**: Enable this boolean attribute for spec compliant non-shareable connections reconnect processing. This allows a connection to be opened in one call and used in another. Note that specifying this behavior disables connection close processing.
- **Debug**: Enable this boolean property for connection close processing. At the completion of an EJB method invocation, unclosed connections are registered with a transaction synchronization. If the transaction ends without the connection being closed, an error is reported and JBoss closes the connection. This is a development feature that should be turned off in production for optimal performance.
- **TransactionManagerServiceName**: This attribute specifies the JMX **ObjectName** of the JTA transaction manager service. Connection close processing is now synchronized with the transaction manager and this attribute specifies the transaction manager to use.

7.2.5. A Sample Skeleton JCA Resource Adaptor

To conclude our discussion of the JBoss JCA framework we will create and deploy a single non-transacted resource adaptor that simply provides a skeleton implementation that stubs out the required interfaces and logs all method calls. We will not discuss the details of the requirements of a resource adaptor provider as these are discussed in detail in the JCA specification. The purpose of the adaptor is to demonstrate the steps required to create and deploy a RAR in JBoss, and to see how JBoss interacts with the adaptor.

The adaptor we will create could be used as the starting point for a non-transacted file system adaptor. The source to the example adaptor can be found in the **src/main/org/jboss/book/jca/ex1**

directory of the book examples. A class diagram that shows the mapping from the required `javax.resource.spi` interfaces to the resource adaptor implementation is given in Figure 7.3, “The file system RAR class diagram”.

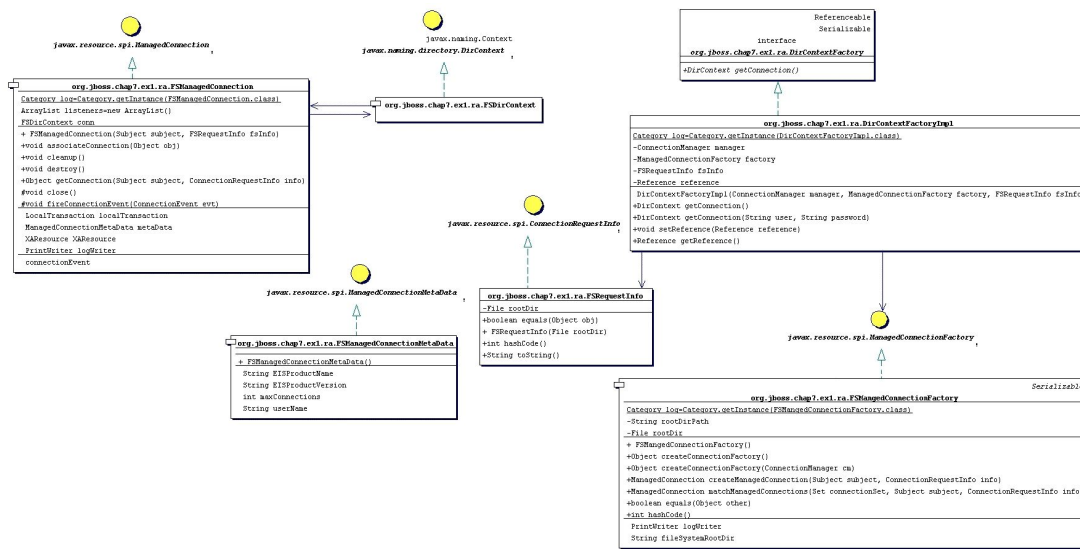


Figure 7.3. The file system RAR class diagram

We will build the adaptor, deploy it to the JBoss server and then run an example client against an EJB that uses the resource adaptor to demonstrate the basic steps in a complete context. We'll then take a look at the JBoss server log to see how the JBoss JCA framework interacts with the resource adaptor to help you better understand the components in the JCA system level contract.

To build the example and deploy the RAR to the JBoss server **deploy/lib** directory, execute the following Ant command in the book examples directory.

```
[examples]$ ant -Dchap=jca build-chap
```

The deployed files include a **jca-ex1.sar** and a **notxfis-service.xml** service descriptor. The example resource adaptor deployment descriptor is shown in Example 7.1, “The nontransactional file system resource adaptor deployment descriptor”.

Example 7.1. The nontransactional file system resource adaptor deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-
J2EE_Certification_and_Standards_Compliance"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd" version="1.5">
  <display-name>File System Adapter</display-name>
  <vendor-name>JBoss</vendor-name>
  <eis-type>FileSystem</eis-type>
  <resourceadapter-version>1.0</resourceadapter-version>
  <license>
    <description>LGPL</description>
    <license-required>false</license-required>
  </license>
  <resourceadapter>
    <resourceadapter-class>
```

```

        org.jboss.resource.deployment.DummyResourceAdapter
    </resourceadapter-class>
    <outbound-resourceadapter>
        <connection-definition>
            <managedconnectionfactory-class>
org.jboss.book.jca.ex1.ra.FSManagedConnectionFactory
        </managedconnectionfactory-class>
            <config-property>
                <config-property-name>FileSystemRootDir</config-
property-name>
                <config-property-type>java.lang.String</config-
property-type>
                <config-property-value>/tmp/db/fs_store</config-
property-value>
            </config-property>
            <config-property>
                <config-property-name>UserName</config-property-
name>
                <config-property-type>java.lang.String</config-
property-type>
                <config-property-value/>
            </config-property>
            <config-property>
                <config-property-name>Password</config-property-
name>
                <config-property-type>java.lang.String</config-
property-type>
                <config-property-value/>
            </config-property>
            <connectionfactory-interface>
org.jboss.book.jca.ex1.ra.DirContextFactory </connectionfactory-
interface> <connectionfactory-impl-class>
org.jboss.book.jca.ex1.ra.DirContextFactoryImpl </connectionfactory-
impl-class> <connection-interface> javax.naming.directory.DirContext
</connection-interface> <connection-impl-class>
org.jboss.book.jca.ex1.ra.FSDirContext </connection-impl-class>
        </connection-definition>
        <transaction-support>NoTransaction</transaction-support>
        <authentication-mechanism>
            <authentication-mechanism-
type>BasicPassword</authentication-mechanism-type>
            <credential-interface>
                javax.resource.spi.security.PasswordCredential
            </credential-interface>
        </authentication-mechanism>
        <reauthentication-support>true</reauthentication-support>
    </outbound-resourceadapter>
    <security-permission>
        <description> Read/Write access is required to the contents
of the
            FileSystemRootDir </description>
        <security-permission-spec> permission
java.io.FilePermission
            "/tmp/db/fs_store/*", "read,write";
        </security-permission-spec>

```

```

        </security-permission>
    </resourceadapter>
</connector>

```

The key items in the resource adaptor deployment descriptor are highlighted in bold. These define the classes of the resource adaptor, and the elements are:

- **managedconnectionfactory-class**: The implementation of the **ManagedConnectionFactory** interface, **org.jboss.book.jca.ex1.ra.FSManagedConnectionFactory**
- **connectionfactory-interface**: This is the interface that clients will obtain when they lookup the connection factory instance from JNDI, here a proprietary resource adaptor value, **org.jboss.book.jca.ex1.ra.DirContextFactory**. This value will be needed when we create the JBoss **ds.xml** to use the resource.
- **connectionfactory-impl-class**: This is the class that provides the implementation of the **connectionfactory-interface**, **org.jboss.book.jca.ex1.ra.DirContextFactoryImpl**.
- **connection-interface**: This is the interface for the connections returned by the resource adaptor connection factory, here the JNDI **javax.naming.directory.DirContext** interface.
- **connection-impl-class**: This is the class that provides the **connection-interface** implementation, **org.jboss.book.jca.ex1.ra.FSDirContext**.
- **transaction-support**: The level of transaction support, here defined as **NoTransaction**, meaning the file system resource adaptor does not do transactional work.

The RAR classes and deployment descriptor only define a resource adaptor. To use the resource adaptor it must be integrated into the JBoss application server using a **ds.xml** descriptor file. An example of this for the file system adaptor is shown in [Example 7.2, “The notxfs-ds.xml resource adaptor MBeans service descriptor.”](#).

Example 7.2. The notxfs-ds.xml resource adaptor MBeans service descriptor.

```

<!DOCTYPE connection-factories PUBLIC
    "-//JBoss//DTD JBOSS JCA Config 1.5//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">
<!--
    The non-transaction FileSystem resource adaptor service
    configuration
-->
<connection-factories>
    <no-tx-connection-factory>
        <jndi-name>NoTransFS</jndi-name>
        <rar-name>jca-ex1.rar</rar-name>
        <connection-definition>
            org.jboss.book.jca.ex1.ra.DirContextFactory
        </connection-definition>
        <config-property name="FileSystemRootDir"
            type="java.lang.String">/tmp/db/fs_store</config-property>
        </no-tx-connection-factory>
    </connection-factories>

```


The main attributes are:

- **jndi-name**: This specifies where the connection factory will be bound into JNDI. For this deployment that binding will be **java:/NoTransFS**.
- **rar-name**: This is the name of the RAR file that contains the definition for the resource we want to provide. For nested RAR files, the name would look like **myapplication.ear#my.rar**. In this example, it is simply **jca-ex1.rar**.
- **connection-definition**: This is the connection factory interface class. It should match the **connectionfactory-interface** in the **ra.xml** file. Here our connection factory interface is **org.jboss.book.jca.ex1.ra.DirContextFactory**.
- **config-property**: This can be used to provide non-default settings to the resource adaptor connection factory. Here the **FileSystemRootDir** is being set to **/tmp/db/fs_store**. This overrides the default value in the **ra.xml** file.

To deploy the RAR and connection manager configuration to the JBoss server, run the following:

```
[examples]$ ant -Dchap=jca config
```

The server console will display some logging output indicating that the resource adaptor has been deployed.

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 below.

Example 7.3. The stateless session bean echo method code that shows the access of the resource adaptor connection factory.

```
public String echo(String arg)
{
    log.info("echo, arg="+arg);
    try {
        InitialContext ctx = new InitialContext();
        Object          ref =
ctx.lookup("java:comp/env/ra/DirContextFactory");
        log.info("echo, ra/DirContextFactory=" + ref);

        DirContextFactory dcf = (DirContextFactory) ref;
        log.info("echo, found dcf=" + dcf);

        DirContext dc = dcf.getConnection();
        log.info("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]$ ant -Dchap=jca -Dex=1 run-example
```

You'll see some output from the bean in the system console, but much more detailed logging output can be found in the **server/production/log/server.log** file. Don't worry if you see exceptions. They are just stack traces to highlight the call path into parts of the adaptor. To help understand the interaction between the adaptor and the JBoss JCA layer, we'll summarize the events seen in the log using a sequence diagram. Figure 7.4, “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.” 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.

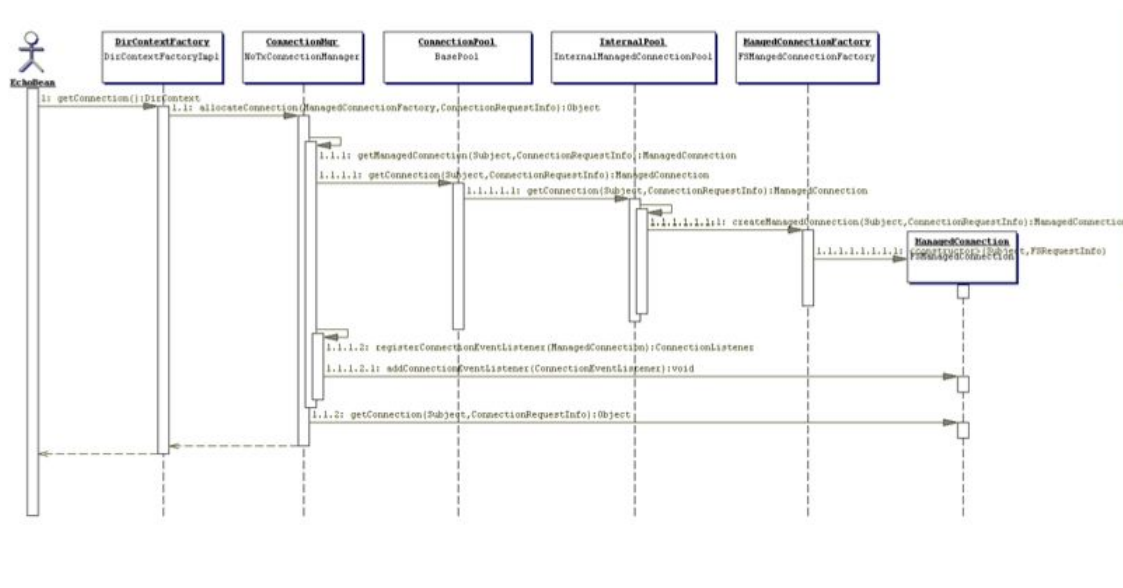


Figure 7.4. A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor that result when the EchoBean accesses the resource adaptor connection factory.

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, the echo method invokes the **getConnection** method on the resource adaptor connection factory obtained from the JNDI lookup on the **java:comp/env/ra/DirContextFactory** name which is a link to the **java:/NoTransFS** location.
- 1.1, the **DirContextFactoryImpl** class asks its associated **ConnectionManager** to allocate a connection. It passes in the **ManagedConnectionFactory** and **FSRequestInfo** that were associated with the **DirContextFactoryImpl** during its construction.
- 1.1.1, the **ConnectionManager** invokes its **getManagedConnection** method with the current **Subject** and **FSRequestInfo**.
- 1.1.1.1, the **ConnectionManager** asks its object pool for a connection object. The **JBossManagedConnectionPool\$BasePool** is get the key for the connection and then asks the matching **InternalPool** for a connection.
- 1.1.1.1.1, Since no connections have been created the pool must create a new connection. This is done by requesting a new managed connection from the **ManagedConnectionFactory**. The **Subject** associated with the pool as well as the **FSRequestInfo** data are passed as arguments to the **createManagedConnection** method invocation.
- 1.1.1.1.1.1, the **ConnectionFactory** creates a new **FManagedConnection** instance and passes in the **Subject** and **FSRequestInfo** data.
- 1.1.1.2, a **javax.resource.spi.ConnectionListener** instance is created. The type of listener created is based on the type of **ConnectionManager**. In this case it is an **org.jboss.resource.connectionmgr.BaseConnectionManager2\$NoTransactionListener** instance.
- 1.1.1.2.1, the listener registers as a **javax.resource.spi.ConnectionEventListener** with the **ManagedConnection** instance created in 1.2.1.1.
- 1.1.2, the **ManagedConnection** is asked for the underlying resource manager connection. The **Subject** and **FSRequestInfo** data are passed as arguments to the **getConnection** method invocation.
- The resulting connection object is cast to a **javax.naming.directory.DirContext** instance since this is the public interface defined by the resource adaptor.
- After the **EchoBean** has obtained the **DirContext** for the resource adaptor, it simply closes the connection to indicate its interaction with the resource manager is complete.

This concludes the resource adaptor example. Our investigation into the interaction between the JBossCX layer and a trivial resource adaptor should give you sufficient understanding of the steps required to configure any resource adaptor. The example adaptor can also serve as a starting point for the creation of your own custom resource adaptors if you need to integrate non-JDBC resources into the JBoss server environment.

7.3. CONFIGURING JDBC DATASOURCES

Rather than configuring the connection manager factory related MBeans discussed in the previous section via a mbean services deployment descriptor, JBoss provides a simplified datasource centric descriptor. This is transformed into the standard **jboss-service.xml** MBean services deployment descriptor using a XSL transform applied by the **org.jboss.deployment.XSLSubDeployer** included

in the **jboss-jca.sar** deployment. The simplified configuration descriptor is deployed the same as other deployable components. The descriptor must be named using a ***-ds.xml** pattern in order to be recognized by the **XSLSubDeployer**.

The schema for the top-level datasource elements of the ***-ds.xml** configuration deployment file is shown in [Figure 7.5, “The simplified JCA DataSource configuration descriptor top-level schema elements”](#).

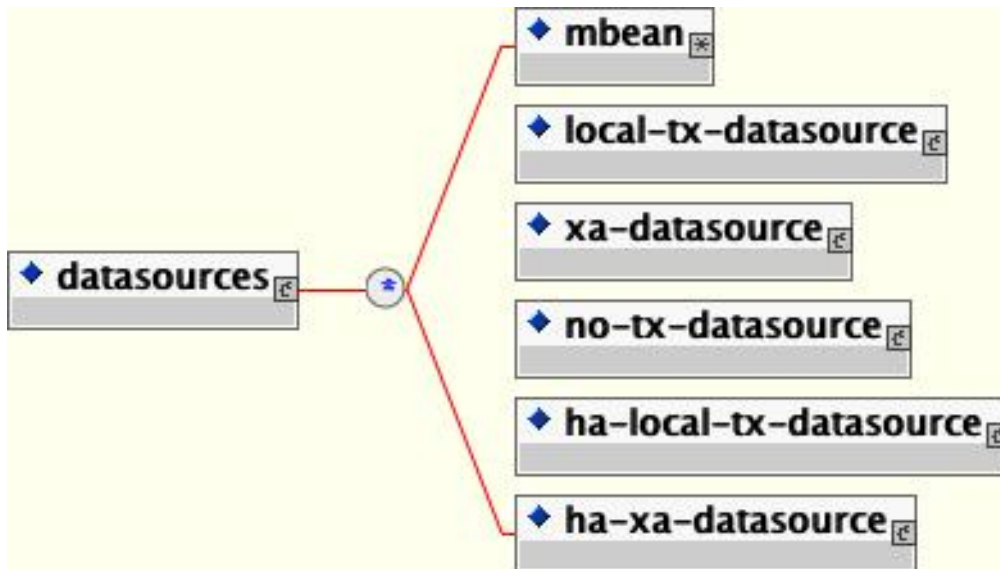


Figure 7.5. The simplified JCA DataSource configuration descriptor top-level schema elements

Multiple datasource configurations may be specified in a configuration deployment file. The child elements of the **datasources** root are:

- **mbean**: Any number **mbean** elements may be specified to define MBean services that should be included in the **jboss-service.xml** descriptor that results from the transformation. This may be used to configure services used by the datasources.
- **no-tx-datasource**: This element is used to specify the (**org.jboss.resource.connectionmanager**) **NoTxConnectionManager** service configuration. **NoTxConnectionManager** is a JCA connection manager with no transaction support. The **no-tx-datasource** child element schema is given in [Figure 7.6, “The non-transactional DataSource configuration schema”](#).
- **local-tx-datasource**: This element is used to specify the (**org.jboss.resource.connectionmanager**) **LocalTxConnectionManager** service configuration. **LocalTxConnectionManager** implements a **ConnectionEventListener** that implements **XAResource** to manage transactions through the transaction manager. To ensure that all work in a local transaction occurs over the same **ManagedConnection**, it includes a **xid** to **ManagedConnection** map. When a **Connection** is requested or a transaction started with a connection handle in use, it checks to see if a **ManagedConnection** already exists enrolled in the global transaction and uses it if found. Otherwise, a free **ManagedConnection** has its **LocalTransaction** started and is used. The **local-tx-datasource** child element schema is given in [Figure 7.7, “The non-XA DataSource configuration schema”](#)
- **xa-datasource**: This element is used to specify the (**org.jboss.resource.connectionmanager**) **XATxConnectionManager** service configuration. **XATxConnectionManager** implements a **ConnectionEventListener** that obtains the **XAResource** to manage transactions through the transaction manager from the

adaptor **ManagedConnection**. To ensure that all work in a local transaction occurs over the same **ManagedConnection**, it includes a xid to **ManagedConnection** map. When a **Connection** is requested or a transaction started with a connection handle in use, it checks to see if a **ManagedConnection** already exists enrolled in the global transaction and uses it if found. Otherwise, a free **ManagedConnection** has its **LocalTransaction** started and is used. The **xa-datasource** child element schema is given in [Figure 7.8, “The XA DataSource configuration schema”](#).

- **ha-local-tx-datasource**: This element is identical to **local-tx-datasource**, with the addition of the datasource failover capability allowing JBoss to failover to an alternate database in the event of a database failure.
- **ha-xa-datasource**: This element is identical to **xa-datasource**, with the addition of the datasource failover capability allowing JBoss to failover to an alternate database in the event of a database failure.



Figure 7.6. The non-transactional DataSource configuration schema

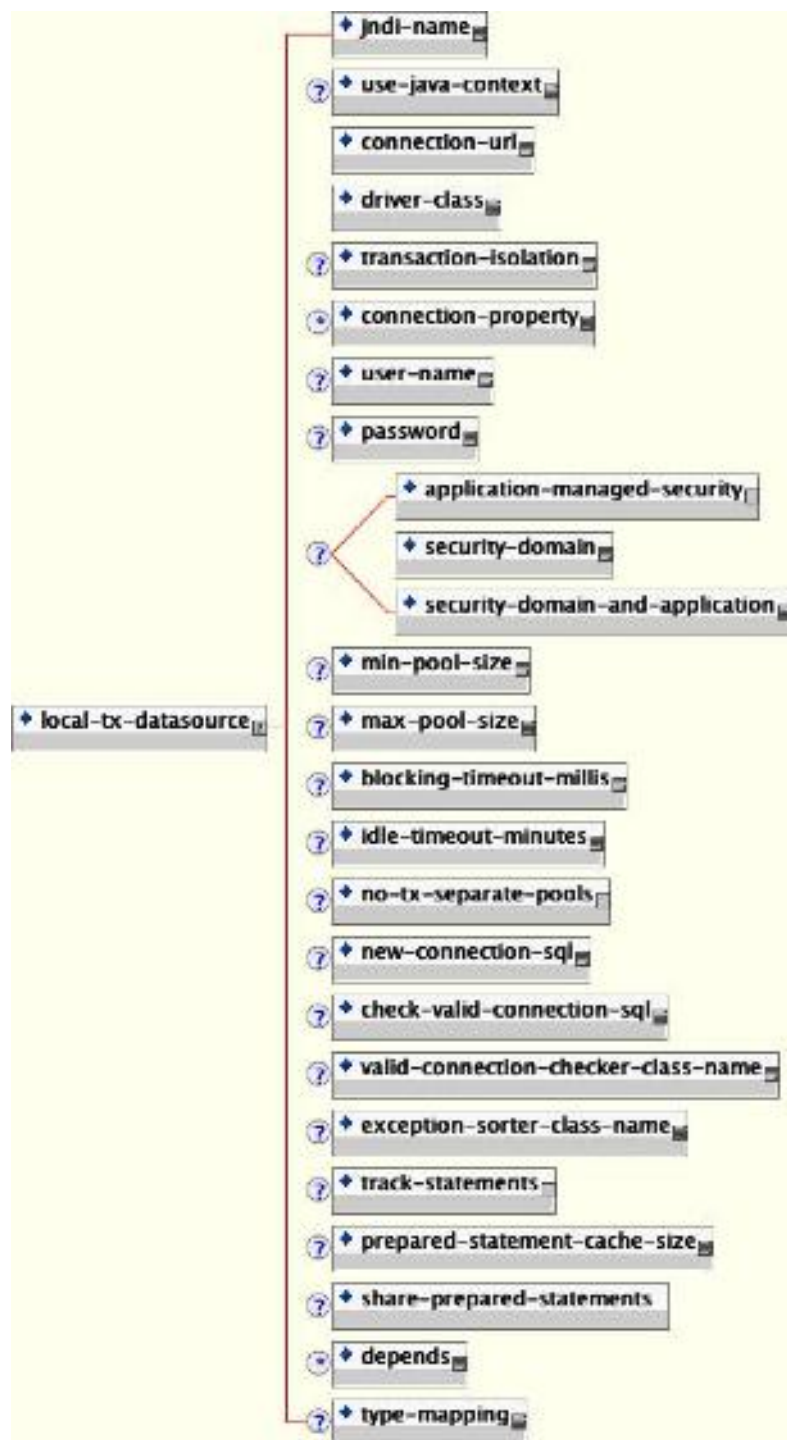


Figure 7.7. The non-XA DataSource configuration schema

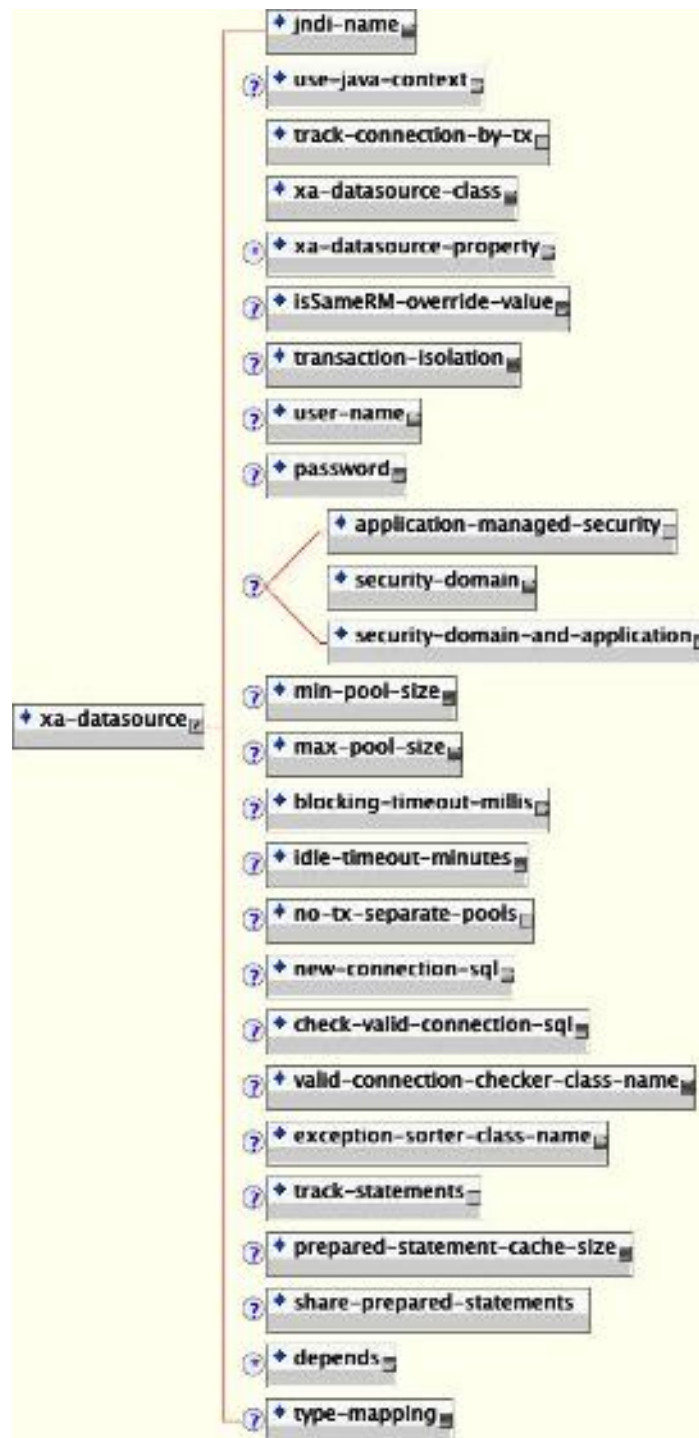


Figure 7.8. The XA DataSource configuration schema

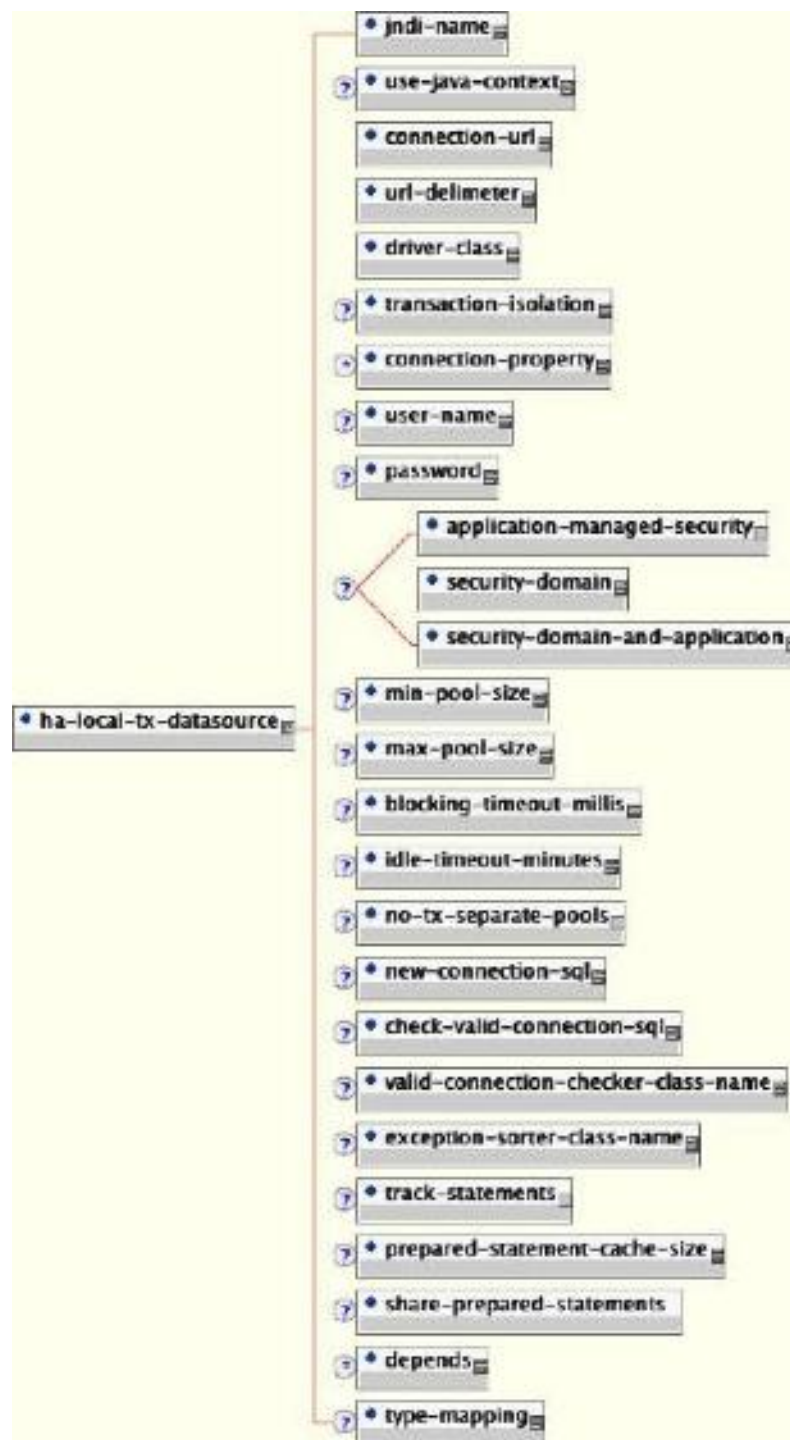


Figure 7.9. The schema for the experimental non-XA DataSource with failover



Figure 7.10. The schema for the experimental XA Datasource with failover

Elements that are common to all datasources include:

- **jndi-name**: The JNDI name under which the **DataSource** wrapper will be bound. Note that this name is relative to the **java:/** context, unless **use-java-context** is set to false. **DataSource** wrappers are not usable outside of the server VM, so they are normally bound under the **java:/**, which isn't shared outside the local VM.
- **use-java-context**: If this is set to false the the datasource will be bound in the global JNDI context rather than the **java:** context.
- **user-name**: This element specifies the default username used when creating a new connection. The actual username may be overridden by the application code **getConnection** parameters or the connection creation context JAAS Subject.

- **password**: This element specifies the default password used when creating a new connection. The actual password may be overridden by the application code **getConnection** parameters or the connection creation context JAAS Subject.
- **application-managed-security**: Specifying this element indicates that connections in the pool should be distinguished by application code supplied parameters, such as from **getConnection(user, pw)**.
- **security-domain**: Specifying this element indicates that connections in the pool should be distinguished by JAAS Subject based information. The content of the **security-domain** is the name of the JAAS security manager that will handle authentication. This name correlates to the JAAS **login-config.xml** descriptor **application-policy/name** attribute.
- **security-domain-and-application**: Specifying this element indicates that connections in the pool should be distinguished both by application code supplied parameters and JAAS Subject based information. The content of the **security-domain** is the name of the JAAS security manager that will handle authentication. This name correlates to the JAAS **login-config.xml** descriptor **application-policy/name** attribute.
- **min-pool-size**: This element specifies the minimum number of connections a pool should hold. These pool instances are not created until an initial request for a connection is made. This default to 0.
- **max-pool-size**: This element specifies the maximum number of connections for a pool. No more than the **max-pool-size** number of connections will be created in a pool. This defaults to 20.
- **blocking-timeout-millis**: This element specifies the maximum time in milliseconds to block while waiting for a connection before throwing an exception. Note that this blocks only while waiting for a permit for a connection, and will never throw an exception if creating a new connection takes an inordinately long time. The default is 30000.
- **idle-timeout-minutes**: This element specifies the maximum time in minutes a connection may be idle before being closed. The actual maximum time depends also on the **IdleRemover** scan time, which is 1/2 the smallest idle-timeout-minutes of any pool.
- **new-connection-sql**: This is a SQL statement that should be executed when a new connection is created. This can be used to configure a connection with database specific settings not configurable via connection properties.
- **check-valid-connection-sql**: This is a SQL statement that should be run on a connection before it is returned from the pool to test its validity to test for stale pool connections. An example statement could be: **select count(*) from x**.
- **exception-sorter-class-name**: This specifies a class that implements the **org.jboss.resource.adapter.jdbc.ExceptionSorter** interface to examine database exceptions to determine whether or not the exception indicates a connection error. Current implementations include:
 - **org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter**
 - **org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter**
 - **org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter**
 - **org.jboss.resource.adapter.jdbc.vendor.InformixExceptionSorter**

- **valid-connection-checker-class-name**: This specifies a class that implements the `org.jboss.resource.adapter.jdbc.ValidConnectionChecker` interface to provide a `SQLException isValidConnection(Connection c)` method that is called with a connection that is to be returned from the pool to test its validity. This overrides the **check-valid-connection-sql** when present. The only provided implementation is `org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker`.
- **track-statements**: This boolean element specifies whether to check for unclosed statements when a connection is returned to the pool. If true, a warning message is issued for each unclosed statement. If the log4j category `org.jboss.resource.adapter.jdbc.WrappedConnection` has trace level enabled, a stack trace of the connection close call is logged as well. This is a debug feature that can be turned off in production.
- **prepared-statement-cache-size**: This element specifies the number of prepared statements per connection in an LRU cache, which is keyed by the SQL query. Setting this to zero disables the cache.
- **depends**: The **depends** element specifies the JMX `ObjectName` string of a service that the connection manager services depend on. The connection manager service will not be started until the dependent services have been started.
- **type-mapping**: This element declares a default type mapping for this datasource. The type mapping should match a **type-mapping/name** element from `standardjbosscmp-jdbc.xml`.

Additional common child elements for both **no-tx-datasource** and **local-tx-datasource** include:

- **connection-url**: This is the JDBC driver connection URL string, for example, `jdbc:hsqldb:hsq1://localhost:1701`.
- **driver-class**: This is the fully qualified name of the JDBC driver class, for example, `org.hsqldb.jdbcDriver`.
- **connection-property**: The **connection-property** element allows you to pass in arbitrary connection properties to the `java.sql.Driver.connect(url, props)` method. Each **connection-property** specifies a string name/value pair with the property name coming from the name attribute and the value coming from the element content.

Elements in common to the **local-tx-datasource** and **xa-datasource** are:

- **transaction-isolation**: This element specifies the `java.sql.Connection` transaction isolation level to use. The constants defined in the Connection interface are the possible element content values and include:
 - TRANSACTION_READ_UNCOMMITTED
 - TRANSACTION_READ_COMMITTED
 - TRANSACTION_REPEATABLE_READ
 - TRANSACTION_SERIALIZABLE
 - TRANSACTION_NONE
- **no-tx-separate-pools**: The presence of this element indicates that two connection pools are

required to isolate connections used with JTA transaction from those used without a JTA transaction. The pools are lazily constructed on first use. Its use case is for Oracle (and possibly other vendors) XA implementations that don't like using an XA connection with and without a JTA transaction.

The unique **xa-datasource** child elements are:

- **track-connection-by-tx**: Specifying a true value for this element makes the connection manager keep an xid to connection map and only put the connection back in the pool when the transaction completes and all the connection handles are closed or disassociated (by the method calls returning). As a side effect, we never suspend and resume the xid on the connection's **XAResource**. This is the same connection tracking behavior used for local transactions.

The XA spec implies that any connection may be enrolled in any transaction using any xid for that transaction at any time from any thread (suspending other transactions if necessary). The original JCA implementation assumed this and aggressively delisted connections and put them back in the pool as soon as control left the EJB they were used in or handles were closed. Since some other transaction could be using the connection the next time work needed to be done on the original transaction, there is no way to get the original connection back. It turns out that most **XADataSource** driver vendors do not support this, and require that all work done under a particular xid go through the same connection.

- **xa-datasource-class**: The fully qualified name of the **javax.sql.XADataSource** implementation class, for example, **com.informix.jdbcx.IfxXADataSource**.
- **xa-datasource-property**: The **xa-datasource-property** element allows for specification of the properties to assign to the **XADataSource** implementation class. Each property is identified by the name attribute and the property value is given by the **xa-datasource-property** element content. The property is mapped onto the **XADataSource** implementation by looking for a JavaBeans style getter method for the property name. If found, the value of the property is set using the JavaBeans setter with the element text translated to the true property type using the **java.beans.PropertyEditor** for the type.
- **isSameRM-override-value**: A boolean flag that allows one to override the behavior of the **javax.transaction.xa.XAResource.isSameRM(XAResource xaRes)** method behavior on the XA managed connection. If specified, this value is used unconditionally as the **isSameRM(xaRes)** return value regardless of the **xaRes** parameter.

The failover options common to **ha-xa-datasource** and **ha-local-tx-datasource** are:

- **url-delimeter**: This element specifies a character used to separate multiple JDBC URLs.
- **url-property**: In the case of XA datasources, this property specifies the name of the **xa-datasource-property** that contains the list of JDBC URLs to use.

Example configurations for many third-party JDBC drivers are included in the **JBoss_DIST/docs/examples/jca** directory. Current example configurations include:

- **asapxcess-jb3.2-ds.xml**
- **cicsr9s-service.xml**
- **db2-ds.xml**
- **db2-xa-ds.xml**

- facets-ds.xml
- fast-objects-jboss32-ds.xml
- firebird-ds.xml
- firstsql-ds.xml
- firstsql-xa-ds.xml
- generic-ds.xml
- hsqldb-ds.xml
- informix-ds.xml
- informix-xa-ds.xml
- jdatastore-ds.xml
- jms-ds.xml
- jsql-ds.xml
- lido-versant-service.xml
- mimer-ds.xml
- mimer-xa-ds.xml
- msaccess-ds.xml
- mssql-ds.xml
- mssql-xa-ds.xml
- mysql-ds.xml
- oracle-ds.xml
- oracle-xa-ds.xml
- postgres-ds.xml
- sapdb-ds.xml
- sapr3-ds.xml
- solid-ds.xml
- sybase-ds.xml

7.4. CONFIGURING GENERIC JCA ADAPTORS

The XLSSubDeployer also supports the deployment of arbitrary non-JDBC JCA resource adaptors. The schema for the top-level connection factory elements of the ***-ds.xml** configuration deployment file is shown in [Figure 7.11](#), “The simplified JCA adaptor connection factory configuration descriptor top-level

schema elements”.

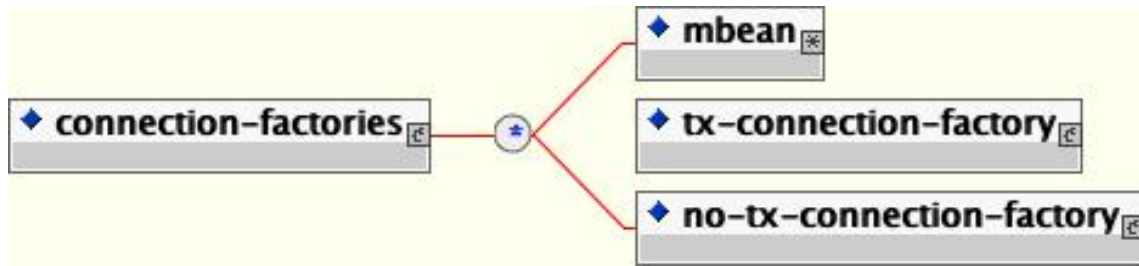


Figure 7.11. The simplified JCA adaptor connection factory configuration descriptor top-level schema elements

Multiple connection factory configurations may be specified in a configuration deployment file. The child elements of the **connection-factories** root are:

- **mbean**: Any number mbean elements may be specified to define MBean services that should be included in the **jboss-service.xml** descriptor that results from the transformation. This may be used to configure additional services used by the adaptor.
- **no-tx-connection-factory**: this element is used to specify the (**org.jboss.resource.connectionmanager**) **NoTxConnectionManager** service configuration. **NoTxConnectionManager** is a JCA connection manager with no transaction support. The **no-tx-connection-factory** child element schema is given in [Figure 7.12](#), “The no-tx-connection-factory element schema”.
- **tx-connection-factory**: this element is used to specify the (**org.jboss.resource.connectionmanager**) **TxConnectionManager** service configuration. The **tx-connection-factory** child element schema is given in [Figure 7.13](#), “The tx-connection-factory element schema”.

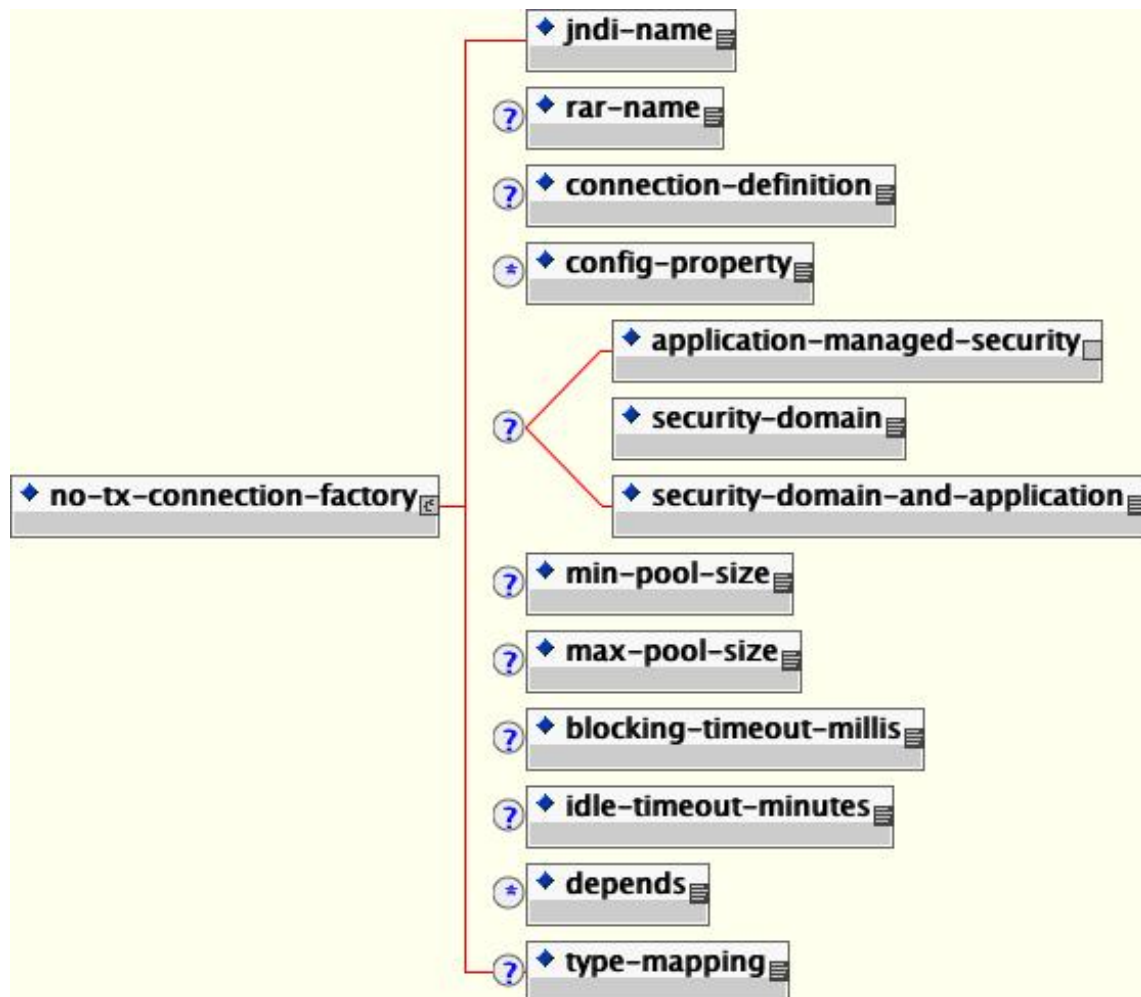


Figure 7.12. The no-tx-connection-factory element schema

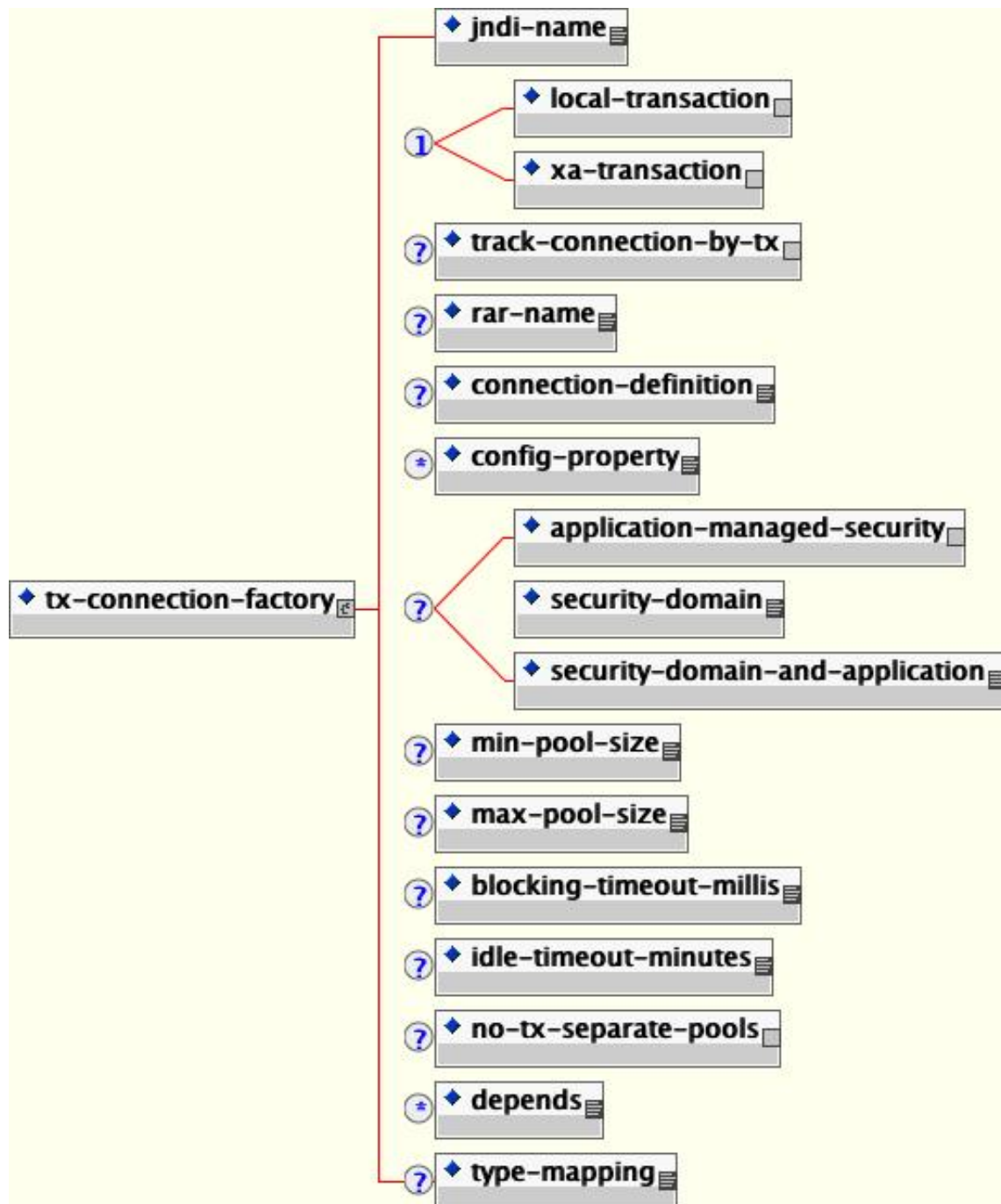


Figure 7.13. The tx-connection-factory element schema

The majority of the elements are the same as those of the datasources configuration. The element unique to the connection factory configuration include:

- **adaptor-display-name**: A human readable display name to assign to the connection manager MBean.
- **local-transaction**: This element specifies that the **tx-connection-factory** supports local transactions.
- **xa-transaction**: This element specifies that the **tx-connection-factory** supports XA transactions.
- **track-connection-by-tx**: This element specifies that a connection should be used only on a single transaction and that a transaction should only be associated with one connection.

- **rar-name**: This is the name of the RAR file that contains the definition for the resource we want to provide. For nested RAR files, the name would look like **myapplication.ear#my.rar**.
- **connection-definition**: This is the connection factory interface class. It should match the **connectionfactory-interface** in the **ra.xml** file.
- **config-property**: Any number of properties to supply to the **ManagedConnectionFactory** (MCF) MBean service configuration. Each **config-property** element specifies the value of a MCF property. The **config-property** element has two required attributes:
 - **name**: The name of the property
 - **type**: The fully qualified type of the property

The content of the **config-property** element provides the string representation of the property value. This will be converted to the true property type using the associated type **PropertyEditor**.

CHAPTER 8. TRANSACTIONS ON JBOSS

The JTA Transaction Service

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

8.1. 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.

8.1.1. Pessimistic and optimistic locking

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

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

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

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

8.1.2. 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 JBoss Messaging are examples of recoverable resources. A recoverable resource is represented using the `javax.transaction.xa.XAResource` interface in JTA.

8.1.3. 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.

8.1.4. 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.

8.1.5. 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.

8.2. JTS SUPPORT

JBoss Transactions is a 100% Java implementation of a distributed transaction management system based on the Sun Microsystems J2EE Java Transaction Service (JTS) standard. Our implementation of the JTS utilizes the Object Management Group's (OMG) Object Transaction Service (OTS) model for transaction interoperability as recommended in the J2EE and EJB standards and leads the market in providing many advanced features such as fully distributed transactions and ORB portability with POA support.

8.3. WEB SERVICES TRANSACTIONS

In traditional ACID transaction systems, transactions are short lived, resources (such as databases) are locked for the duration of the transaction and participants have a high degree of trust with each other. With the advent of the Internet and Web services, the scenario that is now emerging requires involvement of participants unknown to each other in distributed transactions. These transactions have the following characteristics:

- Transactions may be of a long duration, sometimes lasting hours, days, or more.
- Participants may not allow their resources to be locked for long durations.
- The communication infrastructure between participants may not be reliable.
- Some of the ACID properties of traditional transactions are not mandatory.
- A transaction may succeed even if only some of the participants choose to confirm and others cancel.
- All participants may choose to have their own coordinator (Transaction Manager), because of lack of trust.
- All activities are logged.
- Transactions that have to be rolled back have the concept of compensation.

JBoss Transactions adds native support for Web services transactions by providing all of the components necessary to build interoperable, reliable, multi-party, Web services-based applications with

the minimum of effort. The programming interfaces are based on the Java API for XML Transactioning (JAXTX) and the product includes protocol support for the WS-AtomicTransaction and WS-BusinessActivity specifications. JBossTS 4.2 is designed to support multiple coordination protocols and therefore helps to future-proof transactional applications.

8.4. CONFIGURING JBOSS TRANSACTIONS

JBossTS is configured through the `jbossjts-properties.xml` property file. You should consult the JBossTS documentation for all of the configurable options it supports.

8.5. LOCAL VERSUS DISTRIBUTED TRANSACTIONS

Local Transactions

A Local Transaction allows resource enlistment at only one JVM and does not span across multiple process instances (i.e., VMs). However a separate client JVM may still manage transaction boundaries (begin/commit/rollback) for the JTA. Databases and message queues running as separate processes may still be enlisted as XAResources provided they have drivers that support this.

Distributed Transactions

A transaction is considered to be distributed if it spans multiple process instances, i.e., VMs. Typically a distributed transaction will contain participants (e.g., XAResources) that are located within multiple VMs but the transaction is coordinated in a separate VM (or co-located with one of the participants).

JBossTS provides both local and distributed transactions. If your architecture requires distributed transactions then you should consider using either the JTS implementation from JBossTS, which uses CORBA for communication, or the Web Services transactions component, which uses SOAP/HTTP. Although the JTS/XTS component can be used with JBoss Enterprise Application Platform, it is not a supported part of the platform.



NOTE

JTS and XTS components are not supported for JBoss Enterprise Application Platform 4.x

CHAPTER 9. MESSAGING ON JBOSS

The JMS API stands for Java Message Service Application Programming Interface, and it is used by applications to send asynchronous *business-quality* messages to other applications. In the messaging world, messages are not sent directly to other applications. Instead, messages are sent to destinations, known as queues or topics. Applications sending messages do not need to worry if the receiving applications are up and running, and conversely, receiving applications do not need to worry about the sending application's status. Both senders, and receivers only interact with the destinations.

The JMS API is the standardized interface to a JMS provider, sometimes called a Message Oriented Middleware (MOM) system. JBoss comes with a JMS 1.1 compliant JMS provider called JBoss Messaging. When you use the JMS API with JBoss, you are using the JBoss Messaging engine transparently. JBoss Messaging fully implements the JMS specification; therefore, the best JBoss Messaging user guide is the JMS specification. For more information about the JMS API please visit the [JMS Tutorial](#) or [JMS Downloads & Specifications](#).

9.1. JBOSS MESSAGING OVERVIEW

JBoss Messaging is the new state-of-the-art enterprise messaging system from JBoss, providing superior performance, reliability and scalability with high throughput and low latency. JBoss Messaging has replaced JBossMQ as the default JMS provider in JBoss Enterprise Application Platform 4.3. Since JBoss Messaging is JMS 1.1 and JMS 1.0.2b compatible, the JMS code written against JBossMQ will run with JBoss Messaging without any changes.

For more details on configurations and examples, refer to the [JBoss Messaging User Guide](#).

CHAPTER 10. SECURITY ON JBOSS

J2EE Security Configuration and Architecture

Security is a fundamental part of any enterprise application. You need to be able to restrict who is allowed to access your applications and control what operations application users may perform. The J2EE specifications define a simple role-based security model for EJBs and web components. The JBoss component framework that handles security is the JBossSX extension framework. The JBossSX security extension provides support for both the role-based declarative J2EE security model and 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 EJB and servlet specification security models, as well as JAAS to establish the foundation for these details.

10.1. J2EE DECLARATIVE SECURITY OVERVIEW

The J2EE security model declarative in that you describe the security roles and permissions in a standard XML descriptor rather than embedding security into your business component. This isolates security from business-level code because security tends to be more a function of where the component is deployed 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 independently of how you access the bank account, based on what bank is managing the account, where the ATM is located, 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. The following sections look at the purpose and usage of the various security elements.

10.1.1. Security References

Both EJBs and servlets can declare one or more **security-role-ref** elements as shown in [Figure 10.1](#), “The security-role-ref element”. This element declares that a component is using the **role-name** value as an argument to the **isCallerInRole(String)** method. By using the **isCallerInRole** method, a component can verify whether 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 by using the role-based **method-permissions** elements.

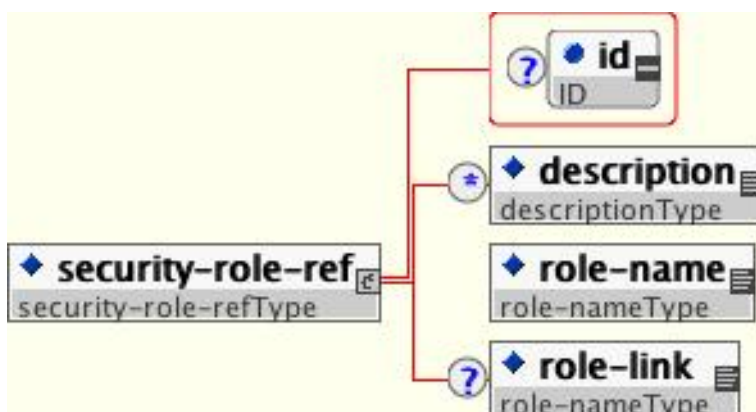


Figure 10.1. The security-role-ref element

Example 10.1, “An ejb-jar.xml descriptor fragment that illustrates the security-role-ref element usage.” shows the use of **security-role-ref** in an **ejb-jar.xml**.

Example 10.1. An ejb-jar.xml descriptor fragment that illustrates 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>
```

Example 10.2, “An example web.xml descriptor fragment that illustrates the security-role-ref element usage.” shows the use of **security-role-ref** in a **web.xml**.

Example 10.2. An example web.xml descriptor fragment that illustrates the security-role-ref element usage.

```
<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>
```

10.1.2. Security Identity

An EJB has the capability to specify what identity an EJB should use when it invokes methods on other components using the **security-identity** element, shown in [Figure 10.2, “The security-identity element”](#)

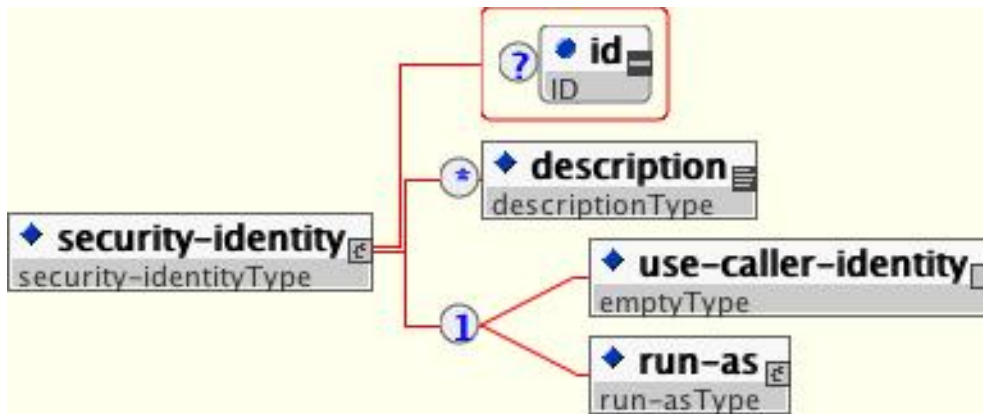


Figure 10.2. The security-identity element

The invocation identity can be that of the current caller, or it can be a specific role. The application assembler uses the **security-identity** element with a **use-caller-identity** child element to indicate that 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 the **EJBContext.getCallerPrincipal()** method. 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. You accomplish this 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. The following descriptor fragment that illustrates **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>

```

When you use **run-as** to assign a specific role to outgoing calls, JBoss associates a principal named

anonymous. If you want another principal to be associated with the call, you need to associate a **run-as-principal** with the bean in the **jboss.xml** file. The following fragment associates a principal named **internal** with **RunAsBean** from the prior example.

```
<session>
  <ejb-name>RunAsBean</ejb-name>
  <security-identity>
    <run-as-principal>internal</run-as-principal>
  </security-identity>
</session>
```

The **run-as** element is also available in servlet definitions in a **web.xml** file. The following example shows how to assign the role **InternalRole** to a servlet:

```
<servlet>
  <servlet-name>AServlet</servlet-name>
  <!-- ... -->
  <run-as>
    <role-name>InternalRole</role-name>
  </run-as>
</servlet>
```

Calls from this servlet will be associated with the anonymous **principal**. The **run-as-principal** element is available in the **jboss-web.xml** file to assign a specific principal to go along with the **run-as** role. The following fragment shows how to associate a principal named **internal** to the servlet in the prior example.

```
<servlet>
  <servlet-name>AServlet</servlet-name>
  <run-as-principal>internal</run-as-principal>
</servlet>
```

10.1.3. 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.

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 such as BankManager, Teller, or Customer.

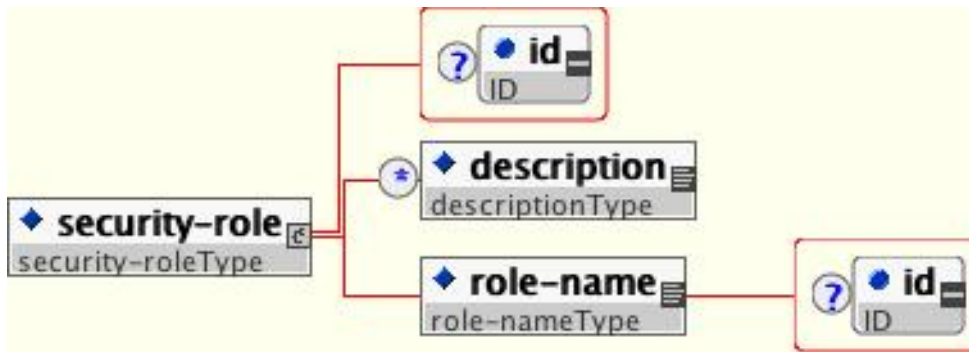


Figure 10.3. The security-role element

In JBoss, a **security-role** element is only used to map **security-role-ref/role-name** values to the logical role that the component role references. 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-role** elements in order to declare method permissions. However, the specification of **security-role** elements is still a recommended practice to ensure portability across application servers and for deployment descriptor maintenance. [Example 10.3, "An ejb-jar.xml descriptor fragment that illustrates the security-role element usage."](#) shows the usage of the **security-role** in an **ejb-jar.xml** file.

Example 10.3. An ejb-jar.xml descriptor fragment that illustrates 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>

```

[Example 10.4, "An example web.xml descriptor fragment that illustrates the security-role element usage."](#) shows the usage of the **security-role** in an **web.xml** file.

Example 10.4. An example web.xml descriptor fragment that illustrates the security-role element usage.

```

<!-- A sample web.xml fragment -->
<web-app>
  <!-- ... -->
  <security-role>
    <description>The single application role</description>
    <role-name>TheApplicationRole</role-name>
  </security-role>
</web-app>

```

10.1.4. 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.

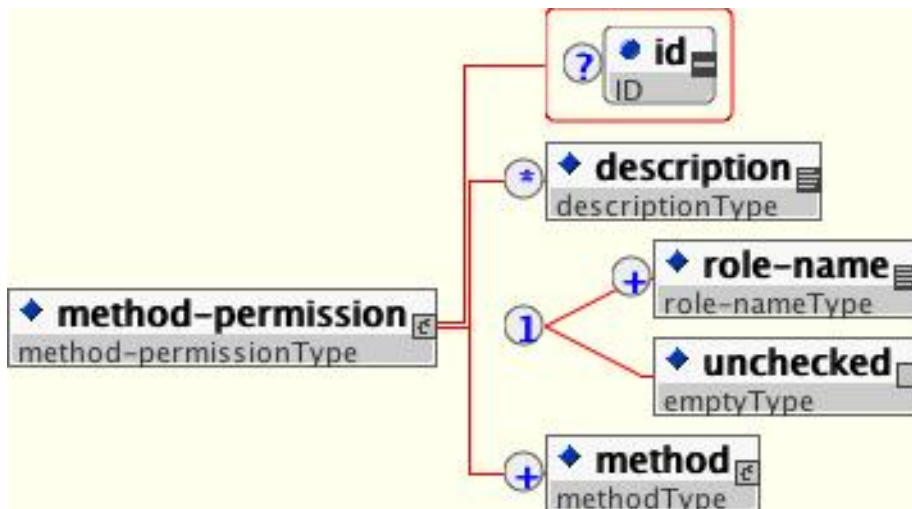


Figure 10.4. The method-permissions element

Each `method-permission` element contains one or more `role-name` child elements that define the logical roles that are allowed to access the EJB methods as identified by `method` child elements. You can also 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 that has 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`.

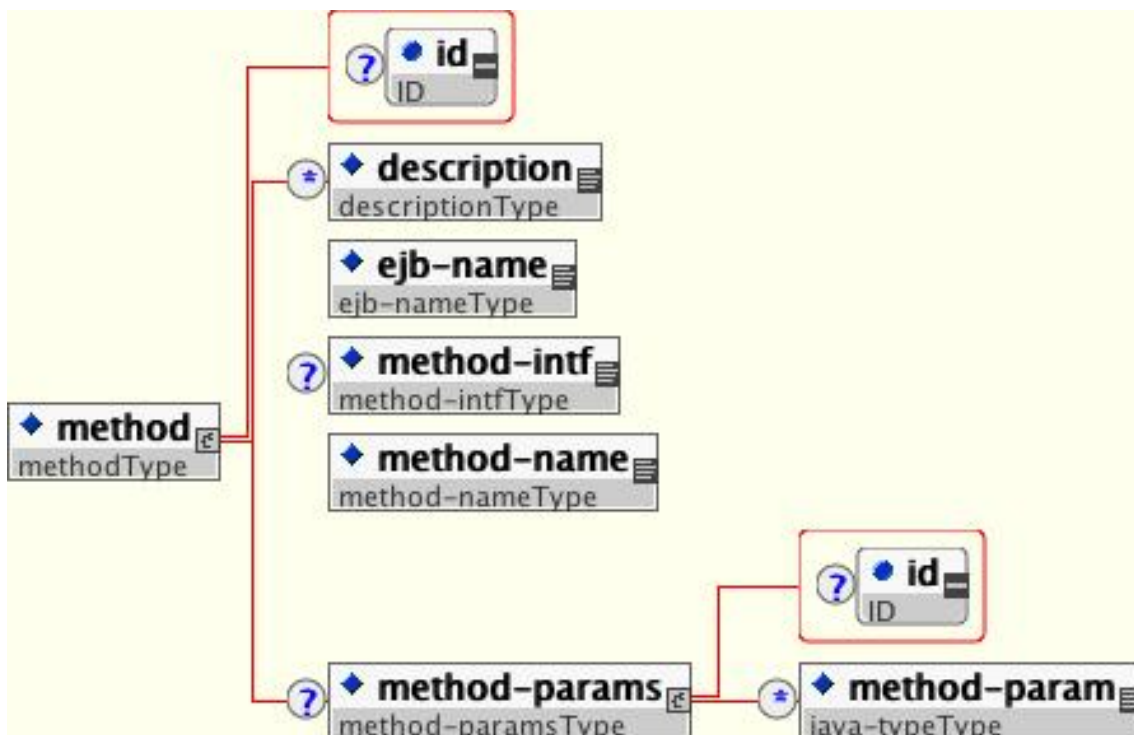


Figure 10.5. The method element

There are three supported styles of method element declarations.

The first is used for referring to all the home and component interface methods of the named enterprise bean:

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>*</method-name>
</method>
```

The second style is used for referring to a specified method of the home or component interface of the named enterprise bean:

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
</method>
```

If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods.

The third style is used to refer to a specified method within a set of methods with an overloaded name:

```
<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 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.

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.

[Example 10.5, “An ejb-jar.xml descriptor fragment that illustrates the method-permission element usage.”](#) provides complete examples of the **method-permission** element usage.

Example 10.5. An ejb-jar.xml descriptor fragment that 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>

```

10.1.5. Web Content Security Constraints

In a web application, security is defined by the roles that are allowed access to content by a URL pattern that identifies the protected content. This set of information is declared by using the `web.xmlsecurity-constraint` element.

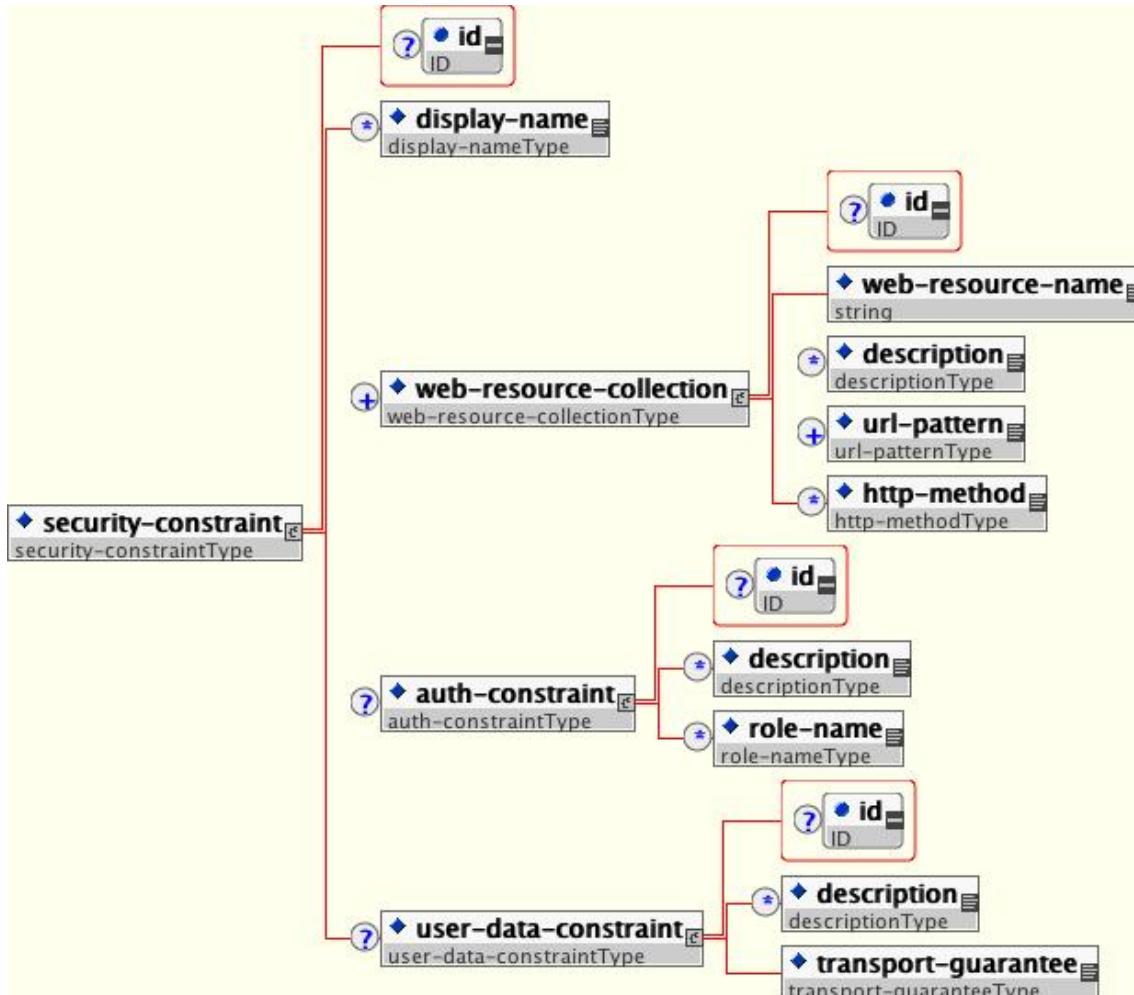


Figure 10.6. The security-constraint element

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 the client and server should be protected. Its values are **NONE**, **INTEGRAL**, and **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 to 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 to 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** element is used to configure the authentication method that should be used, the realm name that should be used for rhw application, and the attributes that are needed by the form login mechanism.

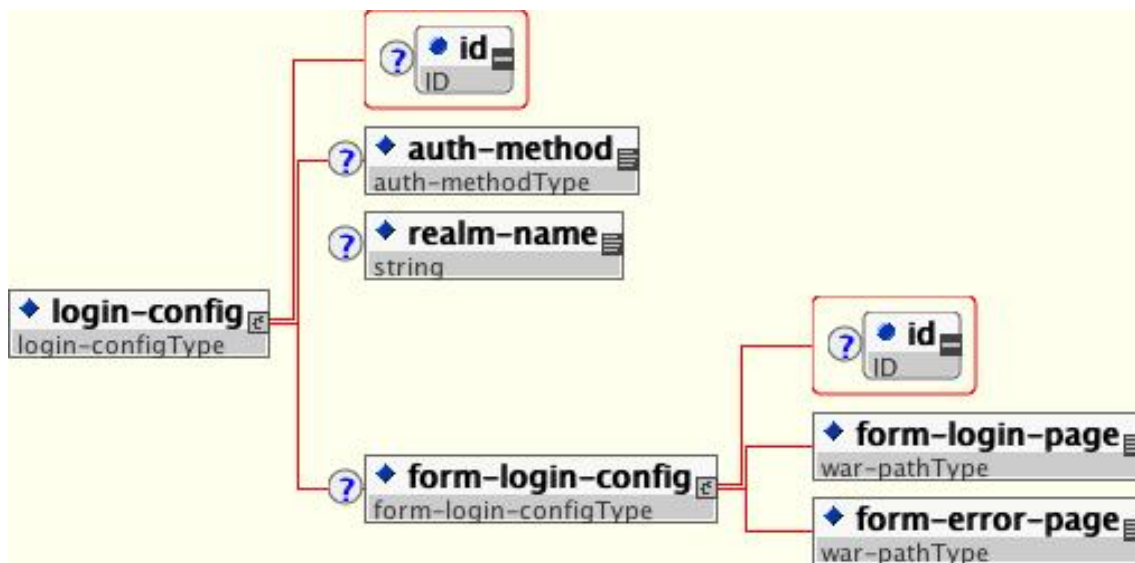


Figure 10.7. The login-config element

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 **auth-method** values are **BASIC**, **DIGEST**, **FORM**, and **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**, then **form-login-config** and its child elements are ignored.

As an example, the **web.xml** descriptor fragment given in [Example 10.6](#), “[A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements.](#)” indicates that any URL lying under the web application's **/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.

Example 10.6. 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>
```

10.1.6. Enabling Declarative Security in JBoss

The J2EE security elements that have been covered so far describe the security requirements only from the application's perspective. Because 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. The details behind the security configuration are discussed in [Section 10.3, "The JBoss Security Model"](#).

10.2. AN INTRODUCTION TO JAAS

The JBossSX framework is based on the JAAS API. It is important that you understand the basic elements of the JAAS API to understand the implementation details of JBossSX. The following sections provide an introduction to JAAS to prepare you for the JBossSX architecture discussion later in this chapter.

10.2.1. 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 JDK 1.4+. 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.

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.

10.2.1.1. 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.

There are the common classes:

- **Subject** (`javax.security.auth.Subject`)

- **Principal** (`java.security.Principal`)

These are the 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`)

10.2.1.1.1. The Subject and Principal Classes

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), a social security number principal (123-45-6789), and a username principal (johnd), all of which help distinguish the subject from other subjects. To retrieve the principals associated with a subject, two methods are available:

```
public Set getPrincipals() {...}
public Set getPrincipals(Class c) {...}
```

The first method returns all principals contained in the subject. The second method returns only those principals that are instances of class `c` or one of its subclasses. An empty set is returned if the subject has no matching principals. Note that the `java.security.acl.Group` interface is a subinterface of `java.security.Principal`, so an instance in the principals set may represent a logical grouping of other principals or groups of principals.

10.2.1.1.2. 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** and passes 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 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 can 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 that is independent of the underlying authentication technology. The **LoginContext** consults a **Configuration** to determine the authentication services configured for a particular application. **LoginModule** classes represent the authentication services. Therefore, you can plug different login modules into an application without changing the application itself. The following code shows the steps required by an application to authenticate a subject.

```
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 an administrator to plug different authentication technologies into an application. You can chain together multiple **LoginModules** to allow for more than one authentication technology to participate in 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 two phases. The steps of the process are as follows:

- The **LoginContext** creates each configured **LoginModule** using its public no-arg constructor.
- 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)**.
- The **login** method is called to start the authentication process. For example, a 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**. A **LoginException** indicates failure. A return value of true indicates that the method succeeded, whereas a return value of false indicates that the login module should be ignored.
- If the **LoginContext**'s overall authentication succeeds, **commit** is invoked on each **LoginModule**. If phase 1 succeeds for a **LoginModule**, then the commit method continues with phase 2 and associates the 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**. Failure to complete the commit phase is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.
- If the **LoginContext**'s overall authentication fails, then the **abort** method is invoked on each **LoginModule**. The **abort** method removes or destroys any authentication state created by the login or initialize methods. The signature of the **abort** method is **boolean abort() throws LoginException**. Failure to complete the **abort** phase is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.
- To remove the authentication state after a successful login, 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**. Failure to complete the logout process is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

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 login modules. Login modules use the **CallbackHandler** both to gather input from users, such as a password or smart card PIN, 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 by using an application server API. The **callbackhandler** interface has one method to implement:

```
void handle(Callback[] callbacks)
    throws java.io.IOException,
           UnsupportedCallbackException;
```

The **Callback** interface is the last authentication class we will look at. This is a tagging interface for which several default implementations are provided, including the **NameCallback** and **PasswordCallback** used in an earlier example. A **LoginModule** uses a **Callback** to request information required by the authentication mechanism. **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.

10.3. 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. Three basic interfaces define the JBoss server security layer: **org.jboss.security.AuthenticationManager**, **org.jboss.security.RealmMapping**, and **org.jboss.security.SecurityProxy**. [Figure 10.8](#), “The key security model interfaces and their relationship to the JBoss server EJB container elements.” shows a class diagram of the security interfaces and their relationship to the EJB container architecture.

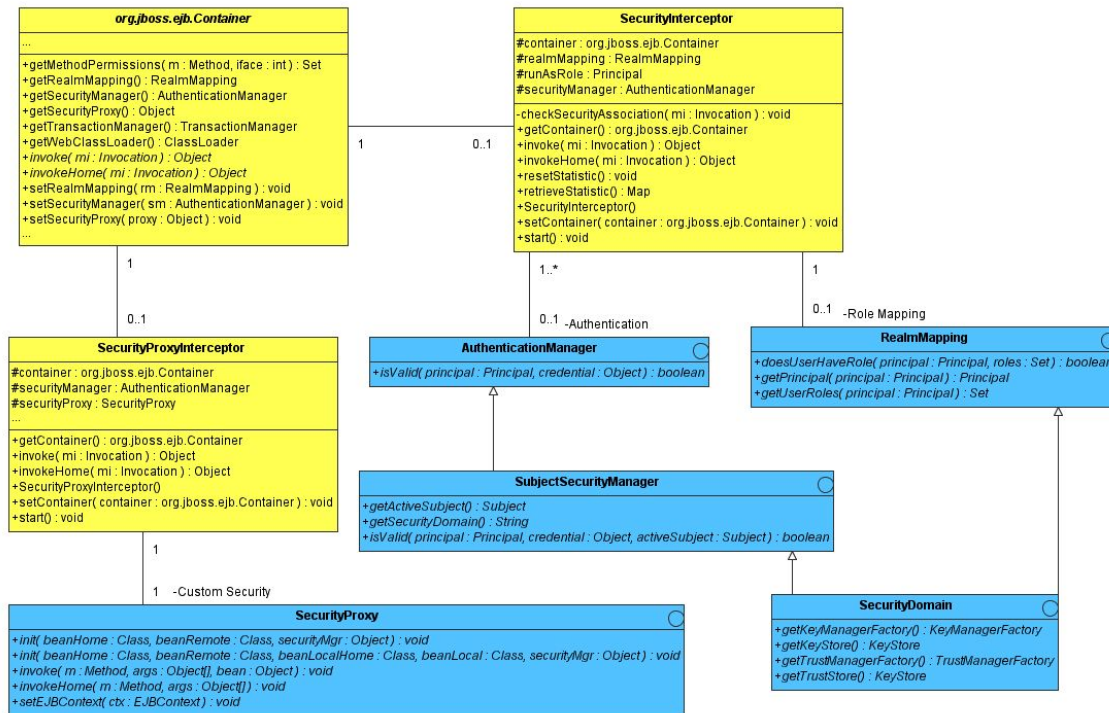


Figure 10.8. The key security model interfaces and their relationship to the JBoss server EJB container elements.

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 **org.jboss.security.AuthenticationManager** and **org.jboss.security.RealmMapping**. The roles of the security interfaces presented in Figure 10.8, “The key security model interfaces and their relationship to the JBoss server EJB container elements.” are summarized in the following list.

- AuthenticationManager:** This interface is responsible for validating credentials associated with principals. Principals are identities, such as usernames, employee numbers, and social security numbers. Credentials are proof of the identity, such as passwords, session keys, and digital signatures. The **isValid** method is invoked to determine whether a user identity and associated credentials as known in the operational environment are valid proof of the user's identity.
- RealmMapping:** This interface is 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:** This interface describes 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:** This is a subinterface of **AuthenticationManager** that adds accessor methods for obtaining the security domain name of the security manager and the current thread's authenticated **Subject**.
- SecurityDomain:** This 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 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 10.9, “The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.”](#) illustrates this fact. The implication of this plug-in architecture is that you are free to replace the JAAS-based JBossSX implementation classes with your own custom security manager implementation that does not make use of JAAS, if you so desire. You'll see how to do this when you look at the JBossSX MBeans available for the configuration of JBossSX in [Figure 10.9, “The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.”](#)

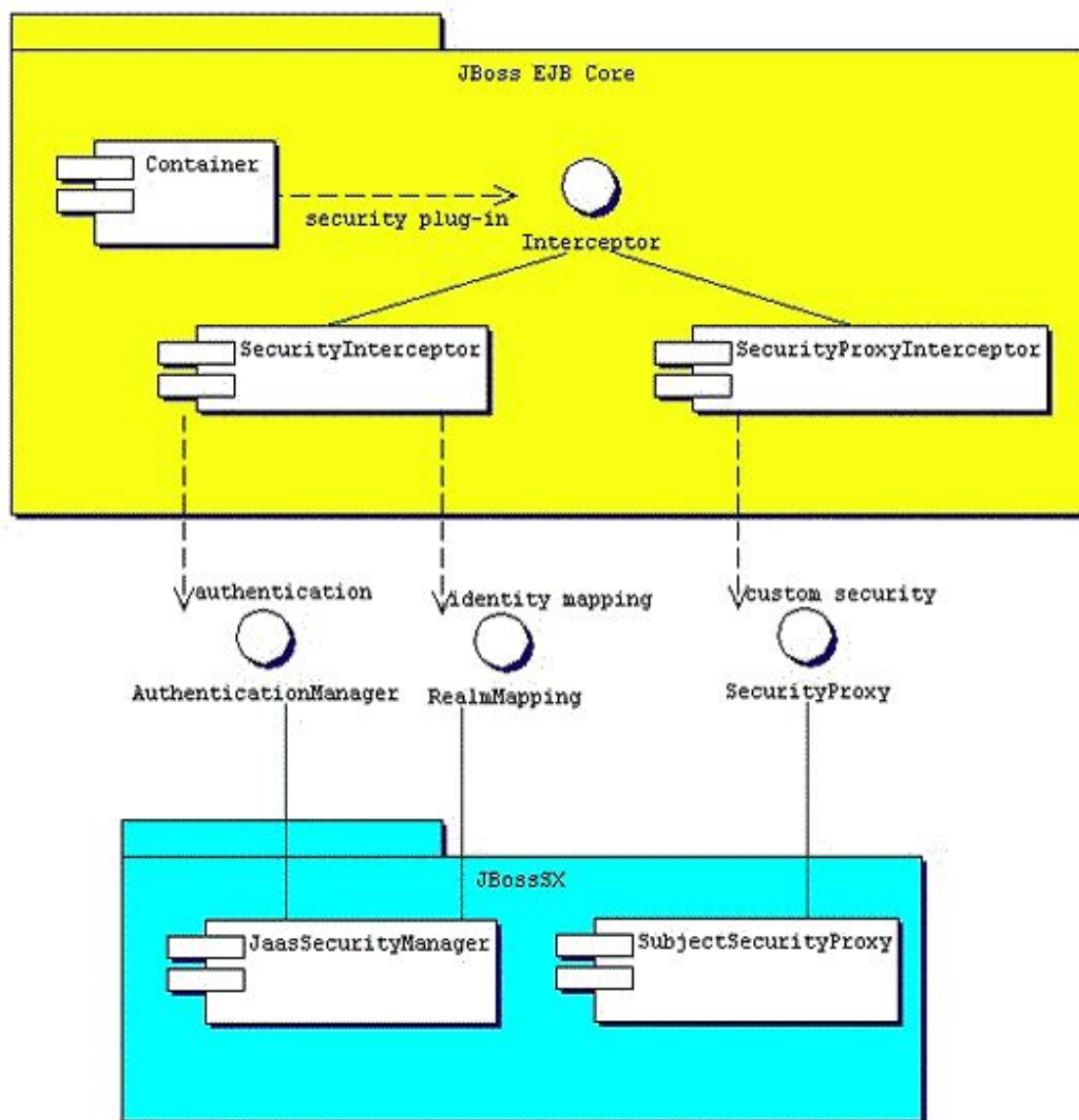


Figure 10.9. The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.

10.3.1. Enabling Declarative Security in JBoss Revisited

Earlier in this chapter, the 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 are presented here. Figure 10.10, “The security element subsets of the JBoss server `jboss.xml` and `jboss-web.xml` deployment descriptors.” shows the JBoss-specific EJB and web application deployment descriptor's security-related elements.

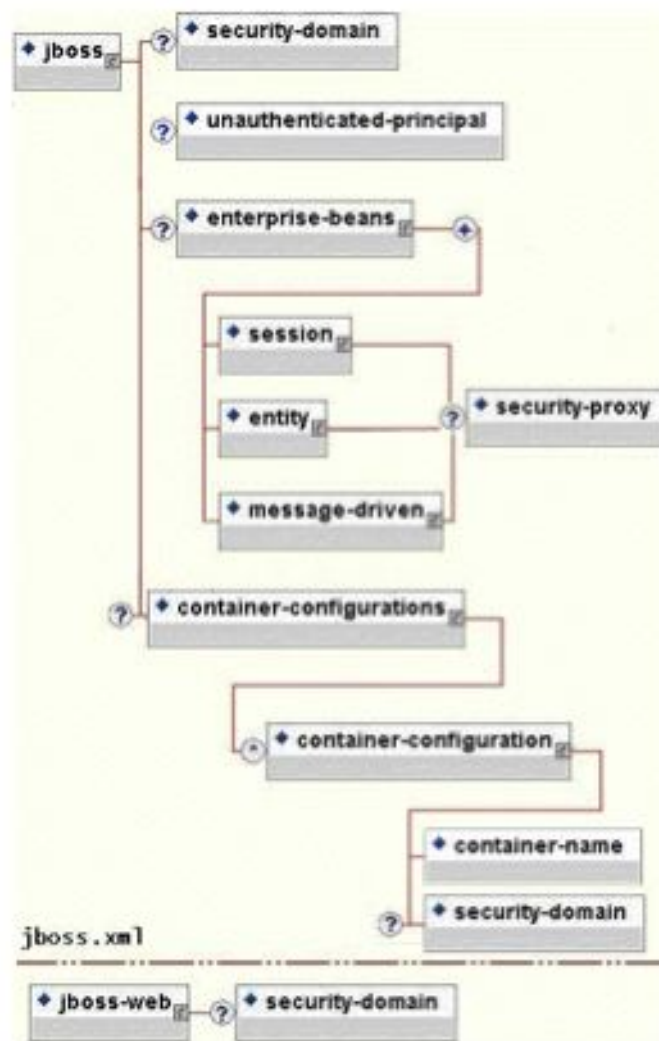


Figure 10.10. The security element subsets of the JBoss server `jboss.xml` and `jboss-web.xml` deployment descriptors.

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, remote, local home or local interfaces 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 **SecurityProxy** 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; 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 [Example 10.7, “The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.”](#), and the full source code is available in the **src/main/org/jboss/book/security/ex1** directory of the book examples.

Example 10.7. The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.

```
package org.jboss.book.security.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: 1.4 $
 */
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
}
}

```

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 associated **jboss.xml** descriptor that installs the **EchoSecurityProxy** as the custom proxy for the **EchoBean** is given in [Example 10.8, “The jboss.xml descriptor, which configures the EchoSecurityProxy as the custom security proxy for the EchoBean.”](#)

Example 10.8. 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.book.security.ex1.EchoSecurityProxy</security-proxy>
        </session>
    </enterprise-beans>
</jboss>
```

Now test the custom proxy by running a client that attempts to invoke the **EchoBean.echo** method with the arguments **Hello** and **Four** as illustrated in this fragment:

```
public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        Logger log = Logger.getLogger("ExClient");
        log.info("Looking up EchoBean");

        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();

        log.info("Created Echo");
        log.info("Echo.echo('Hello') = "+echo.echo("Hello"));
        log.info("Echo.echo('Four') = "+echo.echo("Four"));
    }
}
```

The first call should succeed, while the second should fail due to the fact that **Four** is a four-letter word. Run the client as follows using Ant from the examples directory:

```
[examples]$ ant -Dchap=security -Dex=1 run-example
run-example1:
...
[echo] Waiting for 5 seconds for deploy...
[java] [INFO,ExClient] Looking up EchoBean
[java] [INFO,ExClient] Created Echo
[java] [INFO,ExClient] Echo.echo('Hello') = Hello
[java] Exception in thread "main" java.rmi.AccessException:
SecurityException; nested exception is:
[java]     java.lang.SecurityException: No 4 letter words
...
[java] Caused by: java.lang.SecurityException: No 4 letter words
...
```

The result is that the **echo('Hello')** method call succeeds as expected and the **echo('Four')** method call results in a rather messy looking exception, which is also expected. The above output has been truncated to fit in the book. The key part to the exception is that the **SecurityException("No 4 letter words")** generated by the **EchoSecurityProxy** was thrown to abort the attempted method invocation as desired.

10.4. THE JBOSS SECURITY EXTENSION ARCHITECTURE

The preceding discussion of the general JBoss security layer has stated that the JBossSX security extension framework is an implementation of the security layer interfaces. This is the primary purpose of the JBossSX framework. The details of the implementation are interesting in that it offers a great deal of customization for integration into existing security infrastructures. A security infrastructure can be anything from a database or LDAP server to a sophisticated security software suite. The integration flexibility is achieved using the pluggable authentication model available in the JAAS framework.

The heart of the JBossSX framework is **org.jboss.security.plugins.JaasSecurityManager**. This is the default implementation of the **AuthenticationManager** and **RealmMapping** interfaces. [Figure 10.11](#), “The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.” shows how the **JaasSecurityManager** integrates into the EJB and web container layers based on the **security-domain** element of the corresponding component deployment descriptor.

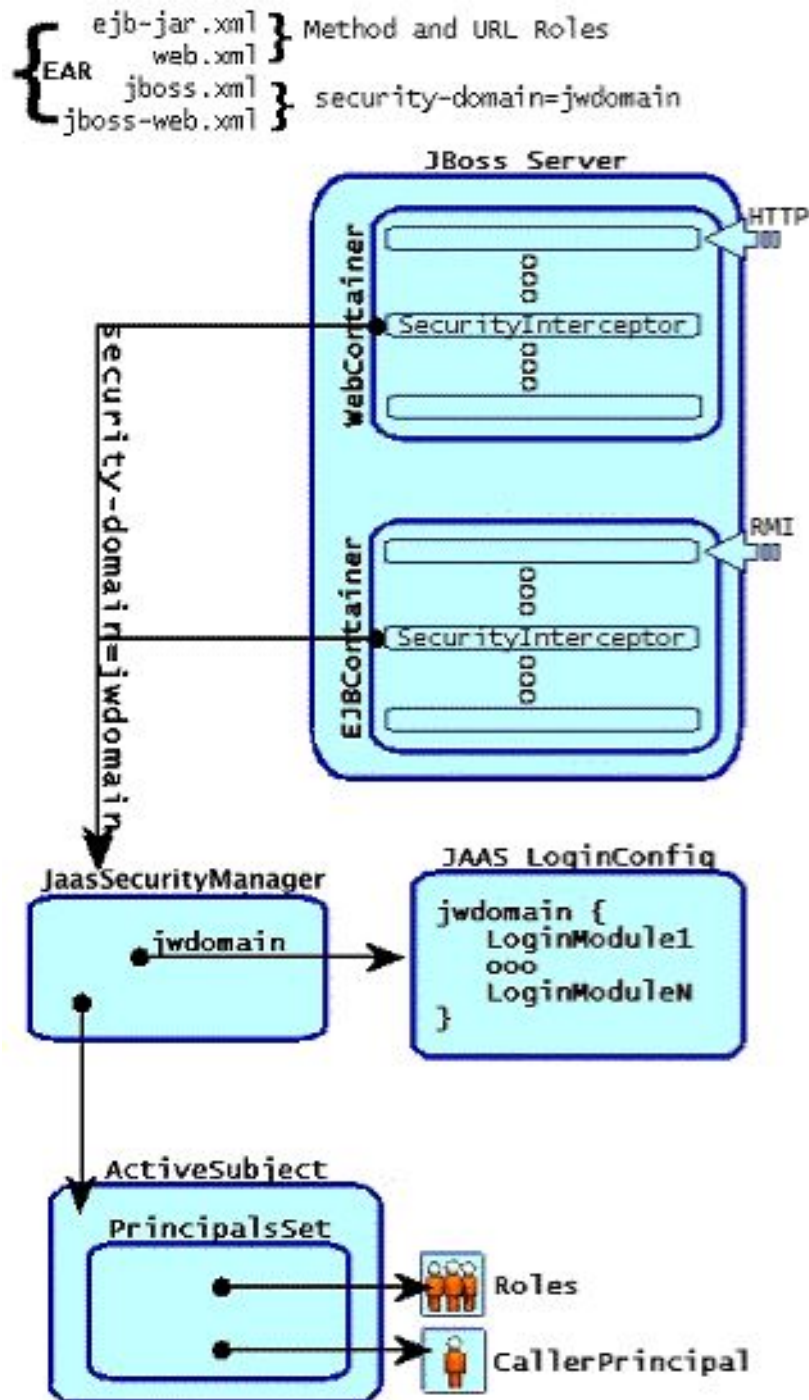


Figure 10.11. The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.

Figure 10.11, “The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.” 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 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.

10.4.1. 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.

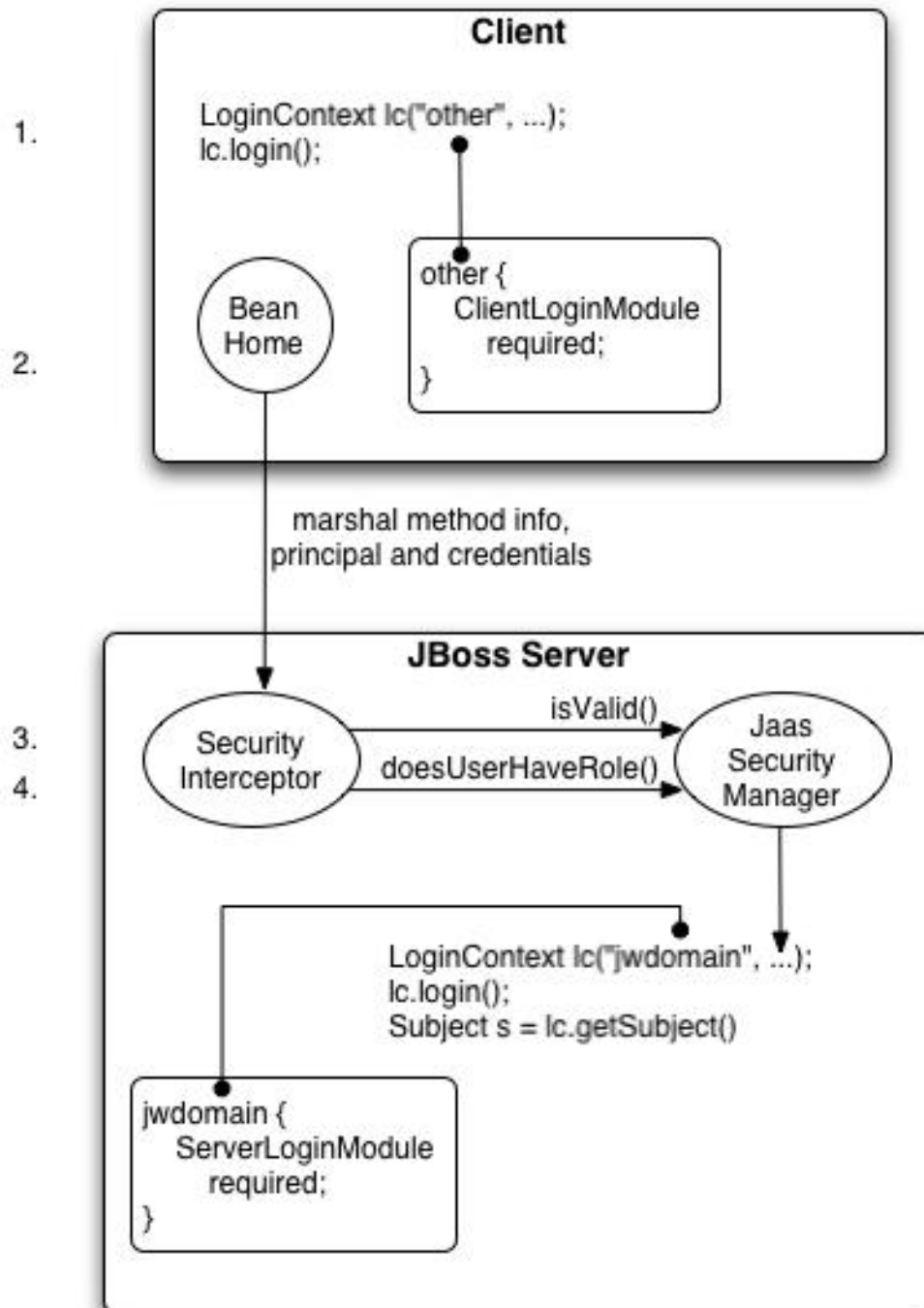


Figure 10.12. An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.

Figure 10.12, “An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.” provides a view of the client to server communication we will discuss. The numbered steps shown are:

1. The client first has to perform a JAAS login to establish the principal and credentials for authentication, and this is labeled *Client Side Login* in the figure. This is how clients establish their login identities in JBoss. Support for presenting the login information via JNDI **InitialContext** properties is provided via an alternate configuration. A JAAS login entails creating a **LoginContext** instance and passing the name of the configuration to use. 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 this example, the **other** client-side login configuration entry is set up to use the

ClientLoginModule module (an **org.jboss.security.ClientLoginModule**). This is the default client side module that simply binds the username and password to the JBoss EJB invocation layer for later authentication on the server. The identity of the client is not authenticated on the client.

2. Later, the client obtains the EJB home interface and attempts to create a bean. This event is labeled as *Home Method Invocation*. This results in a home interface method invocation being sent to the JBoss server. The invocation includes the method arguments passed by the client along with the user identity and credentials from the client-side JAAS login performed in step 1.
3. On the server side, the security interceptor first requires authentication of the user invoking the call, which, as on the client side, involves a JAAS login.
4. The security domain under which the EJB is secured determines the choice of login modules. The security domain name is used as the login configuration entry name passed to the **LoginContext** constructor. 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**:
 - 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.
5. 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 10.12, “An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.”](#). 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 **doesUserHaveRole** method is invoked on the security manager by the security interceptor to see if the caller has one of the assigned role names. This method 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.

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.

10.4.2. 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.

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 `JaasSecurityManagerServiceSecurityManagerClassName` 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 the JAAS-based `org.jboss.security.plugins.JaasSecurityManager`.
- **CallbackHandlerClassName:** The name of the class that provides the `javax.security.auth.callback.CallbackHandler` implementation used by the `JaasSecurityManager`. You can override the handler used by the `JaasSecurityManager` if the default implementation (`org.jboss.security.auth.callback.SecurityAssociationHandler`) does not meet your needs. This is a rather deep configuration that generally should not be set unless you know what you are doing.
- **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. The default value is 1800 seconds (30 minutes). The value you use for the timeout is a tradeoff between frequent authentication operations and how long credential information may be out of sync with respect to the security information store. If you want to disable caching of security credentials, set this to 0 to force authentication to occur every time. 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 at which the cache current timestamp is updated and should be less than the **DefaultCacheTimeout** in order for the timeout to be meaningful. The default resolution is 60 seconds(1 minute). This has no affect if the **AuthenticationCacheJndiName** has been changed from the default value.
- **DefaultUnauthenticatedPrincipal:** Specifies the principal to use for unauthenticated users. This setting makes it possible to set default permissions for users who have not been authenticated.

The `JaasSecurityManagerService` also supports a number of useful operations. These include flushing any security domain authentication cache at runtime, getting the list of active users in a security domain authentication cache, and any of the security manager interface methods.

Flushing a security domain authentication cache can be used 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: **public void flushAuthenticationCache(String securityDomain)**.

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
server.invoke(jaasMgr, "flushAuthenticationCache", params, signature);
```

Getting the list of active users provides a snapshot of the **Principals** keys in a security domain authentication cache that are not expired. The MBean operation signature is: **public List getAuthenticationCachePrincipals(String securityDomain)**.

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
List users = (List) server.invoke(jaasMgr,
    "getAuthenticationCachePrincipals",
    params, signature);
```

The security manager has a few additional access methods.

```
public boolean isValid(String securityDomain, Principal principal, Object
credential);
public Principal getPrincipal(String securityDomain, Principal principal);
public boolean doesUserHaveRole(String securityDomain, Principal
principal,
                                Object credential, Set roles);
public Set getUserRoles(String securityDomain, Principal principal, Object
credential);
```

They provide access to the corresponding **AuthenticationManager** and **RealmMapping** interface method of the associated security domain named by the **securityDomain** argument.

10.4.3. The JaasSecurityDomain MBean

The **org.jboss.security.plugins.JaasSecurityDomain** is an extension of **JaasSecurityManager** that adds the notion of a **KeyStore**, a JSSE **KeyManagerFactory** and a **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. The default is **JKS**.

- **KeyStoreURL**: A URL to the location of the **KeyStore** database. This is used to obtain an **InputStream** to initialize the **KeyStore**. If the string is not a value URL, it is treated as a file.
- **KeyStorePass**: The password associated with the **KeyStore** database contents. The **KeyStorePass** is also used in combination with the **Salt** and **IterationCount** attributes to create a PBE secret key used with the encode/decode operations. The **KeyStorePass** attribute value format is one of the following:
 - The plaintext password for the **KeyStore**. The **toCharArray()** value of the string is used without any manipulation.
 - A command to execute to obtain the plaintext password. The format is **{EXT}...** where the **...** is the exact command line that will be passed to the **Runtime.exec(String)** method to execute a platform-specific command. The first line of the command output is used as the password.
 - A class to create to obtain the plaintext password. The format is **{CLASS}classname[:ctorarg]** where the **[:ctorarg]** is an optional string that will be passed to the constructor when instantiating the **classname**. The password is obtained from **classname** by invoking a **toCharArray()** method if found, otherwise, the **toString()** method is used.
- **Salt**: The **PBEParameterSpec** salt value.
- **IterationCount**: The **PBEParameterSpec** iteration count value.
- **TrustStoreType**: The type of the **TrustStore** implementation. This is the type argument passed to the **java.security.KeyStore.getInstance(String type)** factory method. The default is **JKS**.
- **TrustStoreURL**: A URL to the location of the **TrustStore** database. This is used to obtain an **InputStream** to initialize the **KeyStore**. If the string is not a value URL, it is treated as a file.
- **TrustStorePass**: The password associated with the trust store database contents. The **TrustStorePass** is a simple password and doesn't have the same configuration options as the **KeyStorePass**.
- **ManagerServiceName**: Sets the JMX object name string of the security manager service MBean. This is used to register the defaults to register the **JaasSecurityDomain** as a the security manager under **java:/jaas/<domain>** where **<domain>** is the name passed to the MBean constructor. The name defaults to **jboss.security:service=JaasSecurityManager**.

10.5. DEFINING SECURITY DOMAINS

The standard way of configuring security domains for authentication and authorization in JBoss is to use the XML login configuration file. The login configuration policy defines a set of named security domains that each define a stack of login modules that will be called upon to authenticate and authorize users.

The XML configuration file conforms to the DTD given by [Figure 10.13, “The XMLLoginConfig DTD”](#). This DTD can be found in **docs/dtd/security_config.dtd**.

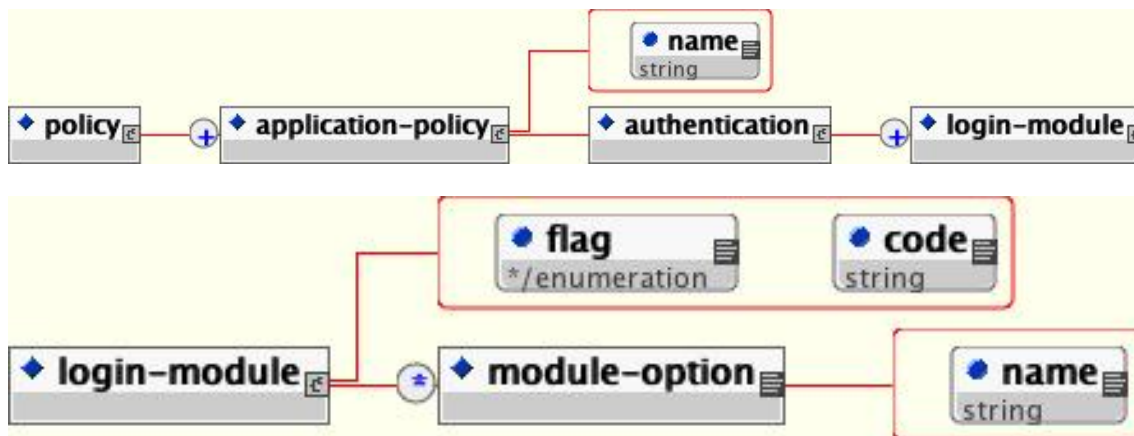


Figure 10.13. The XMLLoginConfig DTD

The following example shows a simple configuration named `jmx-console` that is backed by a single login module. The login module is configured by a simple set of name/value configuration pairs that have meaning to the login module in question. We'll see what these options mean later, for now we'll just be concerned with the structure of the configuration file.

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

The **name** attribute of the **application-policy** is the login configuration name. Applications policy elements will be bound by that name in JNDI under the the `java:/jaas` context. Applications will link to security domains through this JNDI name in their deployment descriptors. (See the **security-domain** elements in `jboss.xml`, `jboss-web.xml` and `jboss-service.xml` files for examples)

The **code** attribute of the **login-module** element specifies the class name of the login module implementation. The **required** flag attribute controls the overall behavior of the authentication stack. The allowed values and meanings are:

- **required**: The login module is required to succeed for the authentication to be successful. If any required module fails, the authentication will fail. The remaining login modules in the stack will be called regardless of the outcome of the authentication.
- **requisite**: The login module is required to succeed. If it succeeds, authentication continues down the login stack. If it fails, control immediately returns to the application.
- **sufficient**: The login module is not required to succeed. If it does succeed, control immediately returns to the application. If it fails, authentication continues down the login stack.
- **optional**: The login module is not required to succeed. Authentication still continues to proceed down the login stack regardless of whether the login module succeeds or fails.

The following example shows the definition of a security domain that uses multiple login modules. Since both modules are marked as sufficient, only one of them need to succeed for login to proceed.

```

<application-policy name="todo">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                  flag="sufficient">
      <!-- LDAP configuration -->
    </login-module>
    <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                  flag="sufficient">
      <!-- database configuration -->
    </login-module>
  </authentication>
</application-policy>

```

Each login module has its own set of configuration options. These are set as name/value pairs using the **module-option** elements. We'll cover module options in more depth when we look at the individual login modules available in JBoss AS.

10.5.1. Loading Security Domains

Authentication security domains are configured statically in the **conf/login-config.xml** file. The **XMLLoginConfig** MBean is responsible for loading security configurations from this configurations from a local configuration file. The MBean is defined as shown below.

```

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

```

The MBean supports the following attributes:

- **ConfigURL**: specifies the URL of the XML login configuration file that should be loaded by this MBean on startup. This must be a valid URL string representation.
- **ConfigResource**: specifies the resource name of the XML login configuration file that should be loaded by this MBean on startup. The name is treated as a classpath resource for which a URL is located using the thread context class loader.
- **ValidateDTD**: a flag indicating if the XML configuration should be validated against its DTD. This defaults to true.

The MBean also supports the following operations that allow one to dynamically extend the login configurations at runtime. Note that any operation that attempts to alter login configuration requires a **javax.security.auth.AuthPermission("refreshLoginConfiguration")** when running with a security manager. The **org.jboss.book.security.service.SecurityConfig** service demonstrates how this can be used to add/remove a deployment specific security configuration dynamically.

- **void addAppConfig(String appName, AppConfiguratonEntry[] entries)**: this adds the given login module configuration stack to the current configuration under the given **appName**. This replaces any existing entry under that name.
- **void removeAppConfig(String appName)**: this removes the login module configuration registered under the given **appName**.

- **String[] loadConfig(URL configURL) throws Exception**: this loads one or more login configurations from a URL representing either an XML or legacy Sun login configuration file. Note that all login configurations must be added or none will be added. It returns the names of the login configurations that were added.
- **void removeConfigs(String[] appNames)**: this removes the login configurations specified **appNames** array.
- **String displayAppConfig(String appName)**: this operation displays a simple string format of the named configuration if it exists.

The **SecurityConfig** MBean is responsible for selecting the **javax.security.auth.login.Configuration** to be used. The default configuration simply references the **XMLLoginConfig** MBean.

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

There is one configurable attribute:

- **LoginConfig**: Specifies the JMX **ObjectName** string of the MBean that provides the default JAAS login configuration. When the **SecurityConfig** is started, this MBean is queried for its **javax.security.auth.login.Configuration** by calling its **getConfiguration(Configuration currentConfig)** operation. If the **LoginConfig** attribute is not specified then the default Sun **Configuration** implementation described in the **Configuration** class JavaDocs is used.

In addition to allowing for a custom JAAS login configuration implementation, this service allows configurations to be chained together in a stack at runtime. This allows one to push a login configuration onto the stack and latter pop it. This is a feature used by the security unit tests to install custom login configurations into a default JBoss installation. Pushing a new configuration is done using:

```
public void pushLoginConfig(String objectName) throws
    JMException, MalformedObjectNameException;
```

The **objectName** parameters specifies an MBean similar to the **LoginConfig** attribute. The current login configuration may be removed using:

```
public void popLoginConfig() throws JMException;
```

10.5.2. The DynamicLoginConfig service

Security domains defined in the **login-config.xml** file are essentially static. They are read when JBoss starts up, but there is no easy way to add a new security domain or change the definition for an existing one. The **DynamicLoginConfig** service allows you to dynamically deploy security domains. This allows you to specify JAAS login configuration as part of a deployment (or just as a standalone service) rather than having to edit the static **login-config.xml** file.

The service supports the following attributes:

- **AuthConfig**: The resource path to the JAAS login configuration file to use. This defaults to `login-config.xml`
- **LoginConfigService**: the `XMLLoginConfig` service name to use for loading. This service must support a `String loadConfig(URL)` operation to load the configurations.
- **SecurityManagerService**: The `SecurityManagerService` name used to flush the registered security domains. This service must support a `flushAuthenticationCache(String)` operation to flush the case for the argument security domain. Setting this triggers the flush of the authentication caches when the service is stopped.

Here is an example MBean definition using the `DynamicLoginConfig` service.

```
<server>
  <mbean code="org.jboss.security.auth.login.DynamicLoginConfig"
name="...">
    <attribute name="AuthConfig">login-config.xml</attribute>

    <!-- The service which supports dynamic processing of login-
config.xml
    configurations.
    -->
    <depends optional-attribute-name="LoginConfigService">
      jboss.security:service=XMLLoginConfig </depends>

    <!-- Optionally specify the security mgr service to use when
    this service is stopped to flush the auth caches of the domains
    registered by this service.
    -->
    <depends optional-attribute-name="SecurityManagerService">
      jboss.security:service=JaasSecurityManager </depends>
  </mbean>
</server>
```

This will load the specified **AuthConfig** resource using the specified **LoginConfigService** MBean by invoking `loadConfig` with the appropriate resource URL. When the service is stopped the configurations are removed. The resource specified may be either an XML file, or a Sun JAAS login configuration.

10.5.3. Using JBoss Login Modules

JBoss includes several bundled login modules suitable for most user management needs. JBoss can read user information from a relational database, an LDAP server or flat files. In addition to these core login modules, JBoss provides several other login modules that provide user information for very customized needs in JBoss. Before we explore the individual login modules, let's take a look at a few login module configuration options that are common to multiple modules.

10.5.3.1. Password Stacking

Multiple login modules can be chained together in a stack, with each login module providing both the authentication and authorization components. This works for many use cases, but sometimes authentication and authorization are split across multiple user management stores. A previous example showed how to combine LDAP and a relational database, allowing a user to be authenticated by either

system. However, consider the case where users are managed in a central LDAP server but application-specific roles are stored in the application's relational database. The password-stacking module option captures this relationship.

- **password-stacking**: When **password-stacking** option is set to **useFirstPass**, 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.

To use password stacking, each login module should set **password-stacking** to **useFirstPass**. If a previous module configured for password stacking has authenticated the user, all the other stacking modules will consider the user authenticated and only attempt to provide a set of roles for the authorization step.

The following listing shows how password stacking could be used:

```
<application-policy name="todo">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                  flag="required">
      <!-- LDAP configuration -->
      <module-option name="password-stacking">useFirstPass</module-
option>
    </login-module>
    <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                  flag="required">
      <!-- database configuration -->
      <module-option name="password-stacking">useFirstPass</module-
option>
    </login-module>
  </authentication>
</application-policy>
```

When using password stacking, it is usually appropriate to set all modules to be required to make sure that all modules are considered and have chance to contribute roles to the authorization process.

10.5.3.2. Password Hashing

Most of the login modules need to compare a client-supplied password to a password stored in a user management system. These modules generally work with plain text passwords, but can also be configured to support hashed passwords to prevent plain text passwords from being stored on the server side.

- **hashAlgorithm**: 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. Typical values are **MD5** and **SHA**.
- **hashEncoding**: The string format for the hashed pass and must be either **base64**, **hex** or **rfc2617**. The default is **base64**.

- **hashCharset**: The encoding used to convert the clear text password to a byte array. The platform default encoding is the default.
- **hashUserPassword**: This indicates that the hashing algorithm should be applied to the password the user submits. The hashed user password will be compared against the value in the login module, which is expected to be a hash of the password. The default is true.
- **hashStorePassword**: This indicates that the hashing algorithm should be applied to the password stored on the server side. This is used for digest authentication where the user submits a hash of the user password along with a request-specific tokens from the server to be compared. JBoss uses the hash algorithm (for digest, this would be **rfc2617**) to compute a server-side hash that should match the hashed value sent from the client.

The following is an login module configuration that assigns unauthenticated users the principal name **nobody** and contains based64-encoded, MD5 hashes of the passwords in a **usersb64.properties** file.

```
<policy>
  <application-policy name="testUsersRoles">
    <authentication>
      <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
          flag="required">
        <module-option name="hashAlgorithm">MD5</module-option>
        <module-option name="hashEncoding">base64</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

If you need to generate passwords in code, the **org.jboss.security.Util** class provides a static helper method that will hash a password using a given encoding.

```
String hashedPassword = Util.createPasswordHash("MD5",
                                                Util.BASE64_ENCODING,
                                                null,
                                                null,
                                                "password");
```

OpenSSL provides an alternative way to quickly generate hashed passwords.

```
echo -n password | openssl dgst -md5 -binary | openssl base64
```

In both cases, the text password should hash to "X03MO1qnZdYdgyfeulLPmQ==". This is the value that would need to be stored in the user store.

10.5.3.3. Unauthenticated Identity

Not all requests come in authenticated. The unauthenticated identity is a login module configuration option that assigns a specific identity (guest, for example) to requests that are made with no associated 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.

- **unauthenticatedIdentity**: This defines the principal name that should be assigned to requests that contain no authentication information.

10.5.3.4. UsersRolesLoginModule

The **UsersRolesLoginModule** is a simple login module that supports multiple users and user roles loaded from Java properties 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:

- **usersProperties**: The name of the properties resource containing the username to password mappings. This defaults to **users.properties**.
- **rolesProperties**: The name of the properties resource containing the username to roles mappings. This defaults to **roles.properties**.

This login module supports password stacking, password hashing and unauthenticated identity.

10.5.3.5. LdapLoginModule

The **LdapLoginModule** is a **LoginModule** implementation that authenticates against an LDAP server. You would use the **LdapLoginModule** if your username and credentials are stored 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**: A prefix to add to the username to form the user distinguished name. See **principalDNSuffix** for more info.
- **principalDNSuffix**: 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 **principalDNPrefix** the **userDN** will be formed as **principalDNPrefix + username + principalDNSuffix**
- **useObjectCredential**: A true/false value that 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**: The fixed distinguished name to the context to search for user roles.
- **userRolesCtxDNAttributeName**: The name of an attribute in the user object that contains the distinguished name to the context to search for user roles. This differs from **rolesCtxDN** in that the context to search for a user's roles can be unique for each user.
- **roleAttributeID**: The name of the attribute that contains the user roles. If not specified this defaults to **roles**.
- **roleAttributeIsDN**: A flag indicating whether the **roleAttributeID** contains the fully distinguished name of a role object, or the role name. If false, the role name is taken from the value of **roleAttributeID**. If true, the role attribute represents the distinguished name of a role object. The role name is taken from the value of the **roleNameAttributeID** attribute of the context name by the distinguished name. In certain directory schemas (e.g., MS ActiveDirectory), role attributes in the user object are stored as DNs to role objects instead of as simple names, in which case, this property should be set to true. The default is false.
- **roleNameAttributeID**: The name of the attribute of the context pointed to by the **roleCtxDN** distinguished name value which contains the role name. If the **roleAttributeIsDN** property is set to true, this property is used to find the role object's name attribute. The default is **group**.

- **uidAttributeID**: 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**: A true/false 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**: The principal name that should be assigned to requests that contain no authentication information. This behavior is inherited from the **UsernamePasswordLoginModule** superclass.
- **allowEmptyPasswords**: A flag indicating if empty (length 0) passwords should be passed to the LDAP server. An empty password is treated as an anonymous login by some LDAP servers and this may not be a desirable feature. Set this to false to reject empty passwords or true to have the LDAP server validate the empty password. The default is true.

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.

The following is a sample **login-config.xml** entry.

```
<application-policy name="testLDAP">
  <authentication>
    <login-module
code="org.jboss.security.auth.spi.LdapLoginModule"
      flag="required">
      <module-option name="java.naming.factory.initial">
        com.sun.jndi.ldap.LdapCtxFactory
      </module-option>
      <module-option name="java.naming.provider.url">
        ldap://ldaphost.jboss.org:1389/
      </module-option>
      <module-option
name="java.naming.security.authentication">
        simple
      </module-option>
      <module-option name="principalDNPrefix">uid=</module-
option>
      <module-option name="principalDNSuffix">
        ,ou=People,dc=jboss,dc=org
      </module-option>
      <module-option name="rolesCtxDN">
        ou=Roles,dc=jboss,dc=org
      </module-option>
```

```

        <module-option name="uidAttributeID">member</module-
option>

        <module-option name="matchOnUserDN">true</module-option>

        <module-option name="roleAttributeID">cn</module-option>
        <module-option name="roleAttributeIsDN">>false </module-
option>

        </login-module>
    </authentication>
</application-policy>

```

An LDIF file representing the structure of the directory this data operates against is shown below.

```

dn: dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: jduke
cn: Java Duke
sn: Duke
userPassword: theduke

dn: ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: Roles

dn: cn=JBossAdmin,ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: groupOfNames
cn: JBossAdmin
member: uid=jduke,ou=People,dc=jboss,dc=org
description: the JBossAdmin group

```

Looking back at the **testLDAP** login module configuration, 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 the LDAP simple authentication method will be use to connect to the LDAP server.

The login module attempts to connect to the LDAP server using a DN representing the user it is trying to authenticate. This DN is constructed from the **principalDNPrefix**, passed in, the username of the user and the **principalDNSuffix** as described above. In this example, the username **jduke** would map to **uid=jduke,ou=People,dc=jboss,dc=org**. We've assumed the LDAP server authenticates

users using the **userPassword** attribute of the user's entry (**theduke** in this example). This is the way most LDAP servers work, however, if your LDAP server handles authentication differently you will need to set the authentication credentials in a way that makes sense for your server.

Once authentication succeeds, the roles on which authorization will be based are retrieved by performing a subtree search of the **rolesCtxDN** for entries whose **uidAttributeID** match the user. If **matchOnUserDN** is true the search will be based on the full DN of the user. Otherwise the search will be based on the actual user name entered. In this example, the search is under **ou=Roles,dc=jboss,dc=org** for any entries that have a **member** attribute equal to **uid=jduke,ou=People,dc=jboss,dc=org**. The search would locate **cn=JBossAdmin** under the roles entry.

The search returns the attribute specified in the **roleAttributeID** option. In this example, the attribute is **cn**. The value returned would be **JBossAdmin**, so the **jduke** user is assigned to the **JBossAdmin** role.

It's often the case that a local LDAP server provides identity and authentication services but is unable to use the authorization services. This is because application roles don't always map well onto LDAP groups, and LDAP administrators are often hesitant to allow external application-specific data in central LDAP servers. For this reason, the LDAP authentication module is often paired with another login module, such as the database login module, that can provide roles more suitable to the application being developed.

This login module also supports unauthenticated identity and password stacking.

10.5.3.6. LdapExtLoginModule

Distinguished Name (DN)

In Lightweight Directory Access Protocol (LDAP), the distinguished name uniquely identifies an object in a directory. Each distinguished name must have a unique name and location from all other objects, which is achieved using a number of attribute-value pairs (AVPs). The AVPs define information such as common names, organization unit, among others. The combination of these values results in a unique string required by the LDAP.

The **org.jboss.security.auth.spi.LdapExtLoginModule** searches for the user to bind, as well as the associated roles, for authentication. The roles query recursively follows DN's to navigate a hierarchical role structure.

The LoginModule options include whatever options are supported by the chosen LDAP JNDI provider supports. Examples of standard property names are:

- **Context.INITIAL_CONTEXT_FACTORY** = "java.naming.factory.initial"
- **Context.SECURITY_PROTOCOL** = "java.naming.security.protocol"
- **Context.PROVIDER_URL** = "java.naming.provider.url"
- **Context.SECURITY_AUTHENTICATION** = "java.naming.security.authentication"
- **Context.REFERRAL** = "java.naming.referral"

Login module implementation logic follows the order below:

1. The initial LDAP server bind is authenticated using the **bindDN** and **bindCredential** properties. The **bindDN** is a user with permissions to search both the **baseCtxDN** and **rolesCtxDN** trees for the user and roles. The user DN to authenticate against is queried using the filter specified by the

baseFilter property.

2. The resulting userDN is authenticated by binding to the LDAP server using the userDN as the InitialLdapContext environment Context.SECURITY_PRINCIPAL. The Context.SECURITY_CREDENTIALS property is either set to the String password obtained by the callback handler.
3. If this is successful, the associated user roles are queried using the rolesCtxDN, roleAttributeID, roleAttributesDN, roleNameAttributeID, and roleFilter options.

LdapExtLoginModule Properties

baseCtxDN

Specifies the fixed DN of the context to start the user search from.

bindDN

Specifies the DN used to bind against the LDAP server for the user and role queries. Set bindDN to a DN with read/search permissions on the baseCtxDN and rolesCtxDn properties.

bindCredential

The password for the bindDN. bindCredential can be encrypted if jaasSecurityDomain is specified. This property allows an external command to read the password. For example **{EXT}cat file_with_password**.

jaasSecurityDomain

The JMX ObjectName of the JaasSecurityDomain used to decrypt the **java.naming.security.principal**. The encrypted form of the password is that returned by the **JaasSecurityDomainencrypt64(byte[])** method. The **org.jboss.security.plugins.PBEUtils** can also be used to generate the encrypted form.

baseFilter

A search filter used to locate the context of the user to authenticate. The input username/userDN as obtained from the login module callback is substituted into the filter anywhere a **{0}** expression exists. This substitution behavior originates from the standard **DirContext.search(Name, String, Object[], SearchControls cons)** method. An common example search filter is **(uid={0})**.

rolesCtxDN

The fixed DN of the context to search for user roles. This is not the DN of where the actual roles are; this is the DN of where the objects containing the user roles are. For example, in active directory, this is the DN where the user account is.

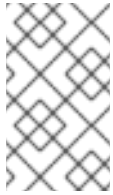
roleFilter

Search filter used to locate the roles associated with the authenticated user. The input username/userDN as obtained from the login module callback is substituted into the filter anywhere a **{0}** expression exists. The authenticated userDN is substituted into the filter anywhere a **{1}** is seen. An example search filter that matches on the input username is **(member={0})**. An alternative that matches on the authenticated userDN is **(member={1})**.

roleAttributesDN

Flag indicating whether the `roleAttributeID` contains the full DN of a role object, or the role name. The role name is derived from the value of the `roleNameAttributeID` attribute of the context name by the distinguished name.

If set to **true**, the role attribute represents the distinguished name of a role object. If set to **false**, the role name is taken from the value of **`roleAttributeID`**. The default is **false**.



NOTE

In certain directory schemas (e.g., MS ActiveDirectory), role attributes in the user object are stored as DNs to role objects instead of simple names. For implementations that use this schema type, `roleAttributesDN` must be set to **true**.

roleAttributeID

Name of the attribute containing the user roles. If `roleAttributesDN` is set to **true**, this property is the DN of the context to query for the `roleNameAttributeID` attribute. If the `roleAttributesDN` property is set to **false**, this property is the attribute name of the role name.

roleNameAttributeID

Name of the attribute of the context pointed to by the `roleCtxDN` distinguished name value which contains the role name. If the `roleAttributesDN` property is set to **true**, this property is used to find the role object's **name** attribute. The default is **group**.

roleRecursion

Specifies how many levels the role search traverses a given matching context. The default is **0** (deactivated).

searchTimeLimit

The timeout in milliseconds for the user/role searches. Defaults to 10000 (10 seconds).

searchScope

Sets the search scope to one of the strings. The default is `SUBTREE_SCOPE`. Other supported values include:

- `OBJECT_SCOPE` : only search the named roles context.
- `ONELEVEL_SCOPE` : search directly under the named roles context.
- `SUBTREE_SCOPE` : If the roles context is not a `DirContext`, search only the object. If the roles context is a `DirContext`, search the subtree rooted at the named object, including the named object itself.

allowEmptyPasswords

A flag indicating if **`empty(length==0)`** passwords should be passed to the LDAP server.

An empty password is treated as an anonymous login by some LDAP servers. If set to **false**, empty passwords are rejected. If set to **true**, the LDAP server validates the empty password. The default is **true**.

defaultRole

A role included for all authenticated users.

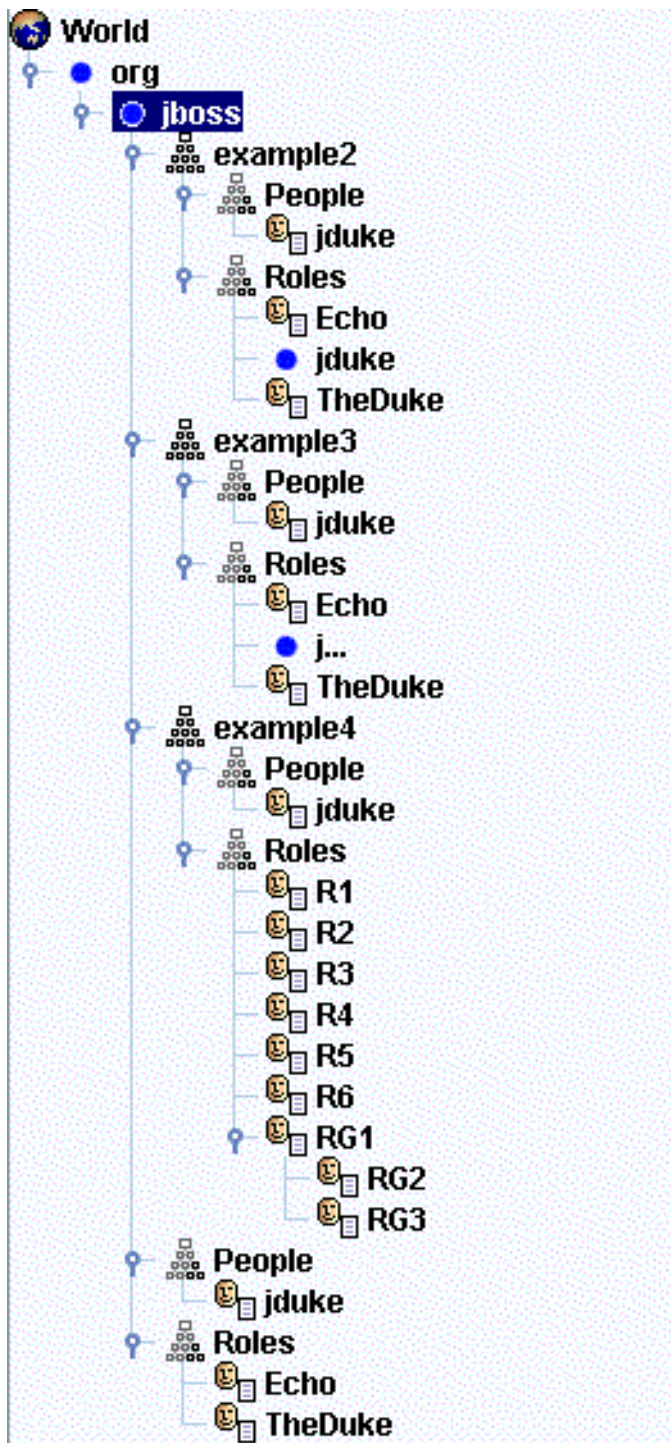


Figure 10.14. LDAP Structure Example

Example 10.9. Example 2 LDAP Configuration

```

version: 1
dn: o=example2,dc=jboss,dc=org
objectClass: top
objectClass: dcObject
objectClass: organization
dc: jboss
o: JBoss
  
```

```
dn: ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke
employeeNumber: judke-123
sn: Duke
uid: jduke
userPassword:: dGhlZHVRZQ==

dn: uid=jduke2,ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke2
employeeNumber: judke2-123
sn: Duke2
uid: jduke2
userPassword:: dGhlZHVRZTI=

dn: ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: uid=jduke,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo,ou=Roles,o=example2,dc=jboss,dc=org
memberOf: cn=TheDuke,ou=Roles,o=example2,dc=jboss,dc=org
uid: jduke

dn: uid=jduke2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo2,ou=Roles,o=example2,dc=jboss,dc=org
memberOf: cn=TheDuke2,ou=Roles,o=example2,dc=jboss,dc=org
uid: jduke2

dn: cn=Echo,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo
description: the echo role
member: uid=jduke,ou=People,dc=jboss,dc=org

dn: cn=TheDuke,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
```

```

cn: TheDuke
description: the duke role
member: uid=jduke,ou=People,o=example2,dc=jboss,dc=org

dn: cn=Echo2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo2
description: the Echo2 role
member: uid=jduke2,ou=People,dc=jboss,dc=org

dn: cn=TheDuke2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke2
description: the duke2 role
member: uid=jduke2,ou=People,o=example2,dc=jboss,dc=org

dn: cn=JBossAdmin,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: JBossAdmin
description: the JBossAdmin group
member: uid=jduke,ou=People,dc=jboss,dc=org

```

The module configuration for this LDAP structure example is outlined in the code sample.

```

testLdapExample2 {
    org.jboss.security.auth.spi.LdapExtLoginModule
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url="ldap://lamia/"
        java.naming.security.authentication=simple
        bindDN="cn=Root,dc=jboss,dc=org"
        bindCredential=secret1
        baseCtxDN="ou=People,o=example2,dc=jboss,dc=org"
        baseFilter="(uid={0})"
        rolesCtxDN="ou=Roles,o=example2,dc=jboss,dc=org";
        roleFilter="(uid={0})"
        roleAttributeIsDN="true"
        roleAttributeID="memberOf"
        roleNameAttributeID="cn"
};

```

Example 10.10. Example 3 LDAP Configuration

```

dn: o=example3,dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,o=example3,dc=jboss,dc=org
objectclass: top

```

```
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example3,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
objectClass: inetOrgPerson
uid: jduke
employeeNumber: judke-123
cn: Java Duke
sn: Duke
userPassword: theduke

dn: ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: uid=jduke,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo,ou=Roles,o=example3,dc=jboss,dc=org
memberOf: cn=TheDuke,ou=Roles,o=example3,dc=jboss,dc=org
uid: jduke

dn: cn=Echo,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo
description: the JBossAdmin group
member: uid=jduke,ou=People,o=example3,dc=jboss,dc=org

dn: cn=TheDuke,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke
member: uid=jduke,ou=People,o=example3,dc=jboss,dc=org
```

The module configuration for this LDAP structure example is outlined in the code sample.

```
testLdapExample3 {
    org.jboss.security.auth.spi.LdapExtLoginModule
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url="ldap://lamia/"
        java.naming.security.authentication=simple
        bindDN="cn=Root,dc=jboss,dc=org"
        bindCredential=secret1
        baseCtxDN="ou=People,o=example3,dc=jboss,dc=org"
        baseFilter="(cn={0})"
        rolesCtxDN="ou=Roles,o=example3,dc=jboss,dc=org";
        roleFilter="(member={1})"
        roleAttributeID="cn"
};
```

Example 10.11. Example 4 LDAP Configuration

```

dn: o=example4,dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,o=example4,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example4,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke
employeeNumber: jduke-123
sn: Duke
uid: jduke
userPassword:: dGhlZHVrZQ==

dn: ou=Roles,o=example4,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG1
member: cn=empty

dn: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG2
member: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: uid=jduke,ou=People,o=example4,dc=jboss,dc=org

dn: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG3
member: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R1
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R2,ou=Roles,o=example4,dc=jboss,dc=org

```

```

objectClass: groupOfNames
objectClass: top
cn: R2
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R3,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R3
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R4,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R4
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R5,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R5
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: uid=jduke,ou=People,o=example4,dc=jboss,dc=org

```

The module configuration for this LDAP structure example is outlined in the code sample.

```

testLdapExample4 {
    org.jboss.security.auth.spi.LdapExtLoginModule
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url="ldap://lamia/"
        java.naming.security.authentication=simple
        bindDN="cn=Root,dc=jboss,dc=org"
        bindCredential=secret1
        baseCtxDN="ou=People,o=example4,dc=jboss,dc=org"
        baseFilter="(cn={0})"
        rolesCtxDN="ou=Roles,o=example4,dc=jboss,dc=org";
        roleFilter="(member={1})"
        roleAttributeID="memberOf"
};

```

Example 10.12. Default ActiveDirectory Configuration

The example below is represents the configuration for a default Active Directory configuration.

```

<?xml version="1.0" encoding="UTF-8"?>
<application-policy name="AD_Default">
    <authentication>
        <login-module
code="org.jboss.security.auth.spi.LdapExtLoginModule" flag="required" >
        <!--
Some AD configurations may require searching against
the Global Catalog on port 3268 instead of the usual
port 389. This is most likely when the AD forest

```



```

        includes multiple domains.
    -->
    <module-option
name="java.naming.provider.url">ldap://ldap.jboss.org:389</module-
option>
        <module-option name="bindDN">JBoss\someadmin</module-option>
        <module-option name="bindCredential">password</module-option>
        <module-option
name="baseCtxDN">cn=Users,dc=jboss,dc=org</module-option>
        <module-option name="baseFilter">(sAMAccountName={0})</module-
option>
        <module-option
name="rolesCtxDN">cn=Users,dc=jboss,dc=org</module-option>
        <module-option name="roleFilter">(sAMAccountName={0})</module-
option>
        <module-option name="roleAttributeID">memberOf</module-option>
        <module-option name="roleAttributeIsDN">true</module-option>
        <module-option name="roleNameAttributeID">cn</module-option>
        <module-option name="searchScope">ONELEVEL_SCOPE</module-
option>
        <module-option name="allowEmptyPasswords">>false</module-
option>
    </login-module>
</authentication>
</application-policy>

```

Example 10.13. Recursive Roles ActiveDirectory Configuration

The example below implements a recursive role search within ActiveDirectory. The key difference between [Example 10.12, "Default ActiveDirectory Configuration"](#) is the role search has been replaced to search the member attribute using the DN of the user. The login module then uses the DN of the role to find groups that the group is a member of.

```

<?xml version="1.0" encoding="UTF-8"?>
<application-policy name="AD_Recursive">
    <authentication>
        <login-module
code="org.jboss.security.auth.spi.LdapExtLoginModule" flag="required" >
            <module-option
name="java.naming.provider.url">ldap://ad.jboss.org:389</module-option>
            <module-option name="bindDN">JBoss\searchuser</module-option>
            <module-option name="bindCredential">password</module-option>
            <module-option
name="baseCtxDN">CN=Users,DC=jboss,DC=org</module-option>
            <module-option name="baseFilter">(sAMAccountName={0})</module-
option>
            <module-option
name="rolesCtxDN">CN=Users,DC=jboss,DC=org</module-option>
            <module-option name="roleFilter">(member={1})</module-option>
            <module-option name="roleAttributeID">cn</module-option>
            <module-option name="roleAttributeIsDN">>false</module-option>
            <module-option name="roleRecursion">2</module-option>
            <module-option name="searchScope">ONELEVEL_SCOPE</module-
option>

```

```

        <module-option name="allowEmptyPasswords">false</module-
option>
        <module-option name="java.naming.referral">follow</module-
option>
    </login-module>
</authentication>
</application-policy>

```

10.5.3.7. 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 relational 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 **PrincipalIDjava** with a **Password** of **echoman** in the **Principals** table, a **PrincipalIDjava** with a role named **Echo** in the **RolesRoleGroup** in the **Roles** table, and a **PrincipalIDjava** with a role named **caller_java** in the **CallerPrincipalRoleGroup** 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.
- **ignorePasswordCase**: A boolean flag indicating if the password comparison should ignore case. This can be useful for hashed password encoding where the case of the hashed password is not significant.
- **principalClass**: An option that specifies a **Principal** implementation class. This must support a constructor taking a string argument for the principal name.

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))
```

A corresponding **login-config.xml** entry would be:

```
<policy>
  <application-policy name="testDB">
    <authentication>
      <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
      flag="required">
        <module-option
name="dsJndiName">java:/MyDatabaseDS</module-option>
        <module-option name="principalsQuery">
          select passwd from Users username where username=?
        </module-option>
        <module-option name="rolesQuery">
          select userRoles, 'Roles' from UserRoles where
username=?</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

This module supports password stacking, password hashing and unauthenticated identity.

10.5.3.8. BaseCertLoginModule

This is a login module which authenticates users based on X509 certificates. A typical use case for this login module is **CLIENT-CERT** authentication in the web tier. This login module only performs authentication. You need to combine it with another login module capable of acquiring the authorization roles to completely define access to a secured web or EJB component. Two subclasses of this login module, **CertRolesLoginModule** and **DatabaseCertLoginModule** extend the behavior to obtain the authorization roles from either a properties file or database.

The **BaseCertLoginModule** needs a **KeyStore** to perform user validation. This is obtained through a **org.jboss.security.SecurityDomain** implementation. Typically, the **SecurityDomain** implementation is configured using the **org.jboss.security.plugins.JaasSecurityDomain** MBean as shown in this **jboss-service.xml** configuration fragment:

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
  name="jboss.ch8:service=SecurityDomain">
  <constructor>
    <arg type="java.lang.String" value="jmx-console"/>
  </constructor>
  <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
  <attribute name="KeyStorePass">unit-tests-server</attribute>
</mbean>
```

This creates a security domain with the name **jmx-console** whose **SecurityDomain** implementation is available via JNDI under the name **java:/jaas/jmx-console** following the JBossSX security

domain naming pattern. To secure a web application such as the **jmx-console.war** using client certs and role based authorization, one would first modify the **web.xml** to declare the resources to be secured, along with the allowed roles and security domain to be used for authentication and authorization.

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    ...
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HtmlAdaptor</web-resource-name>
            <description>An example security config that only allows users
with
                the role JBossAdmin to access the HTML JMX console web
                application </description>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>JBossAdmin</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
        <realm-name>JBoss JMX Console</realm-name>
    </login-config>
    <security-role>
        <role-name>JBossAdmin</role-name>
    </security-role>
</web-app>
```

Next we, need to specify the JBoss security domain in **jboss-web.xml**:

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

Finally, you need to define the login module configuration for the jmx-console security domain you just specified. This is done in the **conf/login-config.xml** file.

```
<application-policy name="jmx-console">
    <authentication>
        <login-module
code="org.jboss.security.auth.spi.BaseCertLoginModule"
            flag="required">
            <module-option name="password-stacking">useFirstPass</module-
option>
            <module-option name="securityDomain">java:/jaas/jmx-
console</module-option>
        </login-module>
    </authentication>
</application-policy>
```

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

Here the **BaseCertLoginModule** is used for authentication of the client cert, and the **UsersRolesLoginModule** is only used for authorization due to the **password-stacking=useFirstPass** option. Both the **localhost.keystore** and the **jmx-console-roles.properties** need an entry that maps to the principal associated with the client cert. By default, the principal is created using the client certificate distinguished name. Consider the following certificate:

```
[starksm@banshee9100 conf]$ keytool -printcert -file unit-tests-
client.export
Owner: CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington,
C=US
Issuer: CN=jboss.com, C=US, ST=Washington, L=Snoqualmie Pass,
EMAILADDRESS=admin
@jboss.com, OU=QA, O=JBoss Inc.
Serial number: 100103
Valid from: Wed May 26 07:34:34 PDT 2004 until: Thu May 26 07:34:34 PDT
2005
Certificate fingerprints:
    MD5: 4A:9C:2B:CD:1B:50:AA:85:DD:89:F6:1D:F5:AF:9E:AB
    SHA1: DE:DE:86:59:05:6C:00:E8:CC:C0:16:D3:C2:68:BF:95:B8:83:E9:58
```

The **localhost.keystore** would need this cert stored with an alias of **CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US** and the **jmx-console-roles.properties** would also need an entry for the same entry. Since the DN contains many characters that are normally treated as delimiters, you will need to escape the problem characters using a backslash ('\') as shown here:

```
# A sample roles.properties file for use with the UsersRolesLoginModule
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\
ST\=Washington,\ C\=US=JBossAdmin
admin=JBossAdmin
```

10.5.3.9. IdentityLoginModule

The **IdentityLoginModule** is a simple login module that associates a hard-coded user name to any subject authenticated against the module. It creates a **SimplePrincipal** instance using the name specified by the **principal** option. This login module is useful when you need to provide a fixed identity to a service and 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**: This is 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**: This is a comma-delimited list of roles that will be assigned to the user.

A sample XMLLoginConfig configuration entry that would authenticate all users as the principal named **jduke** and assign role names of **TheDuke**, and **AnimatedCharacter** is:

```
<policy>
  <application-policy name="testIdentity">
    <authentication>
      <login-module
code="org.jboss.security.auth.spi.IdentityLoginModule"
      flag="required">
        <module-option name="principal">jduke</module-option>
        <module-option
name="roles">TheDuke, AnimatedCharater</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

This module supports password stacking.

10.5.3.10. RunAsLoginModule

JBoss has a helper login module called **RunAsLoginModule** that pushes a run as role for the duration of the login phase of authentication, and pops the run as role in either the commit or abort phase. The purpose of this login module is to provide a role for other login modules that need to access secured resources in order to perform their authentication. An example would be a login module that accesses an secured EJB. This login module must be configured ahead of the login module(s) that need a run as role established.

The only login module configuration option is:

- **roleName**: the name of the role to use as the run as role during login phase. If not specified a default of **nobody** is used.

10.5.3.11. 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**: 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**: When **password-stacking** option is set to **useFirstPass**, 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**.
- **restore-login-identity**: When **restore-login-identity** is true, the **SecurityAssociation** principal and credential seen on entry to the **login()** method are saved and restored on either abort or logout. When false (the default), the abort and logout simply clear the **SecurityAssociation**. A **restore-login-identity** of true is needed if one need to change identities and then restore the original caller identity.

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
};
```

10.5.4. 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 in JBoss 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 **SubjectPrincipals** 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 **SubjectPrincipals** 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 **CallerPrincipalGroup** 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 **CallerPrincipalGroup**, the application identity is the same as operational environment identity.

10.5.4.1. Support for the Subject Usage Pattern

To simplify correct implementation of the **Subject** usage patterns described in the preceding section, JBossSX includes two abstract login modules that handle the population of the authenticated **Subject** with a template pattern that enforces correct **Subject** usage. The most generic of the two is the **org.jboss.security.auth.spi.AbstractServerLoginModule** class. It provides a concrete implementation of the **javax.security.auth.spi.LoginModule** interface and offers abstract methods for the key tasks specific to an operation environment security infrastructure. The key details of the class are highlighted in the following class fragment. The JavaDoc comments detail the responsibilities of subclasses.

```

package org.jboss.security.auth.spi;
/**
 * This class implements the common functionality required for a JAAS
 * server-side LoginModule and implements the JBossSX standard
 * Subject usage pattern of storing identities and roles. Subclass
 * this module to create your own custom LoginModule and override the
 * login(), getRoleSets(), and getIdentity() methods.
 */
public abstract class AbstractServerLoginModule
    implements javax.security.auth.spi.LoginModule
{

```



```

protected Subject subject;
protected CallbackHandler callbackHandler;
protected Map sharedState;
protected Map options;
protected Logger log;

/** Flag indicating if the shared credential should be used */
protected boolean useFirstPass;
/**
 * Flag indicating if the login phase succeeded. Subclasses that
 * override the login method must set this to true on successful
 * completion of login
 */
protected boolean loginOk;

// ...
/**
 * Initialize the login module. This stores the subject,
 * callbackHandler and sharedState and options for the login
 * session. Subclasses should override if they need to process
 * their own options. A call to super.initialize(...) must be
 * made in the case of an override.
 *
 * <p>
 * The options are checked for the <em>password-stacking</em>
parameter.
 * If this is set to "useFirstPass", the login identity will be taken
from the
 * <code>javax.security.auth.login.name</code> value of the
sharedState map,
 * and the proof of identity from the
 * <code>javax.security.auth.login.password</code> value of the
sharedState map.
 *
 * @param subject the Subject to update after a successful login.
 * @param callbackHandler the CallbackHandler that will be used to
obtain the
 * the user identity and credentials.
 * @param sharedState a Map shared between all configured login module
instances
 * @param options the parameters passed to the login module.
 */
public void initialize(Subject subject,
                      CallbackHandler callbackHandler,
                      Map sharedState,
                      Map options)
{
    // ...
}

/**
 * Looks for javax.security.auth.login.name and
 * javax.security.auth.login.password values in the sharedState
 * map if the useFirstPass option was true and returns true if
 * they exist. If they do not or are null this method returns

```

```

    * false.
    * Note that subclasses that override the login method
    * must set the loginOk var to true if the login succeeds in
    * order for the commit phase to populate the Subject. This
    * implementation sets loginOk to true if the login() method
    * returns true, otherwise, it sets loginOk to false.
    */
    public boolean login()
        throws LoginException
    {
        // ...
    }

    /**
     * Overridden by subclasses to return the Principal that
     * corresponds to the user primary identity.
     */
    abstract protected Principal getIdentity();

    /**
     * Overridden by subclasses to return the Groups that correspond
     * to the role sets assigned to the user. Subclasses should
     * create at least a Group named "Roles" that contains the roles
     * assigned to the user. A second common group is
     * "CallerPrincipal," which provides the application identity of
     * the user rather than the security domain identity.
     *
     * @return Group[] containing the sets of roles
     */
    abstract protected Group[] getRoleSets() throws LoginException;
}

```

You'll need to pay attention to the **loginOk** instance variable. This must be set to true if the login succeeds, false otherwise by any subclasses that override the login method. Failure to set this variable correctly will result in the commit method either not updating the subject when it should, or updating the subject when it should not. Tracking the outcome of the login phase was added to allow login modules to be chained together with control flags that do not require that the login module succeed in order for the overall login to succeed.

The second abstract base login module suitable for custom login modules is the **org.jboss.security.auth.spi.UsernamePasswordLoginModule**. This 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 credentials. It also supports the mapping of anonymous users (indicated by a null username and password) to a principal with no roles. The key details of the class are highlighted in the following class fragment. The JavaDoc comments detail the responsibilities of subclasses.

```

package org.jboss.security.auth.spi;

/**
 * An abstract subclass of AbstractServerLoginModule that imposes a
 * an identity == String username, credentials == String password
 * view on the login process. Subclasses override the
 * getUsersPassword() and getUsersRoles() methods to return the
 * expected password and roles for the user.
 */

```

```

public abstract class UsernamePasswordLoginModule
    extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password are seen */
    private Principal unauthenticatedIdentity;

    /**
     * The message digest algorithm used to hash passwords. If null then
     * plain passwords will be used. */
    private String hashAlgorithm = null;

    /**
     * The name of the charset/encoding to use when converting the
     * password String to a byte array. Default is the platform's
     * default encoding.
     */
    private String hashCharset = null;

    /** The string encoding format to use. Defaults to base64. */
    private String hashEncoding = null;

    // ...

    /**
     * Override the superclass method to look for an
     * unauthenticatedIdentity property. This method first invokes
     * the super version.
     *
     * @param options,
     * @option unauthenticatedIdentity: the name of the principal to
     * assign and authenticate when a null username and password are
     * seen.
     */
    public void initialize(Subject subject,
                          CallbackHandler callbackHandler,
                          Map sharedState,
                          Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
                        options);
        // Check for unauthenticatedIdentity option.
        Object option = options.get("unauthenticatedIdentity");
        String name = (String) option;
        if (name != null) {
            unauthenticatedIdentity = new SimplePrincipal(name);
        }
    }

    // ...

    /**
     * A hook that allows subclasses to change the validation of the

```

```

    * input password against the expected password. This version
    * checks that neither inputPassword or expectedPassword are null
    * and that inputPassword.equals(expectedPassword) is true;
    *
    * @return true if the inputPassword is valid, false otherwise.
    */
protected boolean validatePassword(String inputPassword,
                                   String expectedPassword)
{
    if (inputPassword == null || expectedPassword == null) {
        return false;
    }
    return inputPassword.equals(expectedPassword);
}

/**
 * Get the expected password for the current username available
 * via the getUsername() method. This is called from within the
 * login() method after the CallbackHandler has returned the
 * username and candidate password.
 *
 * @return the valid password String
 */
abstract protected String getUsersPassword()
    throws LoginException;
}

```

The choice of subclassing the **AbstractServerLoginModule** versus **UsernamePasswordLoginModule** is simply based on whether a string-based username and credentials are usable for the authentication technology you are writing the login module for. If the string-based semantic is valid, then subclass **UsernamePasswordLoginModule**, otherwise subclass **AbstractServerLoginModule**.

The steps you are required to perform when writing a custom login module are summarized in the following depending on which base login module class you choose. When writing a custom login module that integrates with your security infrastructure, you should start by subclassing **AbstractServerLoginModule** or **UsernamePasswordLoginModule** to ensure that your login module provides the authenticated **Principal** information in the form expected by the JBossSX security manager.

When subclassing the **AbstractServerLoginModule**, you need to override the following:

- **void initialize(Subject, CallbackHandler, Map, Map):** if you have custom options to parse.
- **boolean login():** to perform the authentication activity. Be sure to set the **loginOk** instance variable to true if login succeeds, false if it fails.
- **Principal getIdentity():** to return the **Principal** object for the user authenticated by the **log()** step.
- **Group[] getRoleSets():** to return at least one **Group** named **Roles** that contains the roles assigned to the **Principal** authenticated during **login()**. A second common **Group** is named **CallerPrincipal** and provides the user's application identity rather than the security domain identity.

When subclassing the **UsernamePasswordLoginModule**, you need to override the following:

- **void initialize(Subject, CallbackHandler, Map, Map):** if you have custom options to parse.
- **Group[] getRoleSets():** to return at least one **Group** named **Roles** that contains the roles assigned to the **Principal** authenticated during **login()**. A second common **Group** is named **CallerPrincipal** and provides the user's application identity rather than the security domain identity.
- **String getUsersPassword():** to return the expected password for the current username available via the **getUsername()** method. The **getUsersPassword()** method is called from within **login()** after the **callbackhandler** returns the username and candidate password.

10.5.4.2. A Custom LoginModule Example

In this section we will develop a custom login module example. It will extend the **UsernamePasswordLoginModule** and obtains a user's password and role names from a JNDI lookup. The idea is that there is a JNDI context that will return a user's password if you perform a lookup on the context using a name of the form **password/<username>** where **<username>** is the current user being authenticated. Similarly, a lookup of the form **roles/<username>** returns the requested user's roles.

The source code for the example is located in the **src/main/org/jboss/book/security/ex2** directory of the book examples. [Example 10.14](#), “[A JndiUserAndPass custom login module](#)” shows the source code for the **JndiUserAndPass** custom login module. Note that because this extends the JBoss **UsernamePasswordLoginModule**, all the **JndiUserAndPass** does is obtain the user's password and roles from the JNDI store. The **JndiUserAndPass** does not concern itself with the JAAS **LoginModule** operations.

Example 10.14. A JndiUserAndPass custom login module

```
package org.jboss.book.security.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/**
 * An example custom login module that obtains passwords and roles
 * for a user from a JNDI lookup.
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.4 $
 */
public class JndiUserAndPass
    extends UsernamePasswordLoginModule
{
```

```

    /** The JNDI name to the context that handles the password/username
lookup */
    private String userPathPrefix;
    /** The JNDI name to the context that handles the roles/ username
lookup */
    private String rolesPathPrefix;

    /**
     * Override to obtain the userPathPrefix and rolesPathPrefix
options.
     */
    public void initialize(Subject subject, CallbackHandler
callbackHandler,
                        Map sharedState, Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
options);
        userPathPrefix = (String) options.get("userPathPrefix");
        rolesPathPrefix = (String) options.get("rolesPathPrefix");
    }

    /**
     * Get the roles the current user belongs to by querying the
     * rolesPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected Group[] getRoleSets() throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String rolesPath = rolesPathPrefix + '/' +
super.getUsername();

            String[] roles = (String[]) ctx.lookup(rolesPath);
            Group[] groups = {new SimpleGroup("Roles")};
            log.info("Getting roles for user="+super.getUsername());
            for(int r = 0; r < roles.length; r++) {
                SimplePrincipal role = new SimplePrincipal(roles[r]);
                log.info("Found role="+roles[r]);
                groups[0].addMember(role);
            }
            return groups;
        } catch(NamingException e) {
            log.error("Failed to obtain groups for
                user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }

    /**
     * Get the password of the current user by querying the
     * userPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected String getUsersPassword()
        throws LoginException
    {
        try {

```

```

        InitialContext ctx = new InitialContext();
        String userPath = userPathPrefix + '/' +
super.getUsername();
        log.info("Getting password for user="+super.getUsername());
        String passwd = (String) ctx.lookup(userPath);
        log.info("Found password="+passwd);
        return passwd;
    } catch(NamingException e) {
        log.error("Failed to obtain password for
            user="+super.getUsername(), e);
        throw new LoginException(e.toString(true));
    }
}
}

```

The details of the JNDI store are found in the

org.jboss.book.security.ex2.service.JndiStore MBean. This service binds an **ObjectFactory** that returns a **javax.naming.Context** proxy into JNDI. The proxy handles lookup operations done against it by checking the prefix of the lookup name against **password** and **roles**. When the name begins with **password**, a user's password is being requested. When the name begins with **roles** the user's roles are being requested. The example implementation always returns a password of **theduke** and an array of roles names equal to **{"TheDuke", "Echo"}** regardless of what the username is. You can experiment with other implementations as you wish.

The example code includes a simple session bean for testing the custom login module. To build, deploy and run the example, execute the following command in the examples directory.

```

[examples]$ ant -Dchap=security -Dex=2 run-example
...
run-example2:
    [echo] Waiting for 5 seconds for deploy...
    [java] [INFO,ExClient] Login with username=jduke, password=theduke
    [java] [INFO,ExClient] Looking up EchoBean2
    [java] [INFO,ExClient] Created Echo
    [java] [INFO,ExClient] Echo.echo('Hello') = Hello

```

The choice of using the **JndiUserAndPass** custom login module for the server side authentication of the user is determined by the login configuration for the example security domain. The EJB JAR **META-INF/jboss.xml** descriptor sets the security domain

```

<?xml version="1.0"?>
<jboss>
    <security-domain>java:/jaas/security-ex2</security-domain>
</jboss>

```

The SAR **META-INF/login-config.xml** descriptor defines the login module configuration.

```

<application-policy name = "security-ex2">
    <authentication>
        <login-module code="org.jboss.book.security.ex2.JndiUserAndPass"
            flag="required">
            <module-option name =
"userPathPrefix">/security/store/password</module-option>

```

```
        <module-option name =  
"rolesPathPrefix">/security/store/roles</module-option>  
        </login-module>  
    </authentication>  
</application-policy>
```

10.6. 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 at <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.

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 10.15, “The JBossSX components of the SRP client-server framework,” gives a diagram of the key components involved in the JBossSX implementation of the SRP client/server framework.

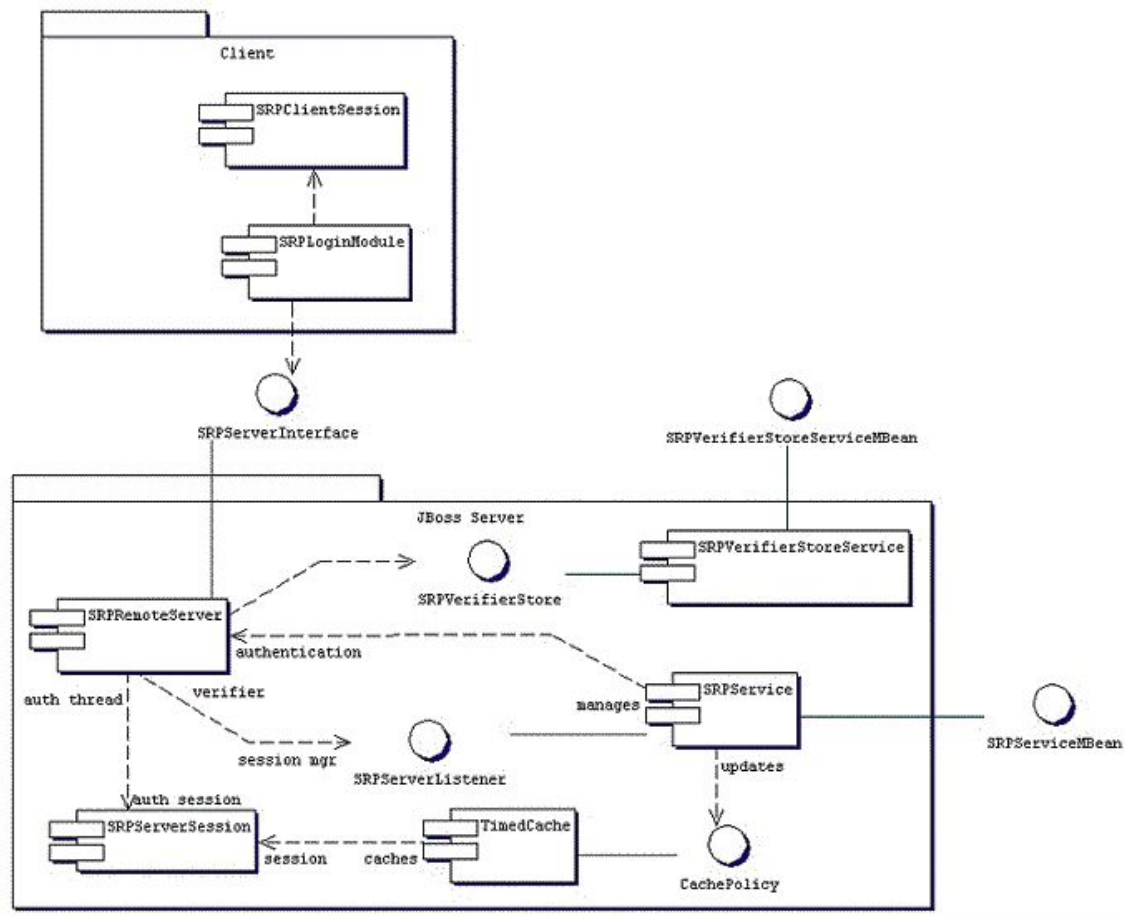


Figure 10.15. The JBossSX components of the SRP client-server framework.

On the client side, SRP shows up as a custom JAAS **LoginModule** implementation that communicates to the authentication server through an **org.jboss.security.srp.SRPServerInterface** proxy. A client enables authentication using SRP by creating a login configuration entry that includes the **org.jboss.security.srp.jaas.SRPLoginModule**. This module supports the following configuration options:

- **principalClassName**: This option is no longer supported. The principal class is now always `org.jboss.security.srp.jaas.SRPPrincipal`.
- **srpServerJndiName**: The JNDI name of the **SRPServerInterface** object to use for communicating with the SRP authentication server. If both **srpServerJndiName** and **srpServerRmiUrl** options are specified, the **srpServerJndiName** is tried before **srpServerRmiUrl**.
- **srpServerRmiUrl**: The RMI protocol URL string for the location of the **SRPServerInterface** proxy to use for communicating with the SRP authentication server.
- **externalRandomA**: A true/false flag indicating if the random component of the client public key A should come from the user callback. This can be used to input a strong cryptographic random number coming from a hardware token for example.
- **hasAuxChallenge**: A true/false flag indicating that a string will be sent to the server as an additional challenge for the server to validate. If the client session supports an encryption cipher then a temporary cipher will be created using the session private key and the challenge object

sent as a `javax.crypto.SealedObject`.

- **multipleSessions**: a true/false flag indicating if a given client may have multiple SRP login sessions active simultaneously.

Any other options passed in that do not match one of the previous named options is treated as a JNDI property to use for the environment passed to the **InitialContext** constructor. This is useful if the SRP server interface is not available from the default **InitialContext**.

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).
- **RequireAuxChallenge**: Set if the client must supply an auxiliary challenge as part of the verify phase. This gives control over whether the **SRPLoginModule** configuration used by the client must have the **useAuxChallenge** option enabled.
- **OverwriteSessions**: A flag indicating if a successful user auth for an existing session should overwrite the current session. This controls the behavior of the server SRP session cache when clients have not enabled the multiple session per user mode. The default is false meaning that the second attempt by a user to authentication will succeed, but the resulting SRP session will not overwrite the previous SRP session state.

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 [Example 10.15, “The SRPVerifierStore interface”](#).

10.6.1. Providing Password Information for SRP

The default implementation of the **SRPVerifierStore** interface is not likely to be suitable for your production security environment as it requires all password hash information to be available as a file of serialized objects. You need to provide an MBean service that provides an implementation of the **SRPVerifierStore** interface that integrates with your existing security information stores. The **SRPVerifierStore** interface is shown in.

Example 10.15. The SRPVerifierStore interface

```
package org.jboss.security.srp;

import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
```

```

{
    public static class VerifierInfo implements Serializable
    {
        /**
         * The username the information applies to. Perhaps redundant
         * but it makes the object self contained.
         */
        public String username;

        /** The SRP password verifier hash */
        public byte[] verifier;
        /** The random password salt originally used to verify the
password */
        public byte[] salt;
        /** The SRP algorithm primitive generator */
        public byte[] g;
        /** The algorithm safe-prime modulus */
        public byte[] N;
    }

    /**
     * Get the indicated user's password verifier information.
     */
    public VerifierInfo getUserVerifier(String username)
        throws KeyException, IOException;

    /**
     * Set the indicated users' password verifier information. This
     * is equivalent to changing a user's password and should
     * generally invalidate any existing SRP sessions and caches.
     */
    public void setUserVerifier(String username, VerifierInfo info)
        throws IOException;

    /**
     * Verify an optional auxiliary challenge sent from the client to
     * the server. The auxChallenge object will have been decrypted
     * if it was sent encrypted from the client. An example of a
     * auxiliary challenge would be the validation of a hardware token
     * (SafeWord, SecureID, iButton) that the server validates to
     * further strengthen the SRP password exchange.
     */
    public void verifyUserChallenge(String username, Object
auxChallenge)
        throws SecurityException;
}

```

The primary function of a **SRPVerifierStore** implementation is to provide access to the **SRPVerifierStore.VerifierInfo** object for a given username. The **getUserVerifier(String)** method is called by the **SRPService** at that start of a user SRP session to obtain the parameters needed by the SRP algorithm. The elements of the **VerifierInfo** objects are:

- **username**: The user's name or id used to login.
- **verifier**: This is the one-way hash of the password or PIN the user enters as proof of their identity. The **org.jboss.security.Util** class has a **calculateVerifier** method that

performs that password hashing algorithm. The output password **H(salt | H(username | ':' | password))** as defined by RFC2945. Here **H** is the SHA secure hash function. The username is converted from a string to a **byte[]** using the UTF-8 encoding.

- **salt**: This is a random number used to increase the difficulty of a brute force dictionary attack on the verifier password database in the event that the database is compromised. It is a value that should be generated from a cryptographically strong random number algorithm when the user's existing clear-text password is hashed.
- **g**: The SRP algorithm primitive generator. In general this can be a well known fixed parameter rather than a per-user setting. The **org.jboss.security.srp.SRPConf** utility class provides several settings for **g** including a good default which can be obtained via **SRPConf.getDefaultParams().g()**.
- **N**: The SRP algorithm safe-prime modulus. In general this can be a well known fixed parameter rather than a per-user setting. The **org.jboss.security.srp.SRPConf** utility class provides several settings for **N** including a good default which can be obtained via **SRPConf.getDefaultParams().N()**.

So, step 1 of integrating your existing password store is the creation of a hashed version of the password information. If your passwords are already store in an irreversible hashed form, then this can only be done on a per-user basis as part of an upgrade procedure for example. Note that the **setUserVerifier(String, VerifierInfo)** method is not used by the current SRPService and may be implemented as no-op method, or even one that throws an exception stating that the store is read-only.

Step 2 is the creation of the custom **SRPVerifierStore** interface implementation that knows how to obtain the **VerifierInfo** from the store you created in step 1. The **verifyUserChallenge(String, Object)** method of the interface is only called if the client **SRPLoginModule** configuration specifies the **hasAuxChallenge** option. This can be used to integrate existing hardware token based schemes like SafeWord or Radius into the SRP algorithm.

Step 3 is the creation of an MBean that makes the step 2 implementation of the **SRPVerifierStore** interface available via JNDI, and exposes any configurable parameters you need. In addition to the default **org.jboss.security.srp.SRPVerifierStoreService** example, the SRP example presented later in this chapter provides a Java properties file based **SRPVerifierStore** implementation. Between the two examples you should have enough to integrate your security store.

10.6.2. 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. 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** obtains the private SRP session key that has been generated as a result of the previous messages exchanges. This is saved as a private credential in the login **Subject**. The server challenge response **M2** from step 4 is verified by invoking the **SRPClientSession.verify** method. If this succeeds, mutual authentication of the client to server, and server to client have been completed. The client side **SRPLoginModule** next creates a challenge **M1** to the server by invoking **SRPClientSession.response** method passing the server random number **B** as an argument. This challenge is sent to the server via the **SRPServerInterface.verify** method and server's response is saved as **M2**. This step corresponds to an exchange of challenges. At this point the server has verified that the user is who they say they are.
5. The client side **SRPLoginModule** saves the login username and **M1** challenge into the **LoginModule** sharedState map. This is used as the Principal name and credentials by the standard JBoss **ClientLoginModule**. The **M1** challenge is used in place of the password as proof of identity on any method invocations on J2EE components. The **M1** challenge is a cryptographically strong hash associated with the SRP session. Its interception via a third party cannot be used to obtain the user's password.
6. At the end of this authentication protocol, the **SRPServiceSession** has been placed into the **SRPService** authentication cache for subsequent use by the **SRPCacheLoginModule**.

Although SRP has many interesting properties, it is still an evolving component in the JBossSX framework and has some limitations of which you should be aware. Issues of note include the following:

- Because of how JBoss detaches the method transport protocol from the component container where authentication is performed, an unauthorized user could snoop the SRP **M1** challenge and effectively use the challenge to make requests as the associated username. Custom interceptors that encrypt the challenge using the SRP session key can be used to prevent this issue.
- The **SRPService** maintains a cache of SRP sessions that time out after a configurable period. Once they time out, any subsequent J2EE component access will fail because there is currently no mechanism for transparently renegotiating the SRP authentication credentials. You must either set the authentication cache timeout very long (up to 2,147,483,647 seconds, or approximately 68 years), or handle re-authentication in your code on failure.
- By default there can only be one SRP session for a given username. Because the negotiated SRP session produces a private session key that can be used for encryption/decryption between the client and server, the session is effectively a stateful one. JBoss supports for multiple SRP sessions per user, but you cannot encrypt data with one session key and then decrypt it with another.

To use end-to-end SRP authentication for J2EE component calls, you need to configure the security domain under which the components are secured to use the

`org.jboss.security.srp.jaas.SRPCacheLoginModule`. The `SRPCacheLoginModule` has a single configuration option named `cacheJndiName` that sets the JNDI location of the SRP authentication `CachePolicy` instance. This must correspond to the `AuthenticationCacheJndiName` attribute value of the `SRPService` MBean. The `SRPCacheLoginModule` authenticates user credentials by obtaining the client challenge from the `SRPServerSession` object in the authentication cache and comparing this to the challenge passed as the user credentials. Figure 10.16, “A sequence diagram illustrating the interaction of the `SRPCacheLoginModule` with the SRP session cache.” illustrates the operation of the `SRPCacheLoginModule.login` method implementation.

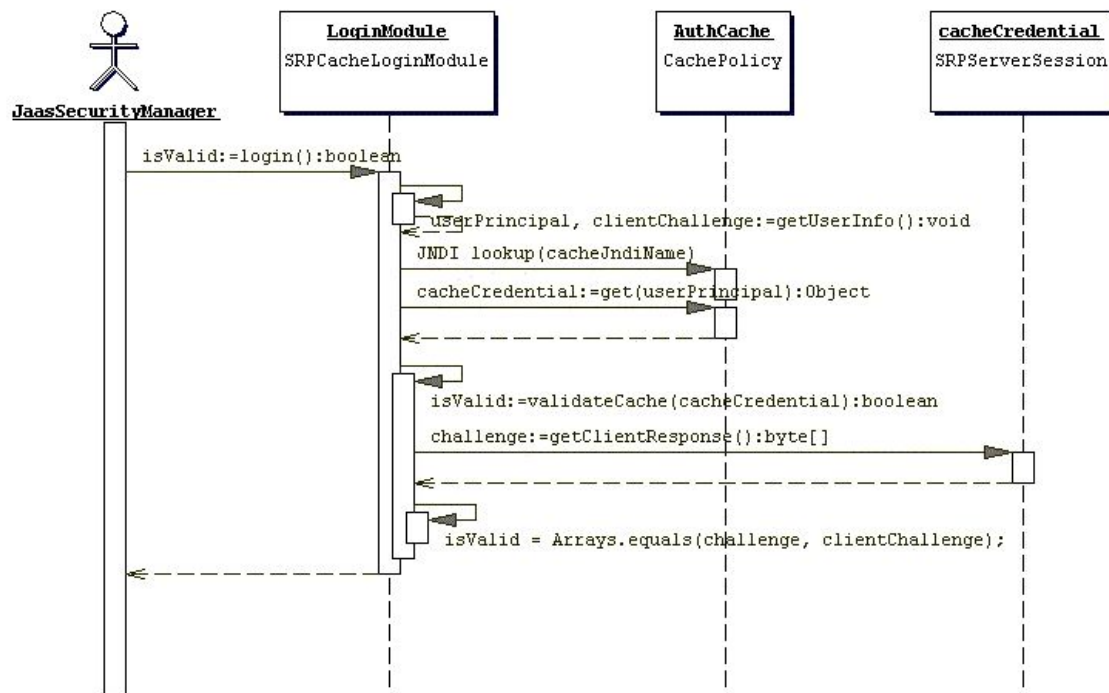


Figure 10.16. A sequence diagram illustrating the interaction of the `SRPCacheLoginModule` with the SRP session cache.

10.6.2.1. An SRP example

We have covered quite a bit of material on SRP and now its time to demonstrate SRP in practice with an example. The example demonstrates client side authentication of the user via SRP as well as subsequent secured access to a simple EJB using the SRP session challenge as the user credential. The test code deploys an EJB JAR that includes a SAR for the configuration of the server side login module configuration and SRP services. As in the previous examples we will dynamically install the server side login module configuration using the `SecurityConfig` MBean. In this example we also use a custom implementation of the `SRPVerifierStore` interface that uses an in memory store that is seeded from a Java properties file rather than a serialized object store as used by the `SRPVerifierStoreService`. This custom service is `org.jboss.book.security.ex3.service.PropertiesVerifierStore`. The following shows the contents of the JAR that contains the example EJB and SRP services.

```
[examples]$ jar tf output/security/security-ex3.jar
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
META-INF/jboss.xml
org.jboss.book.security.ex3/Echo.class
org.jboss.book.security.ex3/EchoBean.class
org.jboss.book.security.ex3/EchoHome.class
```

```
roles.properties
users.properties
security-ex3.sar
```

The key SRP related items in this example are the SRP MBean services configuration, and the SRP login module configurations. The **jboss-service.xml** descriptor of the **security-ex3.sar** is given in [Example 10.16, “The security-ex3.sar jboss-service.xml descriptor for the SRP services”](#), while [Example 10.17, “The client side standard JAAS configuration”](#) and [Example 10.18, “The server side XMLLoginConfig configuration”](#) give the example client side and server side login module configurations.

Example 10.16. The security-ex3.sar jboss-service.xml descriptor for the SRP services

```
<server>
  <!-- The custom JAAS login configuration that installs
        a Configuration capable of dynamically updating the
        config settings -->

    <mbean code="org.jboss.book.security.service.SecurityConfig"
          name="jboss.docs.security:service=LoginConfig-EX3">
      <attribute name="AuthConfig">META-INF/login-
config.xml</attribute>
      <attribute
name="SecurityConfigName">jboss.security:name=SecurityConfig</attribute>
    </mbean>

    <!-- The SRP service that provides the SRP RMI server and server
side
        authentication cache -->
    <mbean code="org.jboss.security.srp.SRPService"
          name="jboss.docs.security:service=SRPService">
      <attribute name="VerifierSourceJndiName">srp-test/security-
ex3</attribute>
      <attribute name="JndiName">srp-
test/SRPServiceInterface</attribute>
      <attribute name="AuthenticationCacheJndiName">srp-
test/AuthenticationCache</attribute>
      <attribute name="ServerPort">0</attribute>

<depends>jboss.docs.security:service=PropertiesVerifierStore</depends>
    </mbean>

    <!-- The SRP store handler service that provides the user password
verifier
        information -->
    <mbean code="org.jboss.security.ex3.service.PropertiesVerifierStore"
          name="jboss.docs.security:service=PropertiesVerifierStore">
      <attribute name="JndiName">srp-test/security-ex3</attribute>
    </mbean>
</server>
```

Example 10.17. The client side standard JAAS configuration

```
srp {
  org.jboss.security.srp.jaas.SRPLoginModule required
```



```

        srpServerJndiName="srp-test/SRPServiceInterface"
        ;

        org.jboss.security.ClientLoginModule required
        password-stacking="useFirstPass"
        ;
    };

```

Example 10.18. The server side XMLLoginConfig configuration

```

<application-policy name="security-ex3">
  <authentication>
    <login-module
      code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
      flag = "required">
      <module-option name="cacheJndiName">srp-
test/AuthenticationCache</module-option>
    </login-module>
    <login-module
      code="org.jboss.security.auth.spi.UsersRolesLoginModule"
      flag = "required">
      <module-option name="password-
stacking">useFirstPass</module-option>
    </login-module>
  </authentication>
</application-policy>

```

The example services are the **ServiceConfig** and the **PropertiesVerifierStore** and **SRPService** MBeans. Note that the **JndiName** attribute of the **PropertiesVerifierStore** is equal to the **VerifierSourceJndiName** attribute of the **SRPService**, and that the **SRPService** depends on the **PropertiesVerifierStore**. This is required because the **SRPService** needs an implementation of the **SRPVerifierStore** interface for accessing user password verification information.

The client side login module configuration makes use of the **SRPLoginModule** with a **srpServerJndiName** option value that corresponds to the JBoss server component **SRPService** JndiName attribute value (**srp-test/SRPServiceInterface**). Also needed is the **ClientLoginModule** configured with the **password-stacking="useFirstPass"** value to propagate the user authentication credentials generated by the **SRPLoginModule** to the EJB invocation layer.

There are two issues to note about the server side login module configuration. First, note the **cacheJndiName=srp-test/AuthenticationCache** configuration option tells the **SRPCacheLoginModule** the location of the **CachePolicy** that contains the **SRPServiceSession** for users who have authenticated against the **SRPService**. This value corresponds to the **SRPServiceAuthenticationCacheJndiName** attribute value. Second, the configuration includes a **UsersRolesLoginModule** with the **password-stacking=useFirstPass** configuration option. It is required to use a second login module with the **SRPCacheLoginModule** because SRP is only an authentication technology. A second login module needs to be configured that accepts the authentication credentials validated by the **SRPCacheLoginModule** to set the principal's roles that determines the

principal's permissions. The **UsersRolesLoginModule** is augmenting the SRP authentication with properties file based authorization. The user's roles are coming the **roles.properties** file included in the EJB JAR.

Now, run the example 3 client by executing the following command from the book examples directory:

```
[examples]$ ant -Dchap=security -Dex=3 run-example
...
run-example3:
    [echo] Waiting for 5 seconds for deploy...
    [java] Logging in using the 'srp' configuration
    [java] Created Echo
    [java] Echo.echo()#1 = This is call 1
    [java] Echo.echo()#2 = This is call 2
```

In the **examples/logs** directory you will find a file called **ex3-trace.log**. This is a detailed trace of the client side of the SRP algorithm. The traces show step-by-step the construction of the public keys, challenges, session key and verification.

Note that the client has taken a long time to run relative to the other simple examples. The reason for this is the construction of the client's public key. This involves the creation of a cryptographically strong random number, and this process takes quite a bit of time the first time it occurs. If you were to log out and log in again within the same VM, the process would be much faster. Also note that **Echo.echo()#2** fails with an authentication exception. The client code sleeps for 15 seconds after making the first call to demonstrate the behavior of the **SRPSERVICE** cache expiration. The **SRPSERVICE** cache policy timeout has been set to a mere 10 seconds to force this issue. As stated earlier, you need to make the cache timeout very long, or handle re-authentication on failure.

10.7. 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** and **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.

Both the **run.bat** and **run.sh** start scripts reference an **JAVA_OPTS** variable which you can use to set the security manager properties.

The next element of Java security is establishing the allowed permissions. You define the permissions by creating a **server.policy** file in the **\$JBOSS_HOME/bin/**directory. In this file you declare the following grant statement:

```
grant signedBy "jboss" {
    permission java.security.AllPermission;
};
```

This statement declares that all code signed by JBoss is trusted. To import the public key to your keystore, follow [Procedure 10.1, “Activate Java Security Manager”](#)

The current set of JBoss specific **java.lang.RuntimePermissions** are described below.

org.jboss.security.SecurityAssociation.getPrincipalInfo

Provides access to the **org.jboss.security.SecurityAssociation** **getPrincipal()** and **getCredentials()** methods. The risk involved with using this runtime permission is the ability to see the current thread caller and credentials.

org.jboss.security.SecurityAssociation.setPrincipalInfo

Provides access to the **org.jboss.security.SecurityAssociation** **setPrincipal()** and **setCredentials()** methods. The risk involved with using this runtime permission is the ability to set the current thread caller and credentials.

org.jboss.security.SecurityAssociation.setServer

Provides access to the **org.jboss.security.SecurityAssociation** **setServer** method. The risk involved with using this runtime permission is the ability to enable or disable multithread storage of the caller principal and credential.

org.jboss.security.SecurityAssociation.setRunAsRole

Provides access to the **org.jboss.security.SecurityAssociation** **pushRunAsRole** and **popRunAsRole** methods. The risk involved with using this runtime permission is the ability to change the current caller run-as role principal.

Procedure 10.1. Activate Java Security Manager

Follow this procedure to correctly configure the JSM for secure, production-ready operation. This procedure is only required while configuring your server for the first time. In this procedure, **\$JAVA_HOME** refers to the installation directory of the JRE.

1. Import public key to keystore

Execute the following command:

Linux

```
[home]$ sudo $JAVA_HOME/bin/keytool -import -alias jboss -file
JBossPublicKey.RSA -keystore $JAVA_HOME/jre/lib/security/cacerts
```

Windows

```
C:\> $JAVA_HOME\bin\keytool -import -alias jboss -file
JBossPublicKey.RSA -keystore $JAVA_HOME\jre\lib\security\cacerts
```

2. Verify key signature

Execute the following command in the terminal.



NOTE

The default JVM Keystore password is **changeit**.

```
$ keytool -list
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 2 entries

jboss, Aug 12, 2009, trustedCertEntry,
Certificate fingerprint (MD5):
93:F2:F1:8B:EF:8A:E0:E3:D0:E7:69:BC:69:96:29:C1
jbosscodesign2009, Aug 12, 2009, trustedCertEntry,
Certificate fingerprint (MD5):
93:F2:F1:8B:EF:8A:E0:E3:D0:E7:69:BC:69:96:29:C1
```

3. Specify additional JAVA_OPTS

Linux

Ensure the following block is present in the **\$JBOSS_HOME/bin/run.conf** file.

```
## Specify the Security Manager options
JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
Djava.security.policy==$DIRNAME/server.policy.cert
-Djava.protocol.handler.pkgs=org.jboss.handlers.stub
-Djava.security.debug=access:failure
-Djboss.home.dir=$DIRNAME/./
-Djboss.server.home.dir=$DIRNAME/./server/default/"
```

Windows

Ensure the following block is present in the **\$JBOSS_HOME\bin\run.conf.bat** file.

```
rem # Specify the Security Manager options
set "JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager
-Djava.security.policy==%DIRNAME%\server.policy.cert
-Djava.protocol.handler.pkgs=org.jboss.handlers.stub
-Djava.security.debug=access:failure
-Djboss.home.dir=%DIRNAME%/./
-Djboss.server.home.dir=%DIRNAME%/./server/default/"
```

4. Start the server

Start JBoss EAP using the **run.sh** or **run.bat** (Windows) script.

A number of Java debugging flags are available to assist you in determining how the security manager is using your security policy file, and what policy files are contributing permissions. Running the VM as follows shows the possible debugging flag settings:

```
[bin]$ java -Djava.security.debug=help

all            turn on all debugging
access         print all checkPermission results
combiner       SubjectDomainCombiner debugging
configfile     JAAS ConfigFile loading
configparser   JAAS ConfigFile parsing
gssloginconfig GSS LoginConfigImpl debugging
jar            jar verification
logincontext   login context results
policy         loading and granting
provider       security provider debugging
scl            permissions SecureClassLoader assigns
```

The following can be used with `access`:

```
stack          include stack trace
domain         dump all domains in context
failure        before throwing exception, dump stack
               and domain that didn't have permission
```

The following can be used with `stack` and `domain`:

```
permission=<classname>
               only dump output if specified permission
               is being checked
codebase=<URL>
               only dump output if specified codebase
               is being checked
```

Running with **`-Djava.security.debug=all`** provides the most output, but the output volume is acutely verbose. This might be a good place to start if you don't understand a given security failure at all. For less verbose output that will still assist with debugging permission failures, use **`-Djava.security.debug=access,failure`**.

10.8. USING SSL WITH JBOSS

10.8.1. Adding SSL to EJB3

By default JBoss EJB3 uses a socket based invoker layer on port 3878. This is set up in **`$JBOSS_HOME/server/<serverconfig>/deploy/ejb3.deployer/META-INF/jboss-service.xml`**. In some cases you may wish to use SSL as the protocol. In order to do this you must generate a keystore and then configure your beans to use SSL transport.

10.8.1.1. Generating the keystore and truststore

For SSL to work you need to create a public/private key pair, which will be stored in a keystore. Generate this using the **`genkey`** command that comes with the JDK.

```
$cd $JBOSS_HOME/server/production/conf/
```

```
$keytool -genkey -alias ejb3-ssl -keypass opensource -keystore
localhost.keystore
Enter keystore password:  opensource
What is your first and last name?
[Unknown]:
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]:
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:
Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
correct?
[no]:  yes
```

where **alias** is the name ("ejb2-ssl") of the key pair within the keystore. **keypass** is the password ("opensource") for the keystore, and **keystore** specifies the location ("localhost.keystore") of the keystore to create/add to.

Since you have not signed our certificate through any certification authority, you also need to create a truststore for the client, explicitly saying that you trust the certificate you just created. The first step is to export the certificate using the JDK keytool:

```
$ keytool -export -alias ejb3-ssl -file mycert.cer -keystore
localhost.keystore
Enter keystore password:  opensource
Certificate stored in file <mycert.cer>
```

Then you need to create the truststore if it does not exist and import the certificate into the truststore:

```
$ keytool -import -alias ejb3-ssl -file mycert.cer -keystore
localhost.truststore
Enter keystore password:  opensource
Owner: CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown,
C=Unknown
Issuer: CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown,
C=Unknown
Serial number: 43bff927
Valid from: Sat Jan 07 18:23:51 CET 2006 until: Fri Apr 07 19:23:51
CEST 2006
Certificate fingerprints:
    MD5:  CF:DC:71:A8:F4:EA:8F:5A:E9:94:E3:E6:5B:A9:C8:F3
    SHA1:
0E:AD:F3:D6:41:5E:F6:84:9A:D1:54:3D:DE:A9:B2:01:28:F6:7C:26
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

10.8.1.2. Setting up the SSL transport

The simplest way to define an SSL transport is to define a new Remoting connector using the **sslsocket** protocol as follows. This transport will listen on port 3843:

```

<mbean code="org.jboss.remoting.transport.Connector"
      xmbean-dd="org/jboss/remoting/transport/Connector.xml"

name="jboss.remoting:type=Connector,transport=socket3843,handler=ejb3">
  <depends>jboss.aop:service=AspectDeployer</depends>
  <attribute
name="InvokerLocator">sslsocket://0.0.0.0:3843</attribute>
  <attribute name="Configuration">
    <handlers>
      <handler subsystem="AOP">
        org.jboss.aspects.remoting.AOPRemotingInvocationHandler
      </handler>
    </handlers>
  </attribute>
</mbean>

```

Now you need to tell JBoss Remoting where to find the keystore to be used for SSL and its password. This is done using **javax.net.ssl.keyStore** and **javax.net.ssl.keyStorePassword=opensource** system properties when starting JBoss, as the following example shows:

```

$cd $JBOSS_HOME/bin
$ run -
Djavax.net.ssl.keyStore=../server/production/conf/localhost.keystore
-Djavax.net.ssl.keyStorePassword=opensource

```

10.8.1.3. Configuring your beans to use the SSL transport

By default all the beans will use the default connector on **socket://0.0.0.0:3873**. By using the **@org.jboss.annotation.ejb.RemoteBinding** annotation you can have the bean invokable via SSL.

```

@RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL"),
@Remote(BusinessInterface.class)
public class StatefulBean implements BusinessInterface
{
    ...
}

```

This bean will be bound under the JNDI name **StatefulSSL** and the proxy implementing the remote interface returned to the client will communicate with the server via SSL.

You can also enable different types of communication for your beans

```

@RemoteBindings({
    @RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL"),
    @RemoteBinding(jndiBinding="StatefulNormal")
})
@Remote(BusinessInterface.class)
public class StatefulBean implements BusinessInterface

```

```
{
    ...
}
```

Now if you look up **StatefulNormal** the returned proxy implementing the remote interface will communicate with the server via the normal unencrypted socket protocol, and if you look up **StatefulSSL** the returned proxy implementing the remote interface will communicate with the server via SSL.

10.8.1.4. Setting up the client to use the truststore

If not using a certificate signed by a certificate authorization authority, you need to point the client to the truststore using the **javax.net.ssl.trustStore** system property and specify the password using the **javax.net.ssl.trustStorePassword** system property:

```
java -Djavax.net.ssl.trustStore=${resources}/test/ssl/localhost.truststore
-Djavax.net.ssl.trustStorePassword=opensource com.acme.RunClient
```

10.8.2. Adding SSL to EJB 2.1 calls

10.8.2.1. Generating the keystore and truststore

This is similar to the steps described for Adding SSL to EJB3 calls.

10.8.2.2. Setting up the SSL transport

Now you need to tell JBoss Remoting where to find the keystore to be used for SSL and its password. This is done using **javax.net.ssl.keyStore** and **javax.net.ssl.keyStorePassword=opensource** system properties when starting JBoss, as the following example shows:

```
$cd $JBOSS_HOME/bin
$ run -
Djavax.net.ssl.keyStore=../server/production/conf/localhost.keystore
-Djavax.net.ssl.keyStorePassword=opensource
```

If you wish to customize the SSLSocketBuilder you need to add the following to your **\$JBOSS_HOME/server/\${serverConf}/conf/jboss-service.xml** file.

```
<!-- This section is for custom (SSL) server socket factory -->
<mbean code="org.jboss.remoting.security.SSLSocketBuilder"
  name="jboss.remoting:service=SocketBuilder,type=SSL"
  display-name="SSL Server Socket Factory Builder">
  <!-- IMPORTANT - If making ANY customizations, this MUST be set to
false. -->
  <!-- Otherwise, will used default settings and the following
attributes will be ignored. -->
  <attribute name="UseSSLServerSocketFactory">false</attribute>
  <!-- This is the url string to the key store to use -->
  <attribute name="KeyStoreURL">localhost.keystore</attribute>
  <!-- The password for the key store -->
  <attribute name="KeyStorePassword">sslsocket</attribute>
  <!-- The password for the keys (will use KeyStorePassword if this is
```



```

not set explicitly. -->
    <attribute name="KeyPassword">sslsocket</attribute>
    <!-- The protocol for the SSLContext. Default is TLS. -->
    <attribute name="SecureSocketProtocol">TLS</attribute>
    <!-- The algorithm for the key manager factory. Default is SunX509.
-->
    <attribute name="KeyManagementAlgorithm">SunX509</attribute>
    <!-- The type to be used for the key store. -->
    <!-- Defaults to JKS. Some acceptable values are JKS (Java Keystore
- Sun's keystore format), -->
    <!-- JCEKS (Java Cryptography Extension keystore - More secure
version of JKS), and -->
    <!-- PKCS12 (Public-Key Cryptography Standards #12
        keystore - RSA's Personal Information Exchange Syntax
Standard). -->
    <!-- These are not case sensitive. -->
    <attribute name="KeyStoreType">JKS</attribute>
</mbean>

<mbean code="org.jboss.remoting.security.SSLServerSocketFactoryService"
    name="jboss.remoting:service=ServerSocketFactory,type=SSL"
    display-name="SSL Server Socket Factory">
    <depends optional-attribute-name="SSLSocketBuilder"
        proxy-
type="attribute">jboss.remoting:service=SocketBuilder,type=SSL</depends>
</mbean>

```

10.8.2.3. Configuring your beans to use the SSL transport

In your `$JBOSS_HOME/server/${serverConf}/conf/jboss-service.xml` file, comment out the following lines:

```

<mbean code="org.jboss.remoting.transport.Connector"
    name="jboss.remoting:service=Connector,transport=socket"
    display-name="Socket transport Connector">

    <!-- Can either just specify the InvokerLocator attribute and not
the invoker element in the -->
    <!-- Configuration attribute, or do the full invoker configuration
in the in invoker element -->
    <!-- of the Configuration attribute. -->

    <!-- Remember that if you do use more than one param on the uri,
will have to include as a CDATA, -->
    <!-- otherwise, parser will complain. -->
    <!--
        <attribute name="InvokerLocator">
            <![CDATA[socket://${jboss.bind.address}:4446/?
datatype=invocation]]>
        </attribute>
    -->

    <attribute name="Configuration">
        <!-- Using the following
            <invoker>

```

```

        element instead of the InvokerLocator above because specific
        attributes needed.
        -->
        <!-- If wanted to use any of the parameters below, can
        just add them as parameters to the url above if wanted use
        the InvokerLocator attribute. -->
        <config>
        <!-- Other than transport type and handler, none of these
        configurations are required
        (will just use defaults). -->
        <invoker transport="socket">
        <attribute name="dataType"
isParam="true">invocation</attribute>
        <attribute name="marshaller"

isParam="true">org.jboss.invocation.unified.marshall.InvocationMarshaller<
/attribute>
        <attribute name="unmarshaller"

isParam="true">org.jboss.invocation.unified.marshall.InvocationUnMarshalle
r</attribute>
        <!-- This will be port on which the marshall loader port
        runs on. -->
        <!-- <attribute name="loaderport"
isParam="true">4447</attribute> -->
        <!-- The following are specific to socket invoker -->
        <!-- <attribute name="numAcceptThreads">1</attribute>-->
        <!-- <attribute name="maxPoolSize">303</attribute>-->
        <!-- <attribute name="clientMaxPoolSize"
isParam="true">304</attribute>-->
        <attribute name="socketTimeout"
isParam="true">600000</attribute>
        <attribute name="serverBindAddress">${jboss.bind.address}
</attribute>
        <attribute name="serverBindPort">4446</attribute>
        <!-- <attribute
name="clientConnectAddress">216.23.33.2</attribute> -->
        <!-- <attribute name="clientConnectPort">7777</attribute>
-->
        <attribute name="enableTcpNoDelay"
isParam="true">true</attribute>
        <!-- <attribute name="backlog">200</attribute>-->
        <!-- The following is for callback configuration and is
independant of invoker type -->
        <!-- <attribute name="callbackMemCeiling">30</attribute>--
>
        <!-- indicates callback store by fully qualified class
name -->
        <!-- <attribute
name="callbackStore">org.jboss.remoting.CallbackStore</attribute>-->
        <!-- indicates callback store by object name -->
        <!--
        <attribute name="callbackStore">

jboss.remoting:service=CallbackStore,type=Serializable
        </attribute>

```

```

-->
<!-- config params for callback store.  if were declaring
callback store via object name, -->
<!-- could have specified these config params there. -->
<!-- StoreFilePath indicates to which directory to write
the callback objects. -->
<!-- The default value is the property value of
'jboss.server.data.dir' and
        if this is not set, -->
<!-- then will be 'data'. Will then append 'remoting' and
the callback client's session id. -->
<!-- An example would be 'data\remoting\5c40051-9jijyx-
e5b6xyph-1-e5b6xyph-2'. -->
<!-- <attribute name="StoreFilePath">callback</attribute>-
->

<!-- StoreFileSuffix indicates the file suffix to use
for the callback objects written to disk. -->
<!-- The default value for file suffix is 'ser'. -->
<!-- <attribute name="StoreFileSuffix">cst</attribute>-->
</invoker>

<!-- At least one handler is required by the connector.  If
have more than one, must decalre -->
<!-- different subsystem values.  Otherwise, all invocations
will be routed to the only one -->
<!-- that is declared. -->
<handlers>
    <!-- can also specify handler by fully qualified classname
-->
    <handler
subsystem="invoker">jboss:service=invoker,type=unified</handler>
</handlers>
</config>
</attribute>
<depends>jboss.remoting:service=NetworkRegistry</depends>
</mbean>

```

and add the following in it's place:

```

<mbean code="org.jboss.remoting.transport.Connector"
xmbean-dd="org/jboss/remoting/transport/Connector.xml"
name="jboss.remoting:service=Connector,transport=sslsocket">
display-name="SSL Socket transport Connector">

    <attribute name="Configuration">
        <config>
            <invoker transport="sslsocket">
                <attribute name="dataType" isParam="true">invocation</attribute>
                <attribute name="marshaller"
isParam="true">org.jboss.invocation.unified.marshall.InvocationMarshaller<
/attribute>
                <attribute name="unmarshaller"
isParam="true">org.jboss.invocation.unified.marshall.InvocationUnMarshalle
r</attribute>
                <attribute name="serverSocketFactory">
jboss.remoting:service=ServerSocketFactory,type=SSL

```

```

        </attribute>
        <attribute name="serverBindAddress">${jboss.bind.address}
</attribute>
        <attribute name="serverBindPort">3843</attribute>
    </invoker>
    <handlers>
        <handler
subsystem="invoker">jboss:service=invoker,type=unified</handler>
    </handlers>
</config>
</attribute>
    <!--If you specify the keystore and password in the command line and
you're
        not using the custom ServerSocketFactory, you should take out the
following line-->

<depends>jboss.remoting:service=ServerSocketFactory,type=SSL</depends>
    <depends>jboss.remoting:service=NetworkRegistry</depends>
</mbean>

```

10.8.2.4. Setting up the client to use the truststore

This is similar to the steps described for EJB3.

10.9. CONFIGURING JBOSS FOR USE BEHIND A FIREWALL

JBoss comes with many socket based services that open listening ports. In this section we list the services that open ports that might need to be configured to work when accessing JBoss behind a firewall. The following table shows the ports, socket type, associated service for the services in the default configuration file set. [Table 10.2, "Additional ports in the all configuration"](#) shows the same information for the additional ports that exist in the all configuration file set.

Table 10.1. The ports found in the default configuration

Port	Type	Service	Notes
1098	TCP	org.jboss.naming.NamingService	
1099	TCP	org.jboss.naming.NamingService	
3873	TCP	org.jboss.remoting.transport.Connector (EJB3)	
4444	TCP	org.jboss.invocation.jrmp.server.JRMPInvoker	
4445	TCP	org.jboss.invocation.pooled.server.PooledInvoker	

Port	Type	Service	Notes
4446	TCP	org.jboss.remoting.transport.Connector (UnifiedInvoker)	
4457	TCP	org.jboss.remoting.transport.Connector (Messaging)	Plus one additional anonymous TCP port. You can specify fixed port number via the <code>secondaryBindPort</code> parameter in the deploy/jboss-messaging.sar/remoting-bisocket-service.xml file.
8009	TCP	org.jboss.web.tomcat.service.JBossWeb	
8080	TCP	org.jboss.web.tomcat.service.JBossWeb	
8083	TCP	org.jboss.web.WebService	

Table 10.2. Additional ports in the all configuration

Port	Type	Service	Notes
1100	TCP	org.jboss.ha.jndi.HANamingService	
1101	TCP	org.jboss.ha.jndi.HANamingService	
1102	UDP	org.jboss.ha.jndi.HANamingService	

Port	Type	Service	Notes
1161	UDP	org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService	Plus one additional anonymous UDP port which does not support configuration the port.
1162	UDP	org.jboss.jmx.adaptor.snmp.trapd.TrapdService	
3528	TCP	org.jboss.invocation.iiop.IIOPInvoker	
4447	TCP	org.jboss.invocation.jrmp.server.JRMPInvokerHA	
7900	TCP	org.jboss.messaging.core.jmx.MessagingPostOfficeService (Messaging, DataChannel)	Plus one additional anonymous TCP port. It can be set using the <code>FD_SOCKET.start_port</code> parameter.
43333	UDP	org.jboss.cache.TreeCache (EJB3Entity)	Plus one additional anonymous UDP port for unicast and one additional anonymous TCP port. The UDP port can be set using the <code>rcv_port</code> parameter and the TCP port can be set using the <code>FD_SOCKET.start_port</code> parameter.
45551	UDP	org.jboss.cache.TreeCache (EJB3SFSB)	Plus one additional anonymous UDP port for unicast and one additional anonymous TCP port. The UDP port can be set using the <code>rcv_port</code> parameter and the TCP port can be set using the <code>FD_SOCKET.start_port</code> parameter.

Port	Type	Service	Notes
45566	UDP	org.jboss.ha.framework.server.ClusterPartition	Plus one additional anonymous UDP port for unicast and one additional anonymous TCP port. The UDP port can be set using the <code>rcv_port</code> parameter and the TCP port can be set using the <code>FD_SOCKET.start_port</code> parameter.
45567	UDP	org.jboss.messaging.core.jmx.MessagingPostOfficeService (Messaging, DataChannel MPING)	
45568	UDP	org.jboss.messaging.core.jmx.MessagingPostOfficeService (Messaging, ControlChannel)	Plus one additional anonymous UDP port for unicast and one additional anonymous TCP port. The UDP port can be set using the <code>rcv_port</code> parameter and the TCP port can be set using the <code>FD_SOCKET.start_port</code> parameter.
45577	UDP	org.jboss.cache.TreeCache (JBossWebCluster)	Plus one additional anonymous UDP port for unicast and one additional anonymous TCP port. The UDP port can be set using the <code>rcv_port</code> parameter and the TCP port can be set using the <code>FD_SOCKET.start_port</code> parameter.

10.10. HOW TO SECURE THE JBOSS SERVER

JBoss comes with several admin access points that need to be secured or removed to prevent unauthorized access to administrative functions in a deployment. This section describes the various admin services and how to secure them.

10.10.1. The JMX Console

The **jmx-console.war** found in the deploy directory provides an html view into the JMX microkernel. As such, it provides access to arbitrary admin type access like shutting down the server, stopping services, deploying new services, etc. It should either be secured like any other web application, or removed.

10.10.2. The Web Console

The **web-console.war** found in the **deploy/management** directory is another web application view into the JMX microkernel. This uses a combination of an applet and a HTML view and provides the same level of access to admin functionality as the **jmx-console.war**. As such, it should either be secured or removed. The **web-console.war** contains commented out templates for basic security in its **WEB-INF/web.xml** as well as commented out setup for a security domain in **WEB-INF/jboss-web.xml**.

10.10.3. The HTTP Invokers

The **http-invoker.sar** found in the deploy directory is a service that provides RMI/HTTP access for EJBs and the JNDI **Naming** service. This includes a servlet that processes posts of marshalled **org.jboss.invocation.Invocation** objects that represent invocations that should be dispatched onto the **MBeanServer**. Effectively this allows access to MBeans that support the detached invoker operation via HTTP since one could figure out how to format an appropriate HTTP post. To security this access point you would need to secure the **JMXInvokerServlet** servlet found in the **http-invoker.sar/invoker.war/WEB-INF/web.xml** descriptor. There is a secure mapping defined for the **/restricted/JMXInvokerServlet** path by default, one would simply have to remove the other paths and configure the **http-invoker** security domain setup in the **http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml** descriptor.

10.10.4. The JMX Invoker

The **jmx-invoker-adaptor-server.sar** is a service that exposes the JMX MBeanServer interface via an RMI compatible interface using the RMI/JRMP detached invoker service. The only way for this service to be secured currently would be to switch the protocol to RMI/HTTP and secure the **http-invoker.sar** as described in the previous section. In the future this service will be deployed as an XMBean with a security interceptor that supports role based access checks.

CHAPTER 11. WEB SERVICES

The biggest new feature of J2EE 1.4 is the ability of J2EE components to act both as web service providers and consumers.

11.1. DOCUMENT/LITERAL

With document style web services two business partners agree on the exchange of complex business documents that are well defined in XML schema. For example, one party sends a document describing a purchase order, the other responds (immediately or later) with a document that describes the status of the purchase order. No need to agree on such low level details as operation names and their associated parameters. The payload of the SOAP message is an XML document that can be validated against XML schema. Document is defined by the style attribute on the SOAP binding.

```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
  <soap:binding style='document'
transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='concat'>
    <soap:operation soapAction=''/>
    <input>
      <soap:body use='literal'/>
    </input>
    <output>
      <soap:body use='literal'/>
    </output>
  </operation>
</binding>
```

With document style web services the payload of every message is defined by a complex type in XML schema.

```
<complexType name='concatType'>
  <sequence>
    <element name='String_1' nillable='true' type='string'/>
    <element name='long_1' type='long'/>
  </sequence>
</complexType>
<element name='concat' type='tns:concatType'/>
Therefore, message parts must refer to an element from the schema.
<message name='EndpointInterface_concat'>
  <part name='parameters' element='tns:concat'/>
</message>
The following message definition is invalid.
<message name='EndpointInterface_concat'>
  <part name='parameters' type='tns:concatType'/>
</message>
```

11.2. DOCUMENT/LITERAL (BARE)

Bare is an implementation detail from the Java domain. Neither in the abstract contract (i.e. wsdl+schema) nor at the SOAP message level is a bare endpoint recognizable. A bare endpoint or client uses a Java bean that represents the entire document payload.

```
@WebService
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public class DocBareServiceImpl
{
    @WebMethod
    public SubmitBareResponse submitPO(SubmitBareRequest poRequest)
    {
        ...
    }
}
```

The trick is that the Java beans representing the payload contain JAXB annotations that define how the payload is represented on the wire.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SubmitBareRequest",
namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/", propOrder
= { "product" })
@XmlRootElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss
.org/", name = "SubmitPO")
public class SubmitBareRequest
{

    @XmlElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/
", required = true)
    private String product;
    ...
}
```

11.3. DOCUMENT/LITERAL (WRAPPED)

Wrapped is an implementation detail from the Java domain. Neither in the abstract contract (i.e. wsdl+schema) nor at the SOAP message level is a wrapped endpoint recognizable. A wrapped endpoint or client uses the individual document payload properties. Wrapped is the default and does not have to be declared explicitly.

```
@WebService
public class DocWrappedServiceImpl
{
    @WebMethod
    @RequestWrapper (className="org.somepackage.SubmitPO")
    @ResponseWrapper (className="org.somepackage.SubmitPOResponse")
    public String submitPO(String product, int quantity)
    {
        ...
    }
}
```

Note, that with JBossWS the request/response wrapper annotations are not required, they will be generated on demand using sensible defaults.

11.4. RPC/LITERAL

With RPC there is a wrapper element that names the endpoint operation. Child elements of the RPC parent are the individual parameters. The SOAP body is constructed based on some simple rules:

- The port type operation name defines the endpoint method name
- Message parts are endpoint method parameters

RPC is defined by the style attribute on the SOAP binding.

```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
  <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http' />
  <operation name='echo'>
    <soap:operation soapAction='' />
    <input>
      <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
use='literal' />
    </input>
    <output>
      <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
use='literal' />
    </output>
  </operation>
</binding>
```

With rpc style web services the portType names the operation (i.e. the java method on the endpoint)

```
<portType name='EndpointInterface'>
  <operation name='echo' parameterOrder='String_1'>
    <input message='tns:EndpointInterface_echo' />
    <output message='tns:EndpointInterface_echoResponse' />
  </operation>
</portType>
```

Operation parameters are defined by individual message parts.

```
<message name='EndpointInterface_echo'>
  <part name='String_1' type='xsd:string' />
</message>
<message name='EndpointInterface_echoResponse'>
  <part name='result' type='xsd:string' />
</message>
```

Note, there is no complex type in XML schema that could validate the entire SOAP message payload.

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
```

```

{
  @WebMethod
  @WebResult(name="result")
  public String echo(@WebParam(name="String_1") String input)
  {
    ...
  }
}

```

The element names of RPC parameters/return values may be defined using the JAX-WS Annotations#`javax.jws.WebParam` and JAX-WS Annotations#`javax.jws.WebResult` respectively.

11.5. RPC/ENCODED

SOAP encoding style is defined by the infamous [chapter 5](#) of the [SOAP-1.1](#) specification. It has inherent interoperability issues that cannot be fixed. The [Basic Profile-1.0](#) prohibits this encoding style in [4.1.7 SOAP encodingStyle Attribute](#). JBossWS has basic support for `rpc/encoded` that is provided as is for simple interop scenarios with SOAP stacks that do not support literal encoding. Specifically, JBossWS does not support:-

- element references
- soap arrays as bean properties

11.6. WEB SERVICE ENDPOINTS

JAX-WS simplifies the development model for a web service endpoint a great deal. In short, an endpoint implementation bean is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes the abstract contract (i.e. `wsdl+schema`) for client consumption. All marshalling/unmarshalling is delegated to JAXB [2].

11.7. PLAIN OLD JAVA OBJECT (POJO)

Let's take a look at simple POJO endpoint implementation. All endpoint associated metadata is provided via JSR-181 annotations

```

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
  @WebMethod
  public String echo(String input)
  {
    ...
  }
}

```

11.8. THE ENDPOINT AS A WEB APPLICATION

A JAX-WS java service endpoint (JSE) is deployed as a web application.

```

<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>

```

```

    <servlet-
class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>TestService</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

11.9. PACKAGING THE ENDPOINT

A JSR-181 java service endpoint (JSE) is packaged as a web application in a *.war file.

```

<war warfile="${build.dir}/libs/jbossws-samples-jsr181pojo.war"
      webxml="${build.resources.dir}/samples/jsr181pojo/WEB-
INF/web.xml">
    <classes dir="${build.dir}/classes">
<include name="org/jboss/test/ws/samples/jsr181pojo/JSEBean01.class"/>
    </classes>
</war>

```

Note, only the endpoint implementation bean and web.xml are required.

11.10. ACCESSING THE GENERATED WSDL

A successfully deployed service endpoint will show up in the service endpoint manager. This is also where you find the links to the generated wsdl.

```
http://yourhost:8080/jbossws/services
```

Note, it is also possible to generate the abstract contract off line using jbossw tools. For details of that please see [#Top Down \(Java to WSDL\)](#)

11.11. EJB3 STATELESS SESSION BEAN (SLSB)

The JAX-WS programming model support the same set of annotations on EJB3 stateless session beans as on [# Plain old Java Object \(POJO\)](#) endpoints. EJB-2.1 endpoints are supported using the JAX-RPC programming model.

```

@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
{
    @WebMethod
    public String echo(String input)

```

```
{  
  ...  
}
```

Above you see an EJB-3.0 stateless session bean that exposes one method both on the remote interface and on and as an endpoint operation.

Packaging the endpoint

A JSR-181 EJB service endpoint is packaged as an ordinary ejb deployment.

```
<jar jarfile="${build.dir}/libs/jbossws-samples-jsr181ejb.jar">  
<fileset dir="${build.dir}/classes">  
<include name="org/jboss/test/ws/samples/jsr181ejb/EJB3Bean01.class"/>  
<include  
  name="org/jboss/test/ws/samples/jsr181ejb/EJB3RemoteInterface.class"/>  
</fileset>  
</jar>
```

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the service endpoint manager. This is also where you find the links to the generated wsdl.

```
http://yourhost:8080/jbossws/services
```

Note, it is also possible to generate the abstract contract off line using jboss tools. For details of that please see [#Top Down \(Java to WSDL\)](#)

11.12. ENDPOINT PROVIDER

JAX-WS services typically implement a native Java service endpoint interface (SEI), perhaps mapped from a WSDL port type, either directly or via the use of annotations.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The Provider interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML message level.

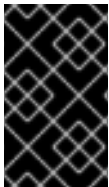
A Provider based service instance's invoke method is called for each message received for the service.

```
@WebServiceProvider  
@ServiceMode(value = Service.Mode.PAYLOAD)  
public class ProviderBeanPayload implements Provider<Source>  
{  
  public Source invoke(Source req)  
  {  
    // Access the entire request PAYLOAD and return the response PAYLOAD  
  }  
}
```

Note, `Service.Mode.PAYLOAD` is the default and does not have to be declared explicitly. You can also use `Service.Mode.MESSAGE` to access the entire SOAP message (i.e. with `MESSAGE` the Provider can also see SOAP Headers)

11.13. POJO ENDPOINT AUTHENTICATION AND AUTHORIZATION

In general the credentials obtained by WS-Security are only used for EJB endpoints or for POJO endpoints when they make a call to another secured resource, it is now possible to enable authentication and authorization checking for POJO endpoints. (This should not be enabled for EJB based endpoints as the EJB container will still take care of the security requirements of the deployed bean).



IMPORTANT

These instructions assume WS-Security has already been enabled, these instructions describe the additional configuration required to enable authentication and authorization for POJO endpoints.

The `.war` containing the POJO endpoint needs to have a security domain defined, this is achieved by defining a security-domain in the `jboss-web` deployment descriptor within the **WEB-INF** folder.

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

The remainder of the configuration to enable authentication and authorization is undertaken within the `jboss-wsse-server.xml` deployment descriptor.

To enable the POJO authentication and authorization a new authorization element needs to be added to the appropriate config element within the descriptor.

```
<jboss-ws-security>
  <config>
    <authorize>
      <!-- Must contain either <unchecked/> or one or more
<role>RoleName</role> definitions. -->
    </authorize>
  </config>
</jboss-ws-security>
```

The config element can be defined globally and be port specific or even operation specific.

The authorize element must contain either the unchecked element or one or more `RoleName` role elements.

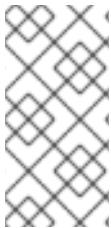
```
<jboss-ws-security>
  <config>
    <authorize>
      <unchecked/>
    </authorize>
  </config>
</jboss-ws-security>
```

The above example has the unchecked element defined, this means that the authentication step will be

performed to validate the users username and credential but no further role checking will take place. If the users username and credential are invalid then the request will be rejected otherwise the request will be allowed to proceed.

```
<jboss-ws-security>
  <config>
    <authorize>
      <role>friend</role>
      <role>family</role>
    </authorize>
  </config>
</jboss-ws-security>
```

This second example has two roles defined, as before the user will be authenticated using their username and credential and they will then be checked to ensure that they have been assigned at least one of the roles of friend or family.



NOTE

Authentication and authorization will still proceed even if no username and password or certificate was provided in the request message. In this scenario authentication may proceed if the login module of the security domain has been configured with an anonymous identity.

11.14. WEBSERVICECONTEXT

The `WebServiceContext` is treated as an injectable resource that can be set at the time an endpoint is initialized. The `WebServiceContext` object will then use thread-local information to return the correct information regardless of how many threads are concurrently being used to serve requests addressed to the same endpoint object.

```
@WebService
public class EndpointJSE
{
  @Resource
  WebServiceContext wsCtx;

  @WebMethod
  public String testGetMessageContext()
  {
    SOAPMessageContext jaxwsContext =
      (SOAPMessageContext)wsCtx.getMessageContext();
    return jaxwsContext != null ? "pass" : "fail";
  }
  ..
  @WebMethod
  public String testGetUserPrincipal()
  {
    Principal principal = wsCtx.getUserPrincipal();
    return principal.getName();
  }

  @WebMethod
```



```

public boolean testIsUserInRole(String role)
{
    return wsCtx.isUserInRole(role);
}
}

```

11.15. WEB SERVICE CLIENTS

11.15.1. Service

Service is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at a particular endpoint address.

For most clients, you will start with a set of stubs generated from the WSDL. One of these will be the service, and you will create objects of that class in order to work with the service (see "static case" below).

11.15.1.1. Service Usage

Static case

Most clients will start with a WSDL file, and generate some stubs using jbossws tools like *wsconsume*. This usually gives a mass of files, one of which is the top of the tree. This is the service implementation class.

The generated implementation class can be recognised as it will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a `java.net.URL`) and the service name (a `javax.xml.namespace.QName`) respectively.

Usually you will use the no-argument constructor. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the `WebServiceClient` annotation that decorates the generated class.

The following code snippet shows the generated constructors from the generated class:

```

// Generated Service Class

@WebServiceClient(name="StockQuoteService",
targetNamespace="http://example.com/stocks",
wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
    public StockQuoteService()
    {
        super(new URL("http://example.com/stocks.wsdl"), new
QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(String wsdlLocation, QName serviceName)
    {
        super(wsdlLocation, serviceName);
    }
}

```

```
...  
}
```

Section [#Dynamic Proxy](#) explains how to obtain a port from the service and how to invoke an operation on the port. If you need to work with the XML payload directly or with the XML representation of the entire SOAP message, have a look at [#Dispatch](#).

Dynamic case

In the dynamic case, when nothing is generated, a web service client uses **Service.create** to create Service instances, the following code illustrates this process.

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");  
QName serviceName = new QName("http://example.org/sample", "MyService");  
Service service = Service.create(wsdlLocation, serviceName);
```

This is the nastiest way to work with JBossWs. Older versions have extensive details on DII as it was then known.

11.15.1.2. Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-WS runtime system. [#Handler Framework](#) describes the handler framework in detail. A Service instance provides access to a HandlerResolver via a pair of getHandlerResolver/setHandlerResolver methods that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a Service instance is used to create a proxy or a Dispatch instance then the handler resolver currently registered with the service is used to create the required handler chain. Subsequent changes to the handler resolver configured for a Service instance do not affect the handlers on previously created proxies, or Dispatch instances.

11.15.1.3. Executor

Service instances can be configured with a `java.util.concurrent.Executor`. The executor will then be used to invoke any asynchronous callbacks requested by the application. The `setExecutor` and `getExecutor` methods of Service can be used to modify and retrieve the executor configured for a service.

11.15.2. Dynamic Proxy

You can create an instance of a client proxy using one of `getPort` methods on the [#Service](#).

```
/**  
 * The getPort method returns a proxy. A service client  
 * uses this proxy to invoke operations on the target  
 * service endpoint. The serviceEndpointInterface  
 * specifies the service endpoint interface that is supported by  
 * the created dynamic proxy instance.  
 */  
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)  
{  
    ...  
}
```

```

    }

    /**
     * The getPort method returns a proxy. The parameter
     * <code>serviceEndpointInterface</code> specifies the service
     * endpoint interface that is supported by the returned proxy.
     * In the implementation of this method, the JAX-WS
     * runtime system takes the responsibility of selecting a protocol
     * binding (and a port) and configuring the proxy accordingly.
     * The returned proxy should not be reconfigured by the client.
     */
    public <T> T getPort(Class<T> serviceEndpointInterface)
    {
        ...
    }

```

The service endpoint interface (SEI) is usually generated using tools. For details see [# Top Down \(WSDL to Java\)](#)

A generated static [#Service](#) usually also offers typed methods to get ports. These methods also return dynamic proxies that implement the SEI.

```

@WebServiceClient(name = "TestEndpointService", targetNamespace =
"http://org.jboss.ws/wsref",
    wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-
webservicesref?wsdl")

public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT, TestEndpoint.class);
    }
}

```

11.15.3. WebServiceRef

The `WebServiceRef` annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the `javax.annotation.Resource` annotation in JSR-250 [5]

There are two uses to the `WebServiceRef` annotation:

1. To define a reference whose type is a generated service class. In this case, the type and value element will both refer to the generated service class type. Moreover, if the reference type can be inferred by the field/method declaration the annotation is applied to, the type and value

elements MAY have the default value (Object.class, that is). If the type cannot be inferred, then at least the type element MUST be present with a non-default value.

2. To define a reference whose type is a SEI. In this case, the type element MAY be present with its default value if the type of the reference can be inferred from the annotated field/method declaration, but the value element MUST always be present and refer to a generated service class type (a subtype of `javax.xml.ws.Service`). The `wsdlLocation` element, if present, overrides the WSDL location information specified in the `WebService` annotation of the referenced generated service class.

```
public class EJB3Client implements EJB3Remote
{
    @WebServiceRef
    public TestEndpointService service4;

    @WebServiceRef
    public TestEndpoint port3;
```

WebServiceRef Customization

In jboss-5.0.x we offer a number of overrides and extensions to the `WebServiceRef` annotation. These include

- define the port that should be used to resolve a container-managed port
- define default Stub property settings for Stub objects
- define the URL of a final WSDL document to be used

Example:

```
<service-ref>
<service-ref-name>OrganizationService</service-ref-name>
<wsdl-override>file:/wsdlRepository/organization-service.wsdl</wsdl-
override>
</service-ref>
..
<service-ref>
<service-ref-name>OrganizationService</service-ref-name>
<config-name>Secure Client Config</config-name>
<config-file>META-INF/jbossws-client-config.xml</config-file>
<handler-chain>META-INF/jbossws-client-handlers.xml</handler-chain>
</service-ref>

<service-ref>
<service-ref-name>SecureService</service-ref-name>
<service-class-
name>org.jboss.tests.ws.jaxws.webserviceref.SecureEndpointService</service-
class-name>
<service-qname>{http://org.jboss.ws/wsref}SecureEndpointService</service-
qname>
<port-info>
<service-endpoint-
```

```

interface>org.jboss.tests.ws.jaxws.webservicesref.SecureEndpoint</service-
endpoint-interface>
<port-qname>{http://org.jboss.ws/wsref}SecureEndpointPort</port-qname>
<stub-property>
<name>javax.xml.ws.security.auth.username</name>
<value>kermit</value>
</stub-property>
<stub-property>
<name>javax.xml.ws.security.auth.password</name>
<value>thefrog</value>
</stub-property>
</port-info>
</service-ref>

```

For details please see **service-ref_5_0.dtd** in the jboss docs directory.

11.15.4. Dispatch

XMLWeb Services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The Dispatch interface provides support for this mode of interaction.

Dispatch supports two usage modes, identified by the constants `javax.xml.ws.Service.Mode.MESSAGE` and `javax.xml.ws.Service.Mode.PAYLOAD` respectively:

Message In this mode, client applications work directly with protocol-specific message structures. E.g., when used with a SOAP protocol binding, a client application would work directly with a SOAP message.

Message Payload In this mode, client applications work with the payload of messages rather than the messages themselves. E.g., when used with a SOAP protocol binding, a client application would work with the contents of the SOAP Body rather than the SOAP message as a whole.

Dispatch is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. Dispatch is a generic class that supports input and output of messages or message payloads of any type.

```

Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class,
Mode.PAYLOAD);

String payload = "<ns1:ping
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new
StringReader(payload)));

```

11.15.5. Asynchronous Invocations

The `BindingProvider` interface represents a component that provides a protocol binding for use by clients, it is implemented by proxies and is extended by the `Dispatch` interface.

`BindingProvider` instances may provide asynchronous operation capabilities. When used, asynchronous operation invocations are decoupled from the `BindingProvider` instance at invocation time such that the response context is not updated when the operation completes. Instead a separate response context is made available using the `Response` interface.

```
public void testInvokeAsync() throws Exception
{
    URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-samples-
asynchronous?wsdl");
    QName serviceName = new QName(targetNS, "TestEndpointService");
    Service service = Service.create(wsdlURL, serviceName);
    TestEndpoint port = service.getPort(TestEndpoint.class);

    Response response = port.echoAsync("Async");

    // access future
    String retStr = (String) response.get();
    assertEquals("Async", retStr);
}
```

11.15.6. Oneway Invocations

`@Oneway` indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method.

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
    private static String feedback;
    ..
    @WebMethod
    @Oneway
    public void ping()
    {
        log.info("ping");
        feedback = "ok";
    }
    ..
    @WebMethod
    public String feedback()
    {
        log.info("feedback");
        return feedback;
    }
}
```

11.16. COMMON API

This sections describes concepts that apply equally to [#Web Service Endpoints](#) and [#Web Service Clients](#)

11.16.1. Handler Framework

The handler framework is implemented by a JAX-WS protocol binding in both client and server side runtimes. Proxies, and Dispatch instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

11.16.1.1. Logical Handler

Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler`.

11.16.1.2. Protocol Handler

Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message. Protocol handlers are handlers that implement any interface derived from `javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

11.16.1.3. Service endpoint handlers

On the service endpoint, handlers are defined using the `@HandlerChain` annotation.

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
{
    ...
}
```

The location of the handler chain file supports 2 formats

1. An absolute `java.net.URL` in externalForm. (ex: <http://myhandlers.foo.com/handlerfile1.xml>)
2. A relative path from the source file or class file. (ex: `bar/handlerfile1.xml`)

11.16.1.4. Service client handlers

On the client side, handler can be configured using the `@HandlerChain` annotation on the SEI or dynamically using the API.

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain); // important!
```

11.16.2. Message Context

`MessageContext` is the super interface for all JAX-WS message contexts. It extends `Map<String, Object>` with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the `put` method to insert a property in the message context that one or more other handlers in the handler chain may subsequently obtain via the `get` method.

Properties are scoped as either `APPLICATION` or `HANDLER`. All properties are available to all handlers for an instance of an MEP on a particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during the execution of an MEP instance. `APPLICATION` scoped properties are also made available to client applications and service endpoint implementations. The default scope for a property is `HANDLER`.

11.16.2.1. Logical Message Context

[#Logical Handlers](#) are passed a message context of type `LogicalMessageContext` when invoked. `LogicalMessageContext` extends `MessageContext` with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of a message. A protocol binding defines what component of a message are available via a logical message context. The SOAP binding defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers whereas the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

11.16.2.2. SOAP Message Context

SOAP handlers are passed a `SOAPMessageContext` when invoked. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.

11.16.3. Fault Handling

An implementation may throw a `SOAPFaultException`

```
public void throwSoapFaultException()
{
    SOAPFactory factory = SOAPFactory.newInstance();
    SOAPFault fault = factory.createFault("this is a fault string!", new
    QName("http://foo", "FooCode"));
}
```



```

fault.setFaultActor("mr.actor");
fault.addDetail().addChildElement("test");
throw new SOAPFaultException(fault);
}

```

or an application specific user exception

```

public void throwApplicationException() throws UserException
{
    throw new UserException("validation", 123, "Some validation error");
}

```



NOTE

In case of the latter JBossWS generates the required fault wrapper beans at runtime if they are not part of the deployment

11.17. DATABINDING

11.17.1. Using JAXB with non annotated classes

Since 2.0.2

JAXB is heavily driven by Java Annotations on the Java Bindings. It currently doesn't support an external binding configuration. This recently became an issue for us on JBossESB since the JBossWS 2.0.0 native SOAP stack uses JAXB to perform the SOAP to Java bindings (see 1, 2). It's an issue for JBossESB simply because it needs to be able to support user definition of JBossWS native Webservice Endpoints (e.g. JSR 181) using Java typesets that have not been "JAXB Annotated" (see JAXB Introductions On JBossWS).

In order to support this, we built on a JAXB RI feature whereby it allows you to specify a RuntimeInlineAnnotationReader implementation during JAXBContext creation (see JAXBRIContext).

We call this feature "JAXB Annotation Introduction" and we've made it available for general consumption i.e. it can be checked out, built and used from SVN:

- <http://anonsvn.jboss.org/repos/jboss/ws/projects/jaxbintros/>

Complete documentation can be found here:

- [JAXB Introductions](#)

11.18. ATTACHMENTS

11.18.1. MTOM/XOP

This section describes Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP), a means of more efficiently serializing XML Infosets that have certain types of content. The related specifications are

- [SOAP Message Transmission Optimization Mechanism \(MTOM\)](#)
- [XML-binary Optimized Packaging \(XOP\)](#)

11.18.1.1. Supported MTOM parameter types

image/jpeg	java.awt.Image
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
application/octet-stream	javax.activation.DataHandler

The above table shows a list of supported endpoint parameter types. The recommended approach is to use the [javax.activation.DataHandler](#) classes to represent binary data as service endpoint parameters.



NOTE

Microsoft endpoints tend to send any data as application/octet-stream. The only Java type that can easily cope with this ambiguity is [javax.activation.DataHandler](#)

11.18.1.2. Enabling MTOM per endpoint

On the server side MTOM processing is enabled through the **@BindingType** annotation. JBossWS does handle SOAP1.1 and SOAP1.2. Both come with or without MTOM flavours:

MTOM enabled service implementations

```
package org.jboss.test.ws.jaxws.samples.xop.doclit;

import javax.ejb.Remote;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.BindingType;

@Remote
@WebService(targetNamespace = "http://org.jboss.ws/xop/doclit")
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT, parameterStyle =
SOAPBinding.ParameterStyle.BARE)
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
(1)
public interface MTOMEndpoint {

    [...]
}
```

1. The MTOM enabled SOAP 1.1 binding ID

MTOM enabled clients

Web service clients can use the same approach described above or rely on the **Binding** API to enable MTOM (Excerpt taken from the `org.jboss.test.ws.jaxws.samples.xop.doclit.XOPTestCase`):

```
[...]
Service service = Service.create(wsdlURL, serviceName);
port = service.getPort(MTOMEndpoint.class);

// enable MTOM
binding = (SOAPBinding)((BindingProvider)port).getBinding();
binding.setMTOMEnabled(true);
```



NOTE

You might as well use the JBossWS configuration templates to setup deployment defaults.

11.18.2. SwaRef

Since 2.0

[WS-I Attachment Profile 1.0](#) defines mechanism to reference MIME attachment parts using [swaRef](#). In this mechanism the content of XML element of type `wsa:swaRef` is sent as MIME attachment and the element inside SOAP Body holds the reference to this attachment in the CID URI scheme as defined by [RFC 2111](#).

11.18.2.1. Using SwaRef with JAX-WS endpoints

JAX-WS endpoints delegate all marshalling/unmarshalling to the JAXB API. The most simple way to enable SwaRef encoding for **DataHandler** types is to annotate a payload bean with the `@XmlAttachmentRef` annotation as shown below:

```
/**
 * Payload bean that will use SwaRef encoding
 */
@XmlRootElement
public class DocumentPayload
{
    private DataHandler data;

    public DocumentPayload()
    {
    }

    public DocumentPayload(DataHandler data)
    {
        this.data = data;
    }

    @XmlElement
    @XmlAttachmentRef
```

```
public DataHandler getData()
{
    return data;
}

public void setData(DataHandler data)
{
    this.data = data;
}
}
```

With document wrapped endpoints you may even specify the `@XmlAttachmentRef` annotation on the service endpoint interface:

```
@WebService
public interface DocWrappedEndpoint
{
    @WebMethod
    DocumentPayload beanAnnotation(DocumentPayload dhw, String test);

    @WebMethod
    @XmlAttachmentRef
    DataHandler parameterAnnotation(@XmlAttachmentRef DataHandler data, String
    test);
}
```

The message would then refer to the attachment part by CID:

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header/>
  <env:Body>
    <ns2:parameterAnnotation
      xmlns:ns2='http://swaref.samples.jaxws.ws.test.jboss.org/'>
      <arg0>cid:0-1180017772935-32455963@ws.jboss.org</arg0>
      <arg1>Wrapped test</arg1>
    </ns2:parameterAnnotation>
  </env:Body>
</env:Envelope>
```

11.18.2.2. Starting from WSDL

If you chose the contract first approach then you need to ensure that any element declaration that should use SwaRef encoding simply refers to `wsi:swaRef` schema type:

```
<element name="data" type="wsi:swaRef"
  xmlns:wsi="http://ws-i.org/profiles/basic/1.1/xsd"/>
```

Any `ws:swaRef` schema type would then be mapped to `DataHandler`.

11.19. TOOLS

The JAX-WS tools provided by JBossWS can be used in a variety of ways. First we will look at server-side development strategies, and then proceed to the client. When developing a Web Service Endpoint (the server-side) you have the option of starting from Java (bottom-up development), or from the abstract contract (WSDL) that defines your service (top-down development). If this is a new service (no existing contract), the bottom-up approach is the fastest route; you only need to add a few annotations to your classes to get a service up and running. However, if you are developing a service with an already defined contract, it is far simpler to use the top-down approach, since the provided tool will generate the annotated code for you.

Bottom-up use cases:

- Exposing an already existing EJB3 bean as a Web Service
- Providing a new service, and you want the contract to be generated for you

Top-down use cases:

- Replacing the implementation of an existing Web Service, and you can't break compatibility with older clients
- Exposing a service that conforms to a contract specified by a third party (e.g. a vender that calls you back using an already defined protocol).
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front

The following JAX-WS command line tools are included in JBossWS:

Command	Description
<code>wsprovide</code>	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
<code>wsconsume</code>	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development
<code>wstrunclient</code>	Executes a Java client (has a main method) using the JBossWS classpath.

11.19.1. Bottom-Up (Using `wsprovide`)

The bottom-up strategy involves developing the Java code for your service, and then annotating it using JAX-WS annotations. These annotations can be used to customize the contract that is generated for your service. For example, you can change the operation name to map to anything you like. However, all of the annotations have sensible defaults, so only the `@WebService` annotation is required.

This can be as simple as creating a single class:

```
package echo;
```

```
@javax.jws.WebService
public class Echo
{
    public String echo(String input)
    {
        return input;
    }
}
```

A JSE or EJB3 deployment can be built using this class, and it is the only Java code needed to deploy on JBossWS. The WSDL, and all other Java artifacts called "wrapper classes" will be generated for you at deploy time. This actually goes beyond the JAX-WS specification, which requires that wrapper classes be generated using an offline tool. The reason for this requirement is purely a vendor implementation problem, and since we do not believe in burdening a developer with a bunch of additional steps, we generate these as well. However, if you want your deployment to be portable to other application servers, you will unfortunately need to use a tool and add the generated classes to your deployment.

This is the primary purpose of the [wsprovide](#) tool, to generate portable JAX-WS artifacts. Additionally, it can be used to "provide" the abstract contract (WSDL file) for your service. This can be obtained by invoking [wsprovide](#) using the "-w" option:

```
$ javac -d . -classpath jboss-jaxws.jar Echo.java
$ wsprovide -w echo.Echo
Generating WSDL:
EchoService.wsdl
Writing Classes:
echo/jaxws/Echo.class
echo/jaxws/EchoResponse.class
```

Inspecting the WSDL reveals a service called EchoService:

```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
    <soap:address location='REPLACE_WITH_ACTUAL_URL' />
  </port>
</service>
```

As expected, this service defines one operation, "echo":

```
<portType name='Echo'>
  <operation name='echo' parameterOrder='echo'>
    <input message='tns:Echo_echo' />
    <output message='tns:Echo_echoResponse' />
  </operation>
</portType>
```

**NOTE**

Remember that **when deploying on JBossWS you do not need to run this tool**. You only need it for generating portable artifacts and/or the abstract contract for your service.

Let's create a POJO endpoint for deployment on JBoss AS. A simple web.xml needs to be created:

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

<servlet>
<servlet-name>Echo</servlet-name>
<servlet-class>echo.Echo</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>Echo</servlet-name>
<url-pattern>/Echo</url-pattern>
</servlet-mapping>
</web-app>
```

The web.xml and the single class can now be used to create a war:

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

The war can then be deployed:

```
cp echo.war jboss-as/server/default/deploy
```

This will internally invoke [wsprovide](#), which will generate the WSDL. If deployment was successful, and you are using the default settings, it should be available here: <http://localhost:8080/echo/Echo?wsdl>

For a portable JAX-WS deployment, the wrapper classes generated earlier could be added to the deployment.

11.19.2. Top-Down (Using wsconsume)

The top-down development strategy begins with the abstract contract for the service, which includes the WSDL file and zero or more schema files. The `wsconsume` tool is then used to consume this contract, and produce annotated Java classes (and optionally sources) that define it.



NOTE

`wsconsume` seems to have a problem with symlinks on unix systems

Using the WSDL file from the bottom-up example, a new Java implementation that adheres to this service can be generated. The `-k` option is passed to `wsconsume` to preserve the Java source files that are generated, instead of providing just classes:

```
$ wsconsume -k EchoService.wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The following table shows the purpose of each generated file:

File	Purpose
Echo.java	Service Endpoint Interface
Echo_Type.java	Wrapper bean for request message
EchoResponse.java	Wrapper bean for response message
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
EchoService.java	Used only by JAX-WS clients

Examining the Service Endpoint Interface reveals annotations that are more explicit than in the class written by hand in the bottom-up example, however, these evaluate to the same contract:

```
@WebService(name = "Echo", targetNamespace = "http://echo/")
public interface Echo {
    @WebMethod
```



```

@WebResult(targetNamespace = "")
@RequestWrapper(localName = "echo", targetNamespace = "http://echo/",
  className = "echo.Echo_Type")
@ResponseWrapper(localName = "echoResponse", targetNamespace =
  "http://echo/", className = "echo.EchoResponse")
public String echo(
  @WebParam(name = "arg0", targetNamespace = "")
  String arg0);
}

```

The only missing piece (besides the packaging) is the implementation class, which can now be written, using the above interface.

```

package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo
{
  public String echo(String arg0)
  {
    return arg0;
  }
}

```

11.19.3. Client Side

Before going to detail on the client-side it is important to understand the decoupling concept that is central to Web Services. Web Services are not the best fit for internal RPC, even though they can be used in this way. There are much better technologies for this (CORBA, and RMI for example). Web Services were designed specifically for interoperable coarse-grained correspondence. There is no expectation or guarantee that any party participating in a Web Service interaction will be at any particular location, running on any particular OS, or written in any particular programming language. So because of this, it is important to clearly separate client and server implementations. The only thing they should have in common is the abstract contract definition. If, for whatever reason, your software does not adhere to this principal, then you should not be using Web Services. For the above reasons, the **recommended methodology for developing a client is** to follow **the top-down approach**, even if the client is running on the same server.

Let's repeat the process of the top-down section, although using the deployed WSDL, instead of the one generated offline by [wsprovide](#). The reason why we do this is just to get the right value for soap:address. This value must be computed at deploy time, since it is based on container configuration specifics. You could of course edit the WSDL file yourself, although you need to ensure that the path is correct.

Offline version:

```

<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
    <soap:address location='REPLACE_WITH_ACTUAL_URL' />
  </port>
</service>

```

```
</port>
</service>
```

Online version:

```
<service name="EchoService">
  <port binding="tns:EchoBinding" name="EchoPort">
    <soap:address location="http://localhost.localdomain:8080/echo/Echo"/>
  </port>
</service>
```

Using the online deployed version with [wsconsume](#):

```
$ wsconsume -k http://localhost:8080/echo/Echo?wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The one class that was not examined in the top-down section, was `EchoService.java`. Notice how it stores the location the WSDL was obtained from.

```
@WebServiceClient(name = "EchoService", targetNamespace = "http://echo/",
wsdlLocation = "http://localhost:8080/echo/Echo?wsdl")
public class EchoService extends Service
{
    private final static URL ECHOSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:8080/echo/Echo?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        ECHOSERVICE_WSDL_LOCATION = url;
    }

    public EchoService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }
}
```

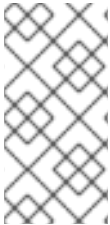
```

public EchoService() {
    super(ECHOSERVICE_WSDL_LOCATION, new QName("http://echo/",
        "EchoService"));
}

@WebEndpoint(name = "EchoPort")
public Echo getEchoPort() {
    return (Echo)super.getPort(new QName("http://echo/", "EchoPort"),
        Echo.class);
}
}

```

As you can see, this generated class extends the main client entry point in JAX-WS, `javax.xml.ws.Service`. While you can use `Service` directly, this is far simpler since it provides the configuration info for you. The only method we really care about is the `getEchoPort()` method, which returns an instance of our Service Endpoint Interface. Any WS operation can then be called by just invoking a method on the returned interface.



NOTE

It's not recommended to refer to a remote WSDL URL in a production application. This causes network I/O every time you instantiate the Service Object. Instead, use the tool on a saved local copy, or use the URL version of the constructor to provide a new WSDL location.

All that is left to do, is write and compile the client:

```

import echo.*;
..
public class EchoClient
{
    public static void main(String args[])
    {
        if (args.length != 1)
        {
            System.err.println("usage: EchoClient <message>");
            System.exit(1);
        }

        EchoService service = new EchoService();
        Echo echo = service.getEchoPort();
        System.out.println("Server said: " + echo.echo(args[0]));
    }
}

```

It can then be easily executed using the [wsrunclient](#) tool. This is just a convenience tool that invokes java with the needed classpath:

```

$ wsrunclient EchoClient 'Hello World!'
Server said: Hello World!

```

It is easy to change the endpoint address of your operation at runtime, setting the `ENDPOINT_ADDRESS_PROPERTY` as shown below:

```
...
EchoService service = new EchoService();
Echo echo = service.getEchoPort();

/* Set NEW Endpoint Location */
String endpointURL = "http://NEW_ENDPOINT_URL";
BindingProvider bp = (BindingProvider)echo;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    endpointURL);

System.out.println("Server said: " + echo.echo(args[0]));
...
```

11.19.4. Command-line & Ant Task Reference

- [wsconsume reference page](#)
- [wsprovide reference page](#)
- [wsrunclient reference page](#)

11.19.5. JAX-WS binding customization

An introduction to binding customizations:

- <http://java.sun.com/webservices/docs/2.0/jaxws/customizations.html>

The schema for the binding customization files can be found here:

- [binding customization](#)

11.20. WEB SERVICE EXTENSIONS

11.20.1. WS-Addressing

This section describes how [WS-Addressing](#) can be used to provide a stable service endpoint.

11.20.1.1. Specifications

WS-Addressing is defined by a combination of the following specifications from the W3C Candidate Recommendation 17 August 2005. The WS-Addressing API is standardized by [JSR-261 - Java API for XML Web Services Addressing](#)

- [Web Services Addressing 1.0 - Core](#)
- [Web Services Addressing 1.0 - SOAP Binding](#)

11.20.1.2. Addressing Endpoint

The following endpoint implementation has a set of operation for a typical stateful shopping chart application.

```
@WebService(name = "StatefulEndpoint", targetNamespace =
"http://org.jboss.ws/samples/wsaddressing
", serviceName = "TestService")
@EndpointConfig(configName = "Standard WSAAddressing Endpoint")
@HandlerChain(file = "WEB-INF/jaxws-handlers.xml")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class StatefulEndpointImpl implements StatefulEndpoint,
ServiceLifecycle
{
@WebMethod
public void addItem(String item)
{ ... }

@WebMethod
public void checkout()
{ ... }

@WebMethod
public String.getItems()
{ ... }
}
```

It uses the [JAX-WS Endpoint Configuration# Standard WSAAddressing Endpoint](#) to enable the server side addressing handler. It processes the incoming WS-Addressing header elements and provides access to them through the JSR-261 API.

The endpoint handler chain

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
javaee_web_services_1_2.xsd">

<handler-chain>
<protocol-bindings>##SOAP11_HTTP</protocol-bindings>
<handler>
<handler-name>Application Server Handler</handler-name>
<handler-
class>org.jboss.test.ws.jaxws.samples.wsaddressing.ServerHandler</handler-
class>
</handler>
</handler-chain>

</handler-chains>
```

defines an application specific handler that assigns/processes stateful client ids.

11.20.1.3. Addressing Client

On the client side there are similar handlers that does the reverse. It uses the JSR-261 API to add WS-Addressing header elements including the clientid association.

The client sets a custom handler chain in the binding

```
Service service = Service.create(wsdlURL, serviceName);
port1 = (StatefulEndpoint)service.getPort(StatefulEndpoint.class);
BindingProvider bindingProvider = (BindingProvider)port1;

List<Handler> customHandlerChain = new ArrayList<Handler>();
customHandlerChain.add(new ClientHandler());
customHandlerChain.add(new WSAddressingClientHandler());
bindingProvider.getBinding().setHandlerChain(customHandlerChain);
```

The WSAddressingClientHandler is provided by JBossWS and reads/writes the addressing properties and puts them into the message context.

A client connecting to the stateful endpoint

```
public class AddressingStatefulTestCase extends JBossWSTest
{
    public void testAddItem() throws Exception
    {
        port1.addItem("Ice Cream");
        port1.addItem("Ferrari");

        port2.addItem("Mars Bar");
        port2.addItem("Porsche");
    }

    public void testGetItems() throws Exception
    {
        String items1 = port1.getItems();
        assertEquals("[Ice Cream, Ferrari]", items1);

        String items2 = port2.getItems();
        assertEquals("[Mars Bar, Porsche]", items2);
    }
}
```

SOAP message exchange

Below you see the SOAP messages that are being exchanged.

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
    <wsa:To>uri:jbossws-samples-wsaddr/TestService</wsa:To>
    <wsa:Action>http://org.jboss.ws/addressing/stateful/action</wsa:Action>
    <wsa:ReferenceParameters>
```

```

<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</wsa:ReferenceParameters>
</env:Header>
<env:Body>
<ns1:addItem xmlns:ns1='http://org.jboss.ws/samples/wsaddr'>
<String_1>Ice Cream</String_1>
</ns1:addItem>
</env:Body>
</env:Envelope>

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/actionReply</wsa:Actio
n>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</env:Header>
<env:Body>
<ns1:addItemResponse xmlns:ns1='http://org.jboss.ws/samples/wsaddr' />
</env:Body>
</env:Envelope>

...

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>uri:jbossws-samples-wsaddr/TestService</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/action</wsa:Action>
<wsa:ReferenceParameters>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</wsa:ReferenceParameters>
</env:Header>
<env:Body>
<ns1:getItems xmlns:ns1='http://org.jboss.ws/samples/wsaddr' />
</env:Body>
</env:Envelope>

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/actionReply</wsa:Actio
n>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</env:Header>
<env:Body>
<ns1:getItemsResponse xmlns:ns1='http://org.jboss.ws/samples/wsaddr'>
<result>[Ice Cream, Ferrari]</result>
</ns1:getItemsResponse>
</env:Body>
</env:Envelope>

```

11.20.2. WS-BPEL

WS-BPEL is not supported with JAX-WS, please refer to [JAX-RPC User Guide#WS-BPEL](#).

11.20.3. WS-Security

WS-Security addresses message level security. It standardizes authorization, encryption, and digital signature processing of web services. Unlike transport security models, such as SSL, WS-Security applies security directly to the elements of the web service message. This increases the flexibility of your web services, by allowing any message model to be used (point to point, multi-hop relay, etc).

This chapter describes how to use WS-Security to sign and encrypt a simple SOAP message.

Specifications

WS-Security is defined by the combination of the following specifications:

- [SOAP Message Security 1.0](#)
- [Username Token Profile 1.0](#)
- [X.509 Token Profile 1.0](#)
- [W3C XML Encryption](#)
- [W3C XML Signature](#)
- [Basic Security Profile 1.0 \(Still in Draft\)](#)

11.20.3.1. Endpoint configuration

JBossWS uses handlers to identify ws-security encoded requests and invoke the security components to sign and encrypt messages. In order to enable security processing, the client and server side need to include a corresponding handler configuration. The preferred way is to reference a predefined [JAX-WS Endpoint Configuration](#) or [JAX-WS Client Configuration](#) respectively.



NOTE

You need to setup both the endpoint configuration and the WSSE declarations. That's two separate steps.

11.20.3.2. Server side WSSE declaration (jboss-wsse-server.xml)

In this example we configure both the client and the server to sign the message body. Both also require this from each other. So, if you remove either the client or the server security deployment descriptor, you will notice that the other party will throw a fault explaining that the message did not conform to the proper security requirements.

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
(1) <key-store-file>WEB-INF/wsse.keystore</key-store-file>
(2) <key-store-password>jbossws</key-store-password>
(3) <trust-store-file>WEB-INF/wsse.truststore</trust-store-file>
(4) <trust-store-password>jbossws</trust-store-password>
(5) <config>
(6)   <sign type="x509v3" alias="wsse"/>
(7)   <requires>
```



```
(8)      <signature/>
</requires>
</config>
</jboss-ws-security>
```

1. This specifies that the key store we wish to use is WEB-INF/wsse.keystore, which is located in our war file.
2. This specifies that the store password is "jbossws". Password can be encrypted using the {EXT} and {CLASS} commands. Please see samples for their usage.
3. This specifies that the trust store we wish to use is WEB-INF/wsse.truststore, which is located in our war file.
4. This specifies that the trust store password is also "jbossws". Password can be encrypted using the {EXT} and {CLASS} commands. Please see samples for their usage.
5. Here we start our root config block. The root config block is the default configuration for all services in this war file.
6. This means that the server must sign the message body of all responses. Type means that we are to use a X.509v3 certificate (a standard certificate). The alias option says that the certificate/key pair to use for signing is in the key store under the "wsse" alias
7. Here we start our optional requires block. This block specifies all security requirements that must be met when the server receives a message.
8. This means that all web services in this war file require the message body to be signed.

By default an endpoint does not use the WS-Security configuration. Use the proprietary `@EndpointConfig` annotation to set the config name. See [JAX-WS_Endpoint_Configuration](#) for the list of available config names.

```
@WebService
@EndpointConfig(configName = "Standard WSSecurity Endpoint")
public class HelloJavaBean
{
    ...
}
```

11.20.3.3. Client side WSSE declaration (jboss-wsse-client.xml)

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
(1)  <config>
(2)    <sign type="x509v3" alias="wsse"/>
(3)    <requires>
(4)      <signature/>
</requires>
```

```
</config>
</jboss-ws-security>
```

1. Here we start our root config block. The root config block is the default configuration for all web service clients (Call, Proxy objects).
2. This means that the client must sign the message body of all requests it sends. Type means that we are to use a X.509v3 certificate (a standard certificate). The alias option says that the certificate/key pair to use for signing is in the key store under the "wsse" alias
3. Here we start our optional requires block. This block specifies all security requirements that must be met when the client receives a response.
4. This means that all web service clients must receive signed response messages.

11.20.3.3.1. Client side key store configuration

We did not specify a key store or trust store, because client apps instead use the wsse System properties instead. If this was a web or ejb client (meaning a webservice client in a war or ejb jar file), then we would have specified them in the client descriptor.

Here is an excerpt from the JBossWS samples:

```
<sysproperty key="org.jboss.ws.wsse.keyStore"
value="${tests.output.dir}/resources/jaxrpc/samples/wssecurity/wsse.keystore"/>
<sysproperty key="org.jboss.ws.wsse.trustStore"
value="${tests.output.dir}/resources/jaxrpc/samples/wssecurity/wsse.truststore"/>
<sysproperty key="org.jboss.ws.wsse.keyStorePassword" value="jbossws"/>
<sysproperty key="org.jboss.ws.wsse.trustStorePassword" value="jbossws"/>
<sysproperty key="org.jboss.ws.wsse.keyStoreType" value="jks"/>
<sysproperty key="org.jboss.ws.wsse.trustStoreType" value="jks"/>
```

SOAP message exchange

Below you see the incoming SOAP message with the details of the security headers omitted. The idea is, that the SOAP body is still plain text, but it is signed in the security header and can therefore not be manipulated in transit.

Incoming SOAPMessage

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Header>
<wsse:Security env:mustUnderstand="1" ...>
<wsu:Timestamp wsu:Id="timestamp">...</wsu:Timestamp>
<wsse:BinarySecurityToken ...>
...
</wsse:BinarySecurityToken>
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
...
</ds:Signature>
```

```

</wsse:Security>
</env:Header>
<env:Body wsu:Id="element-1-1140197309843-12388840" ...>
<ns1:echoUserType xmlns:ns1="http://org.jboss.ws/samples/wssecurity">
<UserType_1 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<msg>Kermit</msg>
</UserType_1>
</ns1:echoUserType>
</env:Body>
</env:Envelope>

```

11.20.3.4. Installing the BouncyCastle JCE provider (JDK 1.4)

The information below has originally been provided by [The Legion of the Bouncy Castle](#).

The provider can be configured as part of your environment via static registration by adding an entry to the java.security properties file (found in \$JAVA_HOME/jre/lib/security/java.security, where \$JAVA_HOME is the location of your JDK/JRE distribution). You'll find detailed instructions in the file but basically it comes down to adding a line:

```
security.provider.<n>=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Where <n> is the preference you want the provider at.



NOTE

Issues may arise if the Sun provided providers are not first.

Where you put the jar is mostly up to you, although with jdk1.4 the best (and in some cases only) place to have it is in \$JAVA_HOME/jre/lib/ext. Under Windows there will normally be a JRE and a JDK install of Java if you think you have installed it correctly and it still doesn't work chances are you have added the provider to the installation not being used.

11.20.3.5. Keystore, truststore - What?



NOTE

If you having a hard time understanding how the different trust- and keystore configurations are used for signature and encryption, then read this thread first: <http://www.jboss.org/index.html?module=bb&op=viewtopic&t=94406>

11.20.4. WS-Transaction

Support for the WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity specifications will be provided by technology recently acquired from Arjuna Technologies Ltd. This technology will be present within the JBoss Transactions 4.2.1 release. Further information can be obtained from the [JBoss Transactions Project](#)

11.20.5. XML Registries

J2EE 1.4 mandates support for Java API for XML Registries (JAXR). Inclusion of a XML Registry with the J2EE 1.4 certified Application Server is optional. Starting jboss-4.0.2, JBoss ships a UDDI v2.0 compliant registry, the Apache jUDDI registry. We also provide support for JAXR Capability Level 0 (UDDI Registries) via integration of Apache Scout.

This chapter describes how to configure the jUDDI registry in JBoss and some sample code outlines for using JAXR API to publish and query the jUDDI registry.

11.20.5.1. Apache jUDDI Configuration

Configuration of the jUDDI registry happens via an MBean Service that is deployed in the juddi-service.sar archive in the "all" configuration. The configuration of this service can be done in the jboss-service.xml of the META-INF directory in the juddi-service.sar

Let us look at the individual configuration items that can be changed.

DataSources configuration

```
<!-- Datasource to Database-->
<attribute name="DataSourceUrl">java:/DefaultDS</attribute>
```

Database Tables (Should they be created on start, Should they be dropped on stop, Should they be dropped on start etc)

```
<!-- Should all tables be created on Start-->
<attribute name="CreateOnStart">false</attribute>
<!-- Should all tables be dropped on Stop-->
<attribute name="DropOnStop">true</attribute>
<!-- Should all tables be dropped on Start-->
<attribute name="DropOnStart">false</attribute>
```

JAXR Connection Factory to be bound in JNDI. (Should it be bound? and under what name?)

```
<!-- Should I bind a Context to which JaxrConnectionFactory bound-->
<attribute name="ShouldBindJaxr">true</attribute>

<!-- Context to which JaxrConnectionFactory to bind to.
If you have remote clients, please bind it to the global namespace(default
behavior).
To just cater to clients running on the same VM as JBoss, change to
java:/JAXR -->
<attribute name="BindJaxr">JAXR</attribute>
```

Other common configuration:

Add authorized users to access the jUDDI registry. (Add a sql insert statement in a single line)

Look at the script META-INF/ddl/juddi_data.ddl for more details. Example

```
for a user 'jboss'
```

```
INSERT INTO PUBLISHER (PUBLISHER_ID,PUBLISHER_NAME,
EMAIL_ADDRESS,IS_ENABLED,IS_ADMIN)
VALUES ('jboss','JBoss User','jboss@xxx','true','true');
```

11.20.5.2. JBoss JAXR Configuration

In this section, we will discuss the configuration needed to run the JAXR API. The JAXR configuration relies on System properties passed to the JVM. The System properties that are needed are:

```
javax.xml.registry.ConnectionFactoryClass=org.apache.ws.scout.registry.ConnectionFactoryImpl
jaxr.query.url=http://localhost:8080/juddi/inquiry
jaxr.publish.url=http://localhost:8080/juddi/publish
juddi.proxy.transportClass=org.jboss.jaxr.juddi.transport.SaaJTransport
```

Please remember to change the hostname from "localhost" to the hostname of the UDDI service/JBoss Server.

You can pass the System Properties to the JVM in the following ways:

- When the client code is running inside JBoss (maybe a servlet or an EJB). Then you will need to pass the System properties in the run.sh/run.bat scripts to the java process via the "-D" option.
- When the client code is running in an external JVM. Then you can pass the properties either as "-D" options to the java process or explicitly set them in the client code(not recommended).

```
System.setProperty(propertyname, propertyvalue);
```

11.20.5.3. JAXR Sample Code

There are two categories of API: JAXR Publish API and JAXR Inquiry API. The important JAXR interfaces that any JAXR client code will use are the following.

- [javax.xml.registry.RegistryService](#) From J2EE 1.4 JavaDoc: "This is the principal interface implemented by a JAXR provider. A registry client can get this interface from a Connection to a registry. It provides the methods that are used by the client to discover various capability specific interfaces implemented by the JAXR provider."
- [javax.xml.registry.BusinessLifeCycleManager](#) From J2EE 1.4 JavaDoc: "The BusinessLifeCycleManager interface, which is exposed by the Registry Service, implements the life cycle management functionality of the Registry as part of a business level API. Note that there is no authentication information provided, because the Connection interface keeps that state and context on behalf of the client."
- [javax.xml.registry.BusinessQueryManager](#) From J2EE 1.4 JavaDoc: "The BusinessQueryManager interface, which is exposed by the Registry Service, implements the business style query interface. It is also referred to as the focused query interface."

Let us now look at some of the common programming tasks performed while using the JAXR API:

Getting a JAXR Connection to the registry.

```
String queryurl = System.getProperty("jaxr.query.url",
"http://localhost:8080/juddi/inquiry");
String puburl = System.getProperty("jaxr.publish.url",
"http://localhost:8080/juddi/publish");
..
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL", queryurl);
props.setProperty("javax.xml.registry.lifeCycleManagerURL", puburl);

String transportClass = System.getProperty("juddi.proxy.transportClass",
"org.jboss.jaxr.juddi.transport.SaajTransport");
System.setProperty("juddi.proxy.transportClass", transportClass);

// Create the connection, passing it the configuration properties
factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

Authentication with the registry.

```
/**
 * Does authentication with the uddi registry
 */
protected void login() throws JAXRException
{
    PasswordAuthentication passwdAuth = new PasswordAuthentication(userid,
passwd.toCharArray());
    Set creds = new HashSet();
    creds.add(passwdAuth);

    connection.setCredentials(creds);
}
```

Save a Business

```
/**
 * Creates a Jaxr Organization with 1 or more services
 */
protected Organization createOrganization(String orgname) throws
JAXRException
{
    Organization org = blm.createOrganization(getIString(orgname));
    org.setDescription(getIString("JBoss Inc"));
    Service service = blm.createService(getIString("JBoss JAXR Service"));
    service.setDescription(getIString("Services of XML Registry"));
}
```

```

//Create serviceBinding
ServiceBinding serviceBinding = blm.createServiceBinding();
serviceBinding.setDescription(blm.createInternationalString("Test Service
Binding"));

//Turn validation of URI off
serviceBinding.setValidateURI(false);
serviceBinding.setAccessURI("http://testjboss.org");
..
// Add the serviceBinding to the service
service.addServiceBinding(serviceBinding);

User user = blm.createUser();
org.setPrimaryContact(user);
PersonName personName = blm.createPersonName("Anil S");
TelephoneNumber telephoneNumber = blm.createTelephoneNumber();
telephoneNumber.setNumber("111-111-7777");
telephoneNumber.setType(null);
PostalAddress address = blm.createPostalAddress("111", "My Drive",
"BuckHead", "GA", "USA", "1111-111", "");
Collection postalAddresses = new ArrayList();
postalAddresses.add(address);
Collection emailAddresses = new ArrayList();
EmailAddress emailAddress = blm.createEmailAddress("anil@apache.org");
emailAddresses.add(emailAddress);

Collection numbers = new ArrayList();
numbers.add(telephoneNumber);
user.setPersonName(personName);
user.setPostalAddresses(postalAddresses);
user.setEmailAddresses(emailAddresses);
user.setTelephoneNumbers(numbers);

ClassificationScheme cScheme = getClassificationScheme("ntis-gov:naics",
""");
Key cKey = blm.createKey("uuid:C0B9FE13-324F-413D-5A5B-2004DB8E5CC2");
cScheme.setKey(cKey);
Classification classification = blm.createClassification(cScheme,
"Computer Systems Design and Related Services",
"5415");
org.addClassification(classification);
ClassificationScheme cScheme1 = getClassificationScheme("D-U-N-S", "");
Key cKey1 = blm.createKey("uuid:3367C81E-FF1F-4D5A-B202-3EB13AD02423");
cScheme1.setKey(cKey1);
ExternalIdentifier ei = blm.createExternalIdentifier(cScheme1, "D-U-N-S
number", "08-146-6849");
org.addExternalIdentifier(ei);
org.addService(service);
return org;
}

```

Query a Business

```
/**
```

```
* Locale aware Search a business in the registry
*/
public void searchBusiness(String bizname) throws JAXRException
{
    try
    {
        // Get registry service and business query manager
        this.getJAXREssentials();

        // Define find qualifiers and name patterns
        Collection findQualifiers = new ArrayList();
        findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);
        Collection namePatterns = new ArrayList();
        String pattern = "%" + bizname + "%";
        LocalizedString ls = blm.createLocalizedString(Locale.getDefault(),
        pattern);
        namePatterns.add(ls);

        // Find based upon qualifier type and values
        BulkResponse response = bpm.findOrganizations(findQualifiers,
        namePatterns, null, null, null, null);

        // check how many organisation we have matched
        Collection orgs = response.getCollection();
        if (orgs == null)
        {
            log.debug(" -- Matched 0 orgs");
        }
        else
        {
            log.debug(" -- Matched " + orgs.size() + " organizations -- ");

            // then step through them
            for (Iterator orgIter = orgs.iterator(); orgIter.hasNext();)
            {
                Organization org = (Organization)orgIter.next();
                log.debug("Org name: " + getName(org));
                log.debug("Org description: " + getDescription(org));
                log.debug("Org key id: " + getKey(org));
                checkUser(org);
                checkServices(org);
            }
        }
        finally
        {
            connection.close();
        }
    }
}
```

For more examples of code using the JAXR API, please refer to the resources in the Resources Section.

11.20.5.4. Troubleshooting

- **I cannot connect to the registry from JAXR.** Please check the inquiry and publish url passed to the JAXR ConnectionFactory.
- **I cannot connect to the jUDDI registry.** Please check the jUDDI configuration and see if there are any errors in the server.log. And also remember that the jUDDI registry is available only in the "all" configuration.
- **I cannot authenticate to the jUDDI registry.** Have you added an authorized user to the jUDDI database, as described earlier in the chapter?
- **I would like to view the SOAP messages in transit between the client and the UDDI Registry.** Please use the tcpmon tool to view the messages in transit. [TCPMon](#)

11.20.5.5. Resources

- [JAXR Tutorial and Code Camps](#)
- [J2EE 1.4 Tutorial](#)
- [J2EE Web Services by Richard Monson-Haefel](#)

11.21. JBOSSWS EXTENSIONS

This section describes proprietary JBoss extensions to JAX-WS.

11.21.1. Proprietary Annotations

For the set of standard annotations, please have a look at [JAX-WS Annotations](#)

11.21.1.1. EndpointConfig

```
/**
 * Defines an endpoint or client configuration.
 * This annotation is valid on an endpoint implementaion bean or a SEI.
 *
 * @author Heiko.Braun@jboss.org
 * @since 16.01.2007
 */
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = { ElementType.TYPE })
public @interface EndpointConfig {
    ...
}

/**
 * The optional config-name element gives the configuration name that must
 * be present in
 * the configuration given by element config-file.
 *
 * Server side default: Standard Endpoint
 * Client side default: Standard Client
 */
String configName() default "";
...
}

/**
 * The optional config-file element is a URL or resource name for the
```

```
configuration.  
*  
* Server side default: standard-jaxws-endpoint-config.xml  
* Client side default: standard-jaxws-client-config.xml  
*/  
String configFile() default "";  
}
```

11.21.1.2. WebContext

```
/**  
 * Provides web context specific meta data to EJB based web service  
 endpoints.  
 *  
 * @author thomas.diesler@jboss.org  
 * @since 26-Apr-2005  
 */  
@Retention(value = RetentionPolicy.RUNTIME)  
@Target(value = { ElementType.TYPE })  
public @interface WebContext {  
    ...  
}/**  
 * The contextRoot element specifies the context root that the web service  
 endpoint is deployed to.  
 * If it is not specified it will be derived from the deployment short  
 name.  
 *  
 * Applies to server side port components only.  
 */  
String contextRoot() default "";  
...  
/**  
 * The virtual hosts that the web service endpoint is deployed to.  
 *  
 * Applies to server side port components only.  
 */  
String[] virtualHosts() default {};  
  
/**  
 * Relative path that is appended to the contextRoot to form fully  
 qualified  
 * endpoint address for the web service endpoint.  
 *  
 * Applies to server side port components only.  
 */  
String urlPattern() default "";  
  
/**  
 * The authMethod is used to configure the authentication mechanism for the  
 web service.  
 * As a prerequisite to gaining access to any web service which are  
 protected by an authorization  
 * constraint, a user must have authenticated using the configured
```

```

mechanism.
*
* Legal values for this element are "BASIC", or "CLIENT-CERT".
*/
String authMethod() default "";

/**
 * The transportGuarantee specifies that the communication
 * between client and server should be NONE, INTEGRAL, or
 * CONFIDENTIAL. NONE means that the application does not require any
 * transport guarantees. A value of INTEGRAL means that the application
 * requires that the data sent between the client and server be sent in
 * such a way that it can't be changed in transit. CONFIDENTIAL means
 * that the application requires that 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 will indicate that the use of SSL is required.
 */
String transportGuarantee() default "";

/**
 * A secure endpoint does not by default publish it's wsdl on an unsecure
 * transport.
 * You can override this behaviour by explicitly setting the
 * secureWSDLAccess flag to false.
 *
 * Protect access to WSDL. See http://jira.jboss.org/jira/browse/JBWS-723
 */
boolean secureWSDLAccess() default true;
}

```

11.21.1.3. SecurityDomain

```

/**
 * Annotation for specifying the JBoss security domain for an EJB
 *
 * @author <a href="mailto:bill@jboss.org">Bill Burke</a>
 */
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface SecurityDomain
{
    /**
     * The required name for the security domain.
     *
     * Do not use the JNDI name
     *
     * Good: "MyDomain"
     * Bad: "java:/jaas/MyDomain"
     */
    String value();

    /**
     * The name for the unauthenticated principal

```

```
*/  
String unauthenticatedPrincipal() default "";  
}
```

CHAPTER 12. ADDITIONAL SERVICES

This chapter discusses useful MBean services that are not discussed elsewhere either because they are utility services not necessary for running JBoss, or they don't fit into a current section of the book.

12.1. MEMORY AND THREAD MONITORING

The `jboss.system:type=ServerInfo` MBean provides several attributes that can be used to monitor the thread and memory usage in a JBoss instance. These attributes can be monitored in many ways: through the JMX Console, from a third-party JMX management tool, from shell scripts using the `twiddle` command, etc... The most interesting attributes are shown below.

FreeMemory

This is the current free memory available in the JVM.

ActiveThreadCount

This is the number of active threads in the JVM.

ActiveThreadGroupCount

This is the number of active thread groups in the JVM.

These are useful metrics for monitoring and alerting, but developers and administrators need a little more insight than this. The Java 5 JVMs from Sun provide more detailed information about the current state of the JVM. Some of these details are exposed by JBoss through operations on the `SystemInfo` MBean.

listMemoryPools

This operation shows the size and current usage of all JVM memory pools. This operation is only available when using Java 5.

listThreadDump

This operation shows all threads currently running in the JVM. When using Java 5, JBoss will display a complete stack trace for each thread, showing you exactly what code each thread is executing.

listThreadCpuUtilization

This operation shows all threads currently running in the JVM along with the total CPU time each thread has used. The operation is only available in Java 5.

12.2. THE LOG4J SERVICE

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

- **ConfigurationURL**: The URL for the log4j configuration file. This can refer to either a XML document parsed by the `org.apache.log4j.xml.DOMConfigurator` or a Java properties file parsed by the `org.apache.log4j.PropertyConfigurator`. The type of the file is determined by the URL content type, or if this is null, the file extension. The default setting of `resource:log4j.xml` refers to the `conf/log4j.xml` file of the active server configuration file set.

- **RefreshPeriod:** The time in seconds between checks for changes in the log4j configuration specified by the **ConfigurationURL** attribute. The default value is 60 seconds.
- **CatchSystemErr:** This boolean flag if true, indicates if the **System.err** stream should be redirected onto a log4j category called **STDERR**. The default is true.
- **CatchSystemOut:** This boolean flag if true, indicates if the **System.out** stream should be redirected onto a log4j category called **STDOUT**. The default is true.
- **Log4jQuietMode:** This boolean flag if true, sets the **org.apache.log4j.helpers.LogLog.setQuietMode**. As of log4j1.2.8 this needs to be set to avoid a possible deadlock on exception at the appender level. See bug#696819.

12.3. SYSTEM PROPERTIES MANAGEMENT

The management of system properties can be done using the system properties service. It supports setting of the VM global property values just as **java.lang.System.setProperty** method and the VM command line arguments do.

Its configurable attributes include:

- **Properties:** a specification of multiple property **name=value** pairs using the **java.util.Properties.load(java.io.InputStream)** method format. Each **property=value** statement is given on a separate line within the body of the **Properties** attribute element.
- **URLList:** a comma separated list of URL strings from which to load properties file formatted content. If a component in the list is a relative path rather than a URL it will be treated as a file path relative to the **<jboss-dist>/server/<config>** directory. For example, a component of **conf/local.properties** would be treated as a file URL that points to the **<jboss-dist>/server/production/conf/local.properties** file when running with the **default** configuration file set.

The following illustrates the usage of the system properties service with an external properties file.

```
<mbean code="org.jboss.varia.property.SystemPropertiesService"
      name="jboss.util:type=Service,name=SystemProperties">

    <!-- Load properties from each of the given comma separated URLs -->
    <attribute name="URLList">
        http://somehost/some-location.properties,
        ./conf/somelocal.properties
    </attribute>
</mbean>
```

The following illustrates the usage of the system properties service with an embedded properties list.

```
<mbean code="org.jboss.varia.property.SystemPropertiesService"
      name="jboss.util:type=Service,name=SystemProperties">
    <!-- Set properties using the properties file style. -->
    <attribute name="Properties">
        property1=This is the value of my property
        property2=This is the value of my other property
    </attribute>
```

```
</mbean>
```

12.4. PROPERTY EDITOR MANAGEMENT

In JBoss, JavaBean property editors are used for reading data types from service files and for editing values in the JMX console. The `java.bean.PropertyEditorManager` class controls the `java.bean.PropertyEditor` instances in the system. The property editor manager can be managed in JBoss using the `org.jboss.varia.property.PropertyEditorManagerService` MBean. The property editor manager service is configured in `deploy/properties-service.xml` and supports the following attributes:

- **BootstrapEditors:** This is a listing of `property_editor_class=editor_value_type_class` pairs defining the property editor to type mappings that should be preloaded into the property editor manager. The value type of this attribute is a string so that it may be set from a string without requiring a custom property editor.
- **Editors:** This serves the same function as the **BootstrapEditors** attribute, but its type is `java.util.Properties`. Setting it from a string value in a service file requires a custom property editor for properties objects already be loaded. JBoss provides a suitable property editor.
- **EditorSearchPath:** This attribute allows one to set the editor packages search path on the `PropertyEditorManager` editor packages search path. Since there can be only one search path, setting this value overrides the default search path established by JBoss. If you set this, make sure to add the JBoss search path, `org.jboss.util.propertyeditor` and `org.jboss.mx.util.propertyeditor`, to the front of the new search path.

12.5. SERVICES BINDING MANAGEMENT

With all of the independently deployed services available in JBoss, running multiple instances on a given machine can be a tedious exercise in configuration file editing to resolve port conflicts. The binding service allows you centrally configure the ports for multiple JBoss instances. After the service is normally loaded by JBoss, the `ServiceConfigurator` queries the service binding manager to apply any overrides that may exist for the service. The service binding manager is configured in `conf/jboss-service.xml`. The set of configurable attributes it supports include:

- **ServerName:** This is the name of the server configuration this JBoss instance is associated with. The binding manager will apply the overrides defined for the named configuration.
- **StoreFactoryClassName:** This is the name of the class that implements the `ServicesStoreFactory` interface. You may provide your own implementation, or use the default XML based store `org.jboss.services.binding.XMLServicesStoreFactory`. The factory provides a `ServicesStore` instance responsible for providing the names configuration sets.
- **StoreURL:** This is the URL of the configuration store contents, which is passed to the `ServicesStore` instance to load the server configuration sets from. For the XML store, this is a simple service binding file.

The following is a sample service binding manager configuration that uses the `ports-01` configuration from the `sample-bindings.xml` file provided in the JBoss examples directory.

```
<mbean code="org.jboss.services.binding.ServiceBindingManager"
```

```

    name="jboss.system:service=ServiceBindingManager">
    <attribute name="ServerName">ports-01</attribute>
    <attribute name="StoreURL">
        ../docs/examples/binding-manager/sample-bindings.xml
    </attribute>
    <attribute name="StoreFactoryClassName">
        org.jboss.services.binding.XMLServicesStoreFactory
    </attribute>
</mbean>

```

The structure of the binding file is shown in [Figure 12.1, “The binding service file structure”](#).

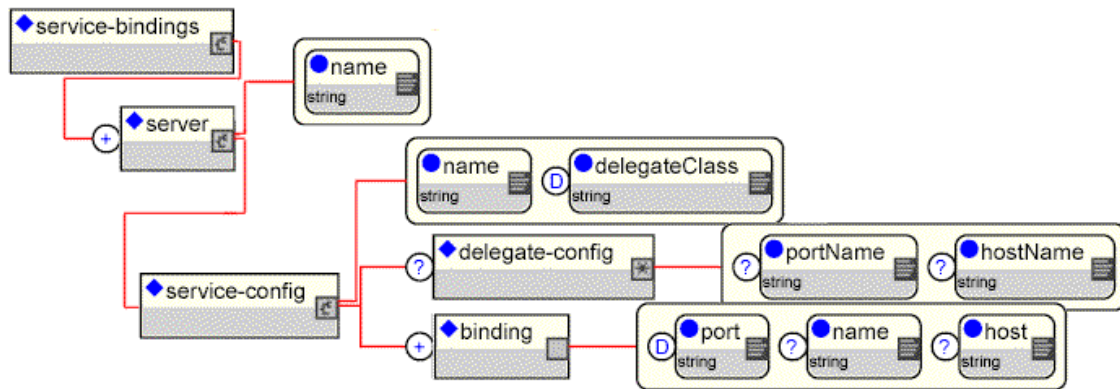


Figure 12.1. The binding service file structure

The elements are:

- service-bindings**: The root element of the configuration file. It contains one or more server elements.
- server**: This is the base of a JBoss server instance configuration. It has a required **name** attribute that defines the JBoss instance name to which it applies. This is the name that correlates with the **ServiceBindingManagerServerName** attribute value. The server element content consists of one or more **service-config** elements.
- service-config**: This element represents a configuration override for an MBean service. It has a required **name** attribute that is the JMX **ObjectName** string of the MBean service the configuration applies to. It also has a required **delegateClass** name attribute that specifies the class name of the **ServicesConfigDelegate** implementation that knows how to handle bindings for the target service. Its contents consists of an optional **delegate-config** element and one or more binding elements.
- binding**: A **binding** element specifies a named port and address pair. It has an optional **name** that can be used to provide multiple binding for a service. An example would be multiple virtual hosts for a web container. The port and address are specified via the optional **port** and **host** attributes respectively. If the port is not specified it defaults to 0 meaning choose an anonymous port. If the host is not specified it defaults to null meaning any address.
- delegate-config**: The **delegate-config** element is an arbitrary XML fragment for use by the **ServicesConfigDelegate** implementation. The **hostName** and **portName** attributes only apply to the **AttributeMappingDelegate** of the example and are there to prevent DTD aware editors from complaining about their existence in the **AttributeMappingDelegate** configurations. Generally both the attributes and content of the **delegate-config** are arbitrary, but there is no way to specify and a element can have any number of attributes with a DTD.

The three **ServicesConfigDelegate** implementations are **AttributeMappingDelegate**, **XSLTConfigDelegate**, and **XSLTFileDelegate**.

12.5.1. AttributeMappingDelegate

The **AttributeMappingDelegate** class is an implementation of the **ServicesConfigDelegate** that expects a **delegate-config** element of the form:

```
<delegate-config portName="portAttrName" hostName="hostAttrName">
  <attribute name="someAttrName">someHostPortExpr</attribute>
  <!-- ... -->
</delegate-config>
```

The **portAttrName** is the attribute name of the MBean service to which the binding port value should be applied, and the **hostAttrName** is the attribute name of the MBean service to which the binding host value should be applied. If the **portName** attribute is not specified then the binding port is not applied. Likewise, if the **hostName** attribute is not specified then the binding host is not applied. The optional attribute element(s) specify arbitrary MBean attribute names whose values are a function of the host and/or port settings. Any reference to **\${host}** in the attribute content is replaced with the host binding and any **\${port}** reference is replaced with the port binding. The **portName**, **hostName** attribute values and attribute element content may reference system properties using the **\${x}** syntax that is supported by the JBoss services descriptor.

The sample listing illustrates the usage of **AttributeMappingDelegate**.

```
<service-config name="jboss:service=Naming"
  delegateClass="org.jboss.services.binding.AttributeMappingDelegate">
  <delegate-config portName="Port"/>
  <binding port="1099" />
</service-config>
```

Here the **jboss:service=Naming** MBean service has its **Port** attribute value overridden to 1099. The corresponding setting from the **jboss1** server configuration overrides the port to 1199.

12.5.2. XSLTConfigDelegate

The **XSLTConfigDelegate** class is an implementation of the **ServicesConfigDelegate** that expects a **delegate-config** element of the form:

```
<delegate-config>
  <xslt-config configName="ConfigurationElement"><![CDATA[
    Any XSL document contents...
  ]]>
</xslt-config>
<xslt-param name="param-name">param-value</xslt-param>
<!-- ... -->
</delegate-config>
```

The **xslt-config** child element content specifies an arbitrary XSL script fragment that is to be applied to the MBean service attribute named by the **configName** attribute. The named attribute must be of type **org.w3c.dom.Element**. The optional **xslt-param** elements specify XSL script parameter

values for parameters used in the script. There are two XSL parameters defined by default called **host** and **port**, and their values are set to the configuration host and port bindings.

The **XSLTConfigDelegate** is used to transform services whose **port/interface** configuration is specified using a nested XML fragment. The following example maps the port number on hypersonic datasource:

```
<service-config
name="jboss.jca:service=ManagedConnectionFactory,name=DefaultDS"

delegateClass="org.jboss.services.binding.XSLTConfigDelegate">
  <delegate-config>
    <xslt-config configName="ManagedConnectionFactoryProperties"><![
CDATA[
<xsl:stylesheet
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>

  <xsl:output method="xml" />
  <xsl:param name="host"/>
  <xsl:param name="port"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="config-property[@name='ConnectionURL']">
    <config-property type="java.lang.String" name="ConnectionURL">
      jdbc:hsqldb:hsqldb://<xsl:value-of select='$host'/>:<xsl:value-of
select='$port'/>
    </config-property>
  </xsl:template>

  <xsl:template match="*|@*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
]]>
    </xslt-config>
  </delegate-config>
  <binding host="localhost" port="1901"/>
</service-config>
```

12.5.3. XSLTFileDelegate

The **XSLTFileDelegate** class works similarly to the **XSLTConfigDelegate** except that instead of transforming an embedded XML fragment, the XSLT script transforms a file read in from the file system. The **delegate-config** takes exactly the same form:

```
<delegate-config>
  <xslt-config configName="ConfigurationElement"><![CDATA[
    Any XSL document contents...
  ]]>
```

```

    </xslt-config>
    <xslt-param name="param-name">param-value</xslt-param>
    <!-- ... -->
</delegate-config>

```

The **xslt-config** child element content specifies an arbitrary XSL script fragment that is to be applied to the MBean service attribute named by the **configName** attribute. The named attribute must be a String value corresponding to an XML file that will be transformed. The optional **xslt-param** elements specify XSL script parameter values for parameters used in the script. There are two XSL parameters defined by default called **host** and **port**, and their values are set to the configuration **host** and **port** bindings.

The following example maps the host and port values for the Tomcat connectors:

```

<service-config name="jboss.web:service=WebServer"
delegateClass="org.jboss.services.binding.XSLTFileDelegate">
  <delegate-config>
    <xslt-config configName="ConfigFile"><![CDATA[
<xsl:stylesheet
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>

  <xsl:output method="xml" />
  <xsl:param name="port"/>

  <xsl:variable name="portAJP" select="$port - 71"/>
  <xsl:variable name="portHttps" select="$port + 363"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match = "Connector">
    <Connector>
      <xsl:for-each select="@*">
        <xsl:choose>
          <xsl:when test="(name() = 'port' and . = '8080')">
            <xsl:attribute name="port">
              <xsl:value-of select="$port" />
            </xsl:attribute>
          </xsl:when>
          <xsl:when test="(name() = 'port' and . = '8009')">
            <xsl:attribute name="port">
              <xsl:value-of select="$portAJP" />
            </xsl:attribute>
          </xsl:when>
          <xsl:when test="(name() = 'redirectPort')">
            <xsl:attribute name="redirectPort">
              <xsl:value-of select="$portHttps" />
            </xsl:attribute>
          </xsl:when>
          <xsl:when test="(name() = 'port' and . = '8443')">
            <xsl:attribute name="port">
              <xsl:value-of select="$portHttps" />
            </xsl:attribute>
          </xsl:when>
        </xsl:choose>
      </xsl:for-each>
    </Connector>
  </xsl:template>

```

```

        </xsl:when>
        <xsl:otherwise>
            <xsl:attribute name="{name()}"><xsl:value-of select="."
/></xsl:attribute>
        </xsl:otherwise>
    </xsl:choose>
</xsl:for-each>
<xsl:apply-templates/>
</Connector>
</xsl:template>

<xsl:template match="*|@">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
</xsl:template>
</xsl:stylesheet>
]]>
    </xslt-config>
</delegate-config>
<binding port="8280"/>
</service-config>

```

12.5.4. The Sample Bindings File

JBoss ships with service binding configuration file for starting up to three separate JBoss instances on one host. Here we will walk through the steps to bring up the two instances and look at the sample configuration. Start by making two server configuration file sets called **jboss0** and **jboss1** by running the following command from the book examples directory:

```
[examples]$ ant -Dchap=misc -Dex=1 run-example
```

This creates duplicates of the **server/default** configuration file sets as **server/jboss0** and **server/jboss1**, and then replaces the **conf/jboss-service.xml** descriptor with one that has the **ServiceBindingManager** configuration enabled as follows:

```

<mbean code="org.jboss.services.binding.ServiceBindingManager"
    name="jboss.system:service=ServiceBindingManager">
    <attribute name="ServerName">${jboss.server.name}</attribute>
    <attribute name="StoreURL">${jboss.server.base.dir}/misc-ex1-
bindings.xml</attribute>
    <attribute name="StoreFactoryClassName">
        org.jboss.services.binding.XMLServicesStoreFactory
    </attribute>
</mbean>

```

Here the configuration name is **\${jboss.server.name}**. JBoss will replace that with name of the actual JBoss server configuration that we pass to the run script with the **-c** option. That will be either **jboss0** or **jboss1**, depending on which configuration is being run. The binding manager will find the corresponding server configuration section from the **misc-ex1-bindings.xml** and apply the configured overrides. The **jboss0** configuration uses the default settings for the ports, while the **jboss1** configuration adds 100 to each port number.

To test the sample configuration, start two JBoss instances using the **jboss0** and **jboss1** configuration

file sets created previously. You can observe that the port numbers in the console log are different for the **jboss1** server. To test out that both instances work correctly, try accessing the web server of the first JBoss on port 8080 and then try the second JBoss instance on port 8180.

12.6. RMI DYNAMIC CLASS LOADING

The **WebService** MBean provides dynamic class loading for RMI access to the server EJBs. The configurable attributes for the service 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.
- **DownloadResources**: A flag indicating whether the server should attempt to download non-class file resources using the thread context class loader. Note that allowing this is generally a security risk as it allows access to server configuration files which may contain security settings.
- **ThreadPool**: The **org.jboss.util.threadpool.BasicThreadPoolMBean** instance thread pool used for the class loading.

12.7. SCHEDULING TASKS

Java includes a simple timer based capability through the **java.util.Timer** and **java.util.TimerTask** utility classes. JMX also includes a mechanism for scheduling JMX notifications at a given time with an optional repeat interval as the **javax.management.timer.TimerMBean** agent service.

JBoss includes two variations of the JMX timer service in the **org.jboss.varia.scheduler.Scheduler** and **org.jboss.varia.scheduler.ScheduleManager** MBeans. Both MBeans rely on the JMX timer service for the basic scheduling. They extend the behavior of the timer service as described in the following sections.

12.7.1. org.jboss.varia.scheduler.Scheduler

The **Scheduler** differs from the **TimerMBean** in that the **Scheduler** directly invokes a callback on an instance of a user defined class, or an operation of a user specified MBean.

- **InitialStartDate**: Date when the initial call is scheduled. It can be either:
 - **NOW**: date will be the current time plus 1 seconds

- A number representing the milliseconds since 1/1/1970
- Date as String able to be parsed by **SimpleDateFormat** with default format pattern **"M/d/yy h:mm a"**. If the date is in the past the **Scheduler** will search a start date in the future with respect to the initial repetitions and the period between calls. This means that when you restart the MBean (restarting JBoss etc.) it will start at the next scheduled time. When no start date is available in the future the **Scheduler** will not start.

For example, if you start your **Schedulable** everyday at Noon and you restart your JBoss server then it will start at the next Noon (the same if started before Noon or the next day if start after Noon).

- **InitialRepetitions**: The number of times the scheduler will invoke the target's callback. If -1 then the callback will be repeated until the server is stopped.
- **StartAtStartup**: A flag that determines if the **Scheduler** will start when it receives its **startService** life cycle notification. If true the **Scheduler** starts on its startup. If false, an explicit **startSchedule** operation must be invoked on the **Scheduler** to begin.
- **SchedulePeriod**: The interval between scheduled calls in milliseconds. This value must be bigger than 0.
- **SchedulableClass**: The fully qualified class name of the **org.jboss.varia.scheduler.Schedulable** interface implementation that is to be used by the **Scheduler**. The **SchedulableArguments** and **SchedulableArgumentTypes** must be populated to correspond to the constructor of the **Schedulable** implementation.
- **SchedulableArguments**: A comma separated list of arguments for the **Schedulable** implementation class constructor. Only primitive data types, **String** and classes with a constructor that accepts a **String** as its sole argument are supported.
- **SchedulableArgumentTypes**: A comma separated list of argument types for the **Schedulable** implementation class constructor. This will be used to find the correct constructor via reflection. Only primitive data types, **String** and classes with a constructor that accepts a **String** as its sole argument are supported.
- **SchedulableMBean**: Specifies the fully qualified JMX **ObjectName** name of the schedulable MBean to be called. If the MBean is not available it will not be called but the remaining repetitions will be decremented. When using **SchedulableMBean** the **SchedulableMBeanMethod** must also be specified.
- **SchedulableMBeanMethod**: Specifies the operation name to be called on the schedulable MBean. It can optionally be followed by an opening bracket, a comma separated list of parameter keywords, and a closing bracket. The supported parameter keywords include:
 - **NOTIFICATION** which will be replaced by the timers notification instance (javax.management.Notification)
 - **DATE** which will be replaced by the date of the notification call (java.util.Date)
 - **REPETITIONS** which will be replaced by the number of remaining repetitions (long)
 - **SCHEDULER_NAME** which will be replaced by the **ObjectName** of the **Scheduler**
 - Any fully qualified class name which the **Scheduler** will set to null.

A given Scheduler instance only support a single schedulable instance. If you need to configure multiple scheduled events you would use multiple **Scheduler** instances, each with a unique **ObjectName**. The following is an example of configuring a **Scheduler** to call a **Schedulable** implementation as well as a configuration for calling a MBean.

```
<server>

    <mbean code="org.jboss.varia.scheduler.Scheduler"
          name="jboss.docs:service=Scheduler">
        <attribute name="StartAtStartup">true</attribute>
        <attribute
name="SchedulableClass">org.jboss.book.misc.ex2.ExSchedulable</attribute>
        <attribute
name="SchedulableArguments">TheName,123456789</attribute>
        <attribute
name="SchedulableArgumentTypes">java.lang.String,long</attribute>

        <attribute name="InitialStartDate">NOW</attribute>
        <attribute name="SchedulePeriod">60000</attribute>
        <attribute name="InitialRepetitions">-1</attribute>
    </mbean>

</server>
```

The **SchedulableClass** `org.jboss.book.misc.ex2.ExSchedulable` example class is given below.

```
package org.jboss.book.misc.ex2;

import java.util.Date;
import org.jboss.varia.scheduler.Schedulable;

import org.apache.log4j.Logger;

/**
 * A simple Schedulable example.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ExSchedulable implements Schedulable
{
    private static final Logger log =
        Logger.getLogger(ExSchedulable.class);

    private String name;
    private long value;

    public ExSchedulable(String name, long value)
    {
        this.name = name;
        this.value = value;
        log.info("ctor, name: " + name + ", value: " + value);
    }

    public void perform(Date now, long remainingRepetitions)
```

```
{
    log.info("perform, now: " + now +
            ", remainingRepetitions: " + remainingRepetitions +
            ", name: " + name + ", value: " + value);
}
```

Deploy the timer SAR by running:

```
[examples]$ ant -Dchap=misc -Dex=2 run-example
```

The server console shows the following which includes the first two timer invocations, separated by 60 seconds:

```
21:09:27,716 INFO [ExSchedulable] ctor, name: TheName, value: 123456789
21:09:28,925 INFO [ExSchedulable] perform, now: Mon Dec 20 21:09:28 CST
2004,
    remainingRepetitions: -1, name: TheName, value: 123456789
21:10:28,899 INFO [ExSchedulable] perform, now: Mon Dec 20 21:10:28 CST
2004,
    remainingRepetitions: -1, name: TheName, value: 123456789
21:11:28,897 INFO [ExSchedulable] perform, now: Mon Dec 20 21:11:28 CST
2004,
    remainingRepetitions: -1, name: TheName, value: 123456789
```

12.8. THE TIMER SERVICE

The JMX standard defines a timer MBean (**javax.management.timer.Timer**) which can send notifications at predetermined times. The a timer MBean can be instantiated within JBoss as any other MBean.

```
<mbean code="javax.management.timer.Timer"
name="jboss.monitor:name=Heartbeat,type=Timer"/>
```

A standard JMX timer doesn't produce any timer events unless it is asked to. To aid in the configuration of the timer MBean, JBoss provides a complementary **TimerService** MBean. It interacts with the timer MBean to configure timer events at regular intervals and to transform them into JMX notifications more suitable for other services. The **TimerService** MBean takes the following attributes:

- **NotificationType**: This is the type of the notification to be generated.
- **NotificationMessage**: This is the message that should be associated with the generated notification.
- **TimerPeriod**: This is the time period between notification. The time period is in milliseconds, unless otherwise specified with a unit like "30min" or "4h". Valid time suffixes are **msec**, **sec**, **min** and **h**.
- **Repetitions**: This is the number of times the alert should be generated. A value of 0 indicates the alert should repeat indefinitely.
- **TimerMbean**: This is the **ObjectName** of the time MBean that this **TimerService** instance should configure notifications for.

The following sample illustrates the use of the **TimerService** MBean.

```
<mbean code="org.jboss.monitor.services.TimerService"
      name="jboss.monitor:name=Heartbeat,type=TimerService">
  <attribute name="NotificationType">jboss.monitor.heartbeat</attribute>
  <attribute name="NotificationMessage">JBoss is alive!</attribute>
  <attribute name="TimerPeriod">60sec</attribute>
  <depends optional-attribute-name="TimerMBean">
    jboss.monitor:name=Heartbeat,type=Timer
  </depends>
</mbean>
```

This MBean configuration configures the **jboss.monitor:name=Heartbeat,type=Timer** timer to generate a **jboss.monitor.heartbeat** notification every 60 seconds. Any service that wants to receive this periodic notifications can subscribe to the notification.

As an example, JBoss provides a simple **NotificationListener** MBean that can listen for a particular notification and log a log message when an event is generated. This MBean is very useful for debugging or manually observing notifications. The following MBean definition listens for any events generated by the heartbeat timer used in the previous examples.

```
<mbean code="org.jboss.monitor.services.NotificationListener"
      name="jboss.monitor:service=NotificationListener">
  <attribute name="SubscriptionList">
    <subscription-list>
      <mbean name="jboss.monitor:name=Heartbeat,type=Timer" />
    </subscription-list>
  </attribute>
</mbean>
```

The **subscription-list** element lists which MBeans the listener should listen to. Notice that the MBean we are listening to is the name of the actual timer MBean and not the **TimerService** MBean. Because the timer might generate multiple events, configured by multiple **TimerService** instances, you may need to filter by notification type. The **filter** element can be used to create notification filters that select only the notification types desired. The following listing shows how we can limit notifications to only the **jboss.monitor.heartbeat** type the timer service configured.

```
<mbean code="org.jboss.monitor.services.NotificationListener"
      name="jboss.monitor:service=NotificationListener">
  <attribute name="SubscriptionList">
    <subscription-list>
      <mbean name="jboss.monitor:name=Heartbeat,type=Timer">
        <filter factory="NotificationFilterSupportFactory">
          <enable type="jboss.monitor.heartbeat"/>
        </filter>
      </mbean>
    </subscription-list>
  </attribute>
</mbean>
```

As an example of a slightly more interesting listener, we'll look at the **ScriptingListener**. This listener listens for particular events and then executes a specified script when events are received. The script can be written in any bean shell scripting language. The **ScriptingListener** accepts has the following parameters.

- **ScriptLanguage:** This is the language the script is written in. This should be **beanshell**, unless you have loaded libraries for another beanshell compatible language.
- **Script:** This is the text of the script to evaluate. It is good practice to enclose the script in a CDATA section to minimize conflicts between scripting language syntax and XML syntax.
- **SubscriptionList:** This is the list of MBeans that this MBean will listen to for events that will trigger the script.

The following example illustrates the use of the **ScriptingListener**. When the previously configured timer generates a heartbeat notification, the beanshell script will execute, printing the current memory values to STDOUT. (This output will be redirected to the log files) Notice that the beanshell script has a reference to the MBean server and can execute operations against other MBeans.

```
<mbean code="org.jboss.monitor.services.ScriptingListener"
      name="jboss.monitor:service=ScriptingListener">
  <attribute name="SubscriptionList">
    <subscription-list>
      <mbean name="jboss.monitor:name=Heartbeat,type=Timer"/>
    </subscription-list>
  </attribute>
  <attribute name="ScriptLanguage">beanshell</attribute>
  <attribute name="Script">
    <![CDATA[
import javax.management.ObjectName;

/* poll free memory and thread count */
ObjectName target = new ObjectName("jboss.system:type=ServerInfo");

long freeMemory = server.getAttribute(target, "FreeMemory");
long threadCount = server.getAttribute(target, "ActiveThreadCount");

log.info("freeMemory" + freeMemory + ", threadCount" + threadCount);
]]>
  </attribute>
</mbean>
```

Of course, you are not limited to these JBoss-provided notification listeners. Other services such as the barrier service (see [Section 12.9, “The BarrierController Service”](#)) receive and act on notifications that could be generated from a timer. Additionally, any MBean can be coded to listen for timer-generated notifications.

12.9. THE BARRIERCONTROLLER SERVICE

Expressing dependencies between services using the `<depends>` tag is a convenient way to make the lifecycle of one service depend on the lifecycle of another. For example, when **serviceA** depends on **serviceB** JBoss will ensure the **serviceB.create()** is called before **serviceA.create()** and **serviceB.start()** is called before **serviceA.start()**.

However, there are cases where services do not conform to the JBoss lifecycle model, i.e. they don't expose create/start/stop/destroy lifecycle methods). This is the case for **jboss.system:type=Server MBean**, which represents the JBoss server itself. No lifecycle operations are exposed so you cannot simply express a dependency like: if JBoss is fully started then start my own service.

Or, even if they do conform to the JBoss lifecycle model, the completion of a lifecycle method (e.g. the

`start` method) may not be sufficient to describe a dependency. For example the `jboss.web:service=WebServer` MBean that wraps the embedded Tomcat server in JBoss does not start the Tomcat connectors until after the server is fully started. So putting a dependency on this MBean, if we want to hit a webpage through Tomcat, will do no good.

Resolving such non-trivial dependencies is currently performed using JMX notifications. For example the `jboss.system:type=Server` MBean emits a notification of type `org.jboss.system.server.started` when it has completed startup, and a notification of type `org.jboss.system.server.stopped` when it shuts down. Similarly, `jboss.web:service=WebServer` emits a notification of type `jboss.tomcat.connectors.started` when it starts up. Services can subscribe to those notifications in order to implement more complex dependencies. This technique has been generalized with the barrier controller service.

The barrier controller is a relatively simple MBean service that extends `ListenerServiceMBeanSupport` and thus can subscribe to any notification in the system. It uses the received notifications to control the lifecycle of a dynamically created MBean called the barrier.

The barrier is instantiated, registered and brought to the create state when the barrier controller is deployed. After that, the barrier is started and stopped when matching notifications are received. Thus, other services need only depend on the barrier MBean using the usual `<depends>` tag, without having to worry about complex lifecycle issues. They will be started and stopped in tandem with the Barrier. When the barrier controller is undeployed the barrier is destroyed.

The notifications of interest are configured in the barrier controller using the `SubscriptionList` attribute. In order to identify the starting and stopping notifications we associate with each subscription a handback string object. Handback objects, if specified, are passed back along with the delivered notifications at reception time (i.e. when `handleNotification()` is called) to qualify the received notifications, so that you can identify quickly from which subscription a notification is originating (because your listener can have many active subscriptions).

So we tag the subscriptions that produce the starting/stopping notifications of interest using any handback strings, and we configure this same string to the `StartBarrierHandback` (and `StopBarrierHandback` correspondingly) attribute of the barrier controller. Thus we can have more than one notifications triggering the starting or stopping of the barrier.

The following example shows a service that depends on the Tomcat connectors. In fact, this is a very common pattern for services that want to hit a servlet inside tomcat. The service that depends on the Barrier in the example, is a simple memory monitor that creates a background thread and monitors the memory usage, emitting notifications when thresholds get crossed, but it could be anything. We've used this because it prints out to the console starting and stopping messages, so we know when the service gets activated/deactivated.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: J2EE_Additional_Services.xml,v 1.1 2007/11/09 07:30:09 vrenish
Exp $ -->

<server>
  <!--
    In this example we have the BarrierController controlling a Barrier
    that is started when we receive the "jboss.tomcat.connectors.started"
    notification from the Tomcat mbean, and stopped when we receive the
    "org.jboss.system.server.stopped" notification from the server mbean.

    The dependent services need only define a dependency on the Barrier
    mbean!
```

```

-->
<mbean code="org.jboss.system.BarrierController"
      name="jboss:service=BarrierController">

    <!-- Whether to have the Barrier initially started or not -->
    <attribute name="BarrierEnabledOnStartup">false</attribute>

    <!-- Whether to subscribe for notifications after startup -->
    <attribute name="DynamicSubscriptions">true</attribute>

    <!-- Dependent services will depend on this mbean -->
    <attribute
name="BarrierObjectName">jboss:name=TomcatConnector,type=Barrier</attribut
e>

    <!-- The notification subscription handback that starts the barrier --
>
    <attribute name="StartBarrierHandback">start</attribute>

    <!-- The notification subscription handback that stops the barrier -->
    <attribute name="StopBarrierHandback">stop</attribute>

    <!-- The notifications to subscribe for, along with their handbacks --
>
    <attribute name="SubscriptionList">
        <subscription-list>
            <mbean name="jboss.web:service=WebServer" handback="start">
                <filter factory="NotificationFilterSupportFactory">
                    <enable type="jboss.tomcat.connectors.started"/>
                </filter>
            </mbean>
            <mbean name="jboss.system:type=Server" handback="stop">
                <filter factory="NotificationFilterSupportFactory">
                    <enable type="org.jboss.system.server.stopped"/>
                </filter>
            </mbean>
        </subscription-list>
    </attribute>
</mbean>

<!--
    An example service that depends on the Barrier we declared above.
    This services creates a background thread and monitors the memory
    usage. When it exceeds the defined thresholds it emits notifications
-->
<mbean code="org.jboss.monitor.services.MemoryMonitor"
      name="jboss.monitor:service=MemoryMonitor">

    <attribute name="FreeMemoryWarningThreshold">20m</attribute>
    <attribute name="FreeMemoryCriticalThreshold">15m</attribute>

    <!-- The BarrierObjectName configured in the BarrierController -->
    <depends>jboss:name=TomcatConnector,type=Barrier</depends>
</mbean>

</server>

```

If you hot-deploy this on a running server the Barrier will be stopped because by the time the barrier controller is deployed the starting notification is already seen. (There are ways to overcome this.) However, if you re-start the server, the barrier will be started just after the Tomcat connectors get activated. You can also manually start or stop the barrier by using the **startBarrier()** and **stopBarrier()** operations on the barrier controller. The attribute **BarrierStateString** indicates the status of the barrier.

12.10. EXPOSING MBEAN EVENTS VIA SNMP

JBoss has an SNMP adaptor service that can be used to intercept JMX notifications emitted by MBeans, convert them to traps and send them to SNMP managers. In this respect the snmp-adaptor acts as a SNMP agent. Future versions may offer support for full agent get/set functionality that maps onto MBean attributes or operations.

This service can be used to integrate JBoss with higher order system/network management platforms (HP OpenView, for example), making the MBeans visible to those systems. The MBean developer can instrument the MBeans by producing notifications for any significant event (e.g. server coldstart), and adaptor can then be configured to intercept the notification and map it onto an SNMP traps. The adaptor uses the JoeSNMP package from OpenNMS as the SNMP engine.

The SNMP service is configured in **snmp-adaptor.sar**. This service is only available in the **all** configuration, so you'll need to copy it to your configuration if you want to use it. Inside the **snmp-adaptor.sar** directory, there are two configuration files that control the SNMP service.

- **managers.xml**: configures where to send traps. The content model for this file is shown in [Figure 12.2, “The schema for the SNMP managers file”](#).
- **notifications.xml**: specifies the exact mapping of each notification type to a corresponding SNMP trap. The content model for this file is shown in [Figure 12.3, “The schema for the notification to trap mapping file”](#).

The **SNMPAgentService** MBean is configured in **snmp-adaptor.sar/META-INF/jboss-service.xml**. The configurable parameters are:

- **HeartBeatPeriod**: The period in seconds at which heartbeat notifications are generated.
- **ManagersResName**: Specifies the resource name of the **managers.xml** file.
- **NotificationMapResName**: Specifies the resource name of the **notifications.xml** file.
- **TrapFactoryClassName**: The **org.jboss.jmx.adaptor.snmp.agent.TrapFactory** implementation class that takes care of translation of JMX Notifications into SNMP V1 and V2 traps.
- **TimerName**: Specifies the JMX ObjectName of the JMX timer service to use for heartbeat notifications.
- **SubscriptionList**: Specifies which MBeans and notifications to listen for.

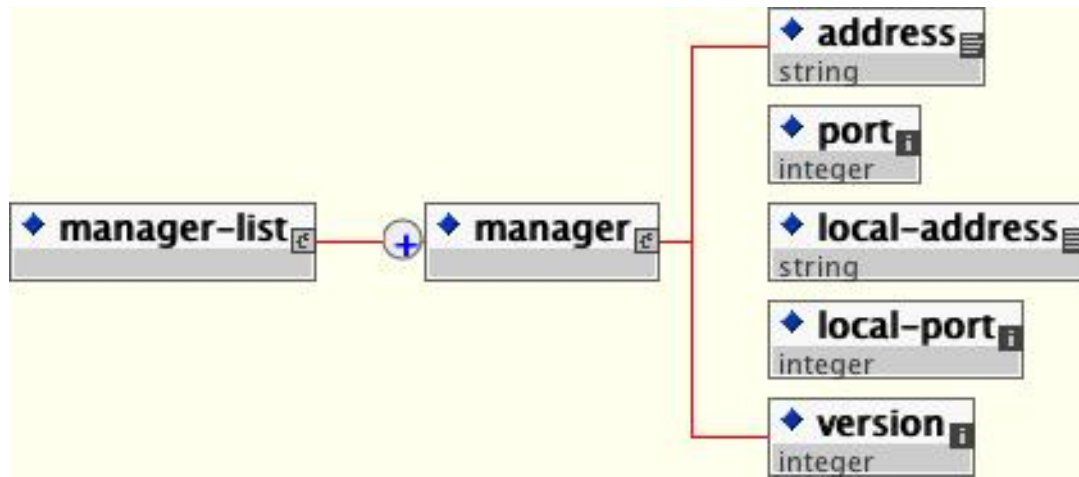


Figure 12.2. The schema for the SNMP managers file

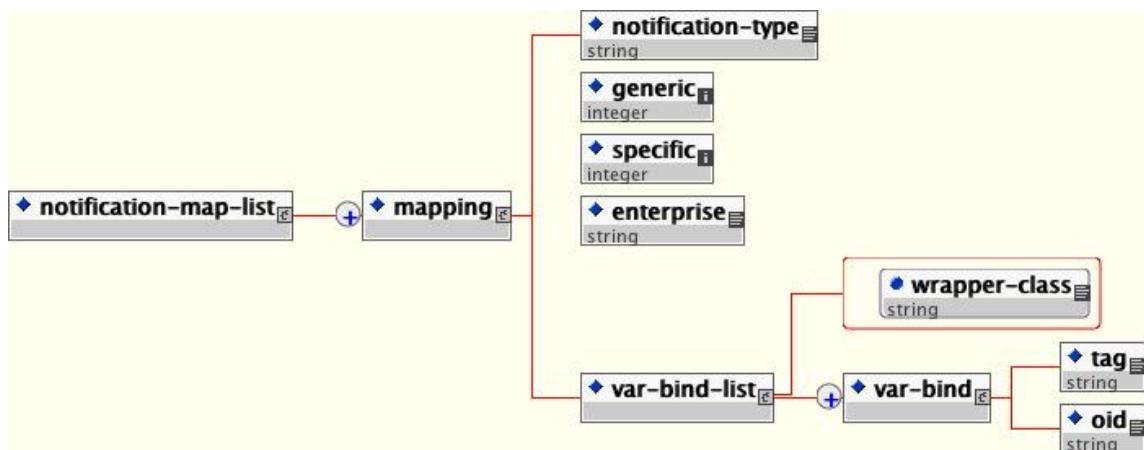


Figure 12.3. The schema for the notification to trap mapping file

TrapdService is a simple MBean that acts as an SNMP Manager. It listens to a configurable port for incoming traps and logs them as DEBUG messages using the system logger. You can modify the log4j configuration to redirect the log output to a file. **SnmpAgentService** and **TrapdService** are not dependent on each other.

PART III. CLUSTERING CONFIGURATION

CHAPTER 13. CLUSTERING

High Availability Enterprise Services via JBoss Clusters

13.1. INTRODUCTION

Clustering allows us to run an application on several parallel servers (a.k.a cluster nodes) while providing a single view to application clients. Load is distributed across different servers, and even if one or more of the servers fails, the application is still accessible via the surviving cluster nodes. Clustering is crucial for scalable enterprise applications, as you can improve performance by simply adding more nodes to the cluster. Clustering is crucial for highly available enterprise applications, as it is the clustering infrastructure that supports the redundancy needed for high availability.

The JBoss Application Server (AS) comes with clustering support out of the box. The simplest way to start a JBoss server cluster is to start several JBoss instances on the same local network, using the **run -c all** command for each instance. Those server instances, all started in the **all** configuration, detect each other and automatically form a cluster.

In the first section of this chapter, we discuss basic concepts behind JBoss's clustering services. It is important that you understand these concepts before reading the rest of the chapter. Clustering configurations for specific types of applications are covered after this section.

13.2. CLUSTER DEFINITION

A cluster is a set of nodes that communicate with each other and work toward a common goal. In a JBoss Application Server cluster (also known as a “partition”), a node is an JBoss Application Server instance. Communication between the nodes is handled by the JGroups group communication library, with a JGroups Channel providing the core functionality of tracking who is in the cluster and reliably exchanging messages between the cluster members. JGroups channels with the same configuration and name have the ability to dynamically discover each other and form a group. This is why simply executing “run -c all” on two AS instances on the same network is enough for them to form a cluster – each AS starts a Channel (actually, several) with the same default configuration, so they dynamically discover each other and form a cluster. Nodes can be dynamically added to or removed from clusters at any time, simply by starting or stopping a Channel with a configuration and name that matches the other cluster members. In summary, a JBoss cluster is a set of AS server instances each of which is running an identically configured and named JGroups Channel.

On the same AS instance, different services can create their own Channel. In a default 4.2.x AS, four different services create channels – the web session replication service, the EJB3 SFSB replication service, the EJB3 entity caching service, and a core general purpose clustering service known as HAPartition. In order to differentiate these channels, each must have a unique name, and its configuration must match its peers yet differ from the other channels.

So, if you go to two AS 4.2.x instances and execute **run -c all**, the channels will discover each other and you'll have a conceptual **cluster**. It's easy to think of this as a two node cluster, but it's important to understand that you really have 4 channels, and hence 4 two node clusters.

On the same network, even for the same service, we may have different clusters. [Figure 13.1, “Clusters and server nodes”](#) shows an example network of JBoss server instances divided into three clusters, with the third cluster only having one node. This sort of topology can be set up simply by configuring the AS instances such that within a set of nodes meant to form a cluster the Channel configurations and names match while they differ from any other channels on the same network.

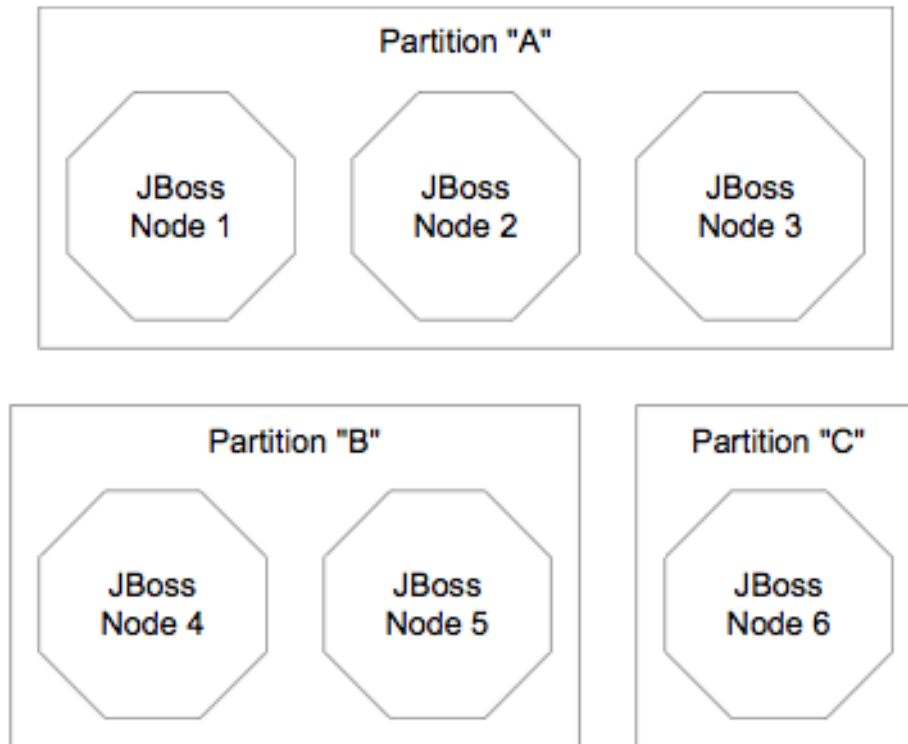


Figure 13.1. Clusters and server nodes

The section on “JGroups Configuration” and on “Isolating JGroups Channels” covers in detail how to configure Channels such that desired peers find each other and unwanted peers do not. As mentioned above, by default JBoss AS uses four separate JGroups Channels. These can be divided into two broad categories: the Channel used by the general purpose HAPartition service, and three Channels created by JBoss Cache for special purpose caching and cluster wide state replication.

13.3. SEPARATING CLUSTERS

As discussed in [Section 4.2, “Standard Server Configurations”](#), the production configuration is based on the all configuration and supports clustering as a result. Server instances that run on the same network, which are started using the default configuration settings, will belong to the same cluster. As such, it is important to understand how to separate clusters during server startup. This will be required, for instance, to separate test clusters from production clusters.

The attributes and settings discussed below will be covered in detail in the sections that follow. They are introduced here to emphasize the importance of separating clusters during server startup. The relevant MBeans and their associated properties (PartitionName and mcast_addr) *both* require modification to successfully separate clusters.

The HAPartition MBean

Starting JBoss servers with their default clustering settings on a local network results in a default cluster named DefaultPartition that includes all server instances as its nodes. This setting is defined in the HAPartition MBean definition packaged with the standard JBoss AS distribution as in the following code sample:

```

<mbean code="org.jboss.ha.framework.server.ClusterPartition"
  name="jboss:service=DefaultPartition">

  <!-- Name of the partition being built -->
  <attribute name="PartitionName">
    ${jboss.partition.name:DefaultPartition}
  </attribute>
</mbean>
  
```

```

    </attribute>

    ...
    ...

</mbean>

```

PartitionName is an optional attribute to specify the name of the cluster. Its default value is DefaultPartition as indicated in the code sample above. It is necessary to use the -g (a.k.a. --partition) command line switch to set this value at JBoss startup and create unique cluster (group) names.

```
/run.sh -g QAPartition -b 192.168.1.100 -c all
```

As will be discussed in [Section 13.4, “HAPartition”](#) to follow, the nodes that form a cluster must have exactly the same PartitionName and PartitionConfig elements. PartitionConfig elements are defined in the Cluster Partition MBean which will be discussed briefly here.

The Cluster Partition MBean

Section [Section 19.1, “JGroups Configuration”](#) discusses how the JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster.

JGroups configurations often appear as a nested attribute in cluster related MBean services, such as the PartitionConfig attribute in the ClusterPartition MBean or the ClusterConfig attribute in the TreeCache MBean. The behavior and properties of each protocol in JGroups can be configured via the MBean attributes. Below is an extract of a JGroups configuration in the ClusterPartition MBean showing the UDP protocol configuration:

```

<mbean code="org.jboss.ha.framework.server.ClusterPartition"
  name="jboss:service=${jboss.partition.name:DefaultPartition}">

  ... ..

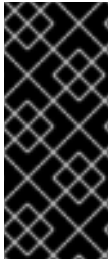
  <attribute name="PartitionConfig">
    <Config>
      <UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
        mcast_port="${jboss.hapartition.mcast_port:45566}"
        tos="8"
        ucast_rcv_buf_size="20000000"
        ucast_send_buf_size="640000"
        mcast_rcv_buf_size="25000000"
        mcast_send_buf_size="640000"
        loopback="false"
          discard_incompatible_packets="true"
          enable_bundling="false"
          max_bundle_size="64000"
          max_bundle_timeout="30"
          use_incoming_packet_handler="true"
          use_outgoing_packet_handler="false"
          ip_ttl="${jgroups.udp.ip_ttl:2}"
          down_thread="false" up_thread="false"/>
        ...
        ...
      </Config>
    </attribute>
  </mbean>

```

The `mcast_addr` above specifies the multicast address (class D) for joining a group (i.e., the cluster). If omitted, the default is 228.8.8.8. The `-u` (a.k.a. `--udp`) command line switch may be used to control the multicast address used by the JGroups channels opened by all standard AS services.

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the `jboss.partition.udpGroup` system property as indicated in the above `ClusterPartition` MBean code sample.



IMPORTANT

The production server configuration is based on the all configuration and supports clustering as a result. Any server instance started using the default production configuration, that is located on the same network, will form part of single default cluster. Test clusters may inadvertently from a part of a production cluster unless the `-u` and `-g` flags are set during server startup as described above.

13.4. HAPARTITION

HAPartition is a general purpose service used for a variety of tasks in AS clustering. At its core, it is an abstraction built on top of a JGroups Channel that provides support for making/receiving RPC invocations on/from one or more cluster members. HAPartition also supports a distributed registry of which clustering services are running on which cluster members. It provides notifications to interested listeners when the cluster membership changes or the clustered service registry changes. HAPartition forms the core of many of the clustering services we'll be discussing in the rest of this guide, including smart client-side clustered proxies, EJB 2 SFSB replication and entity cache management, farming, HA-JNDI and HA singletons.

The following example shows the **HAPartition** MBean definition packaged with the standard JBoss AS distribution. So, if you simply start JBoss servers with their default clustering settings on a local network, you would get a default cluster named **DefaultPartition** that includes all server instances as its nodes.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
  name="jboss:service=DefaultPartition">

  <!-- Name of the partition being built -->
  <attribute name="PartitionName">
    ${jboss.partition.name:DefaultPartition}
  </attribute>

  <!-- The address used to determine the node name -->
  <attribute name="NodeAddress">${jboss.bind.address}</attribute>

  <!-- Determine if deadlock detection is enabled -->
  <attribute name="DeadlockDetection">False</attribute>

  <!-- Max time (in ms) to wait for state transfer to complete.
  Increase for large states -->
  <attribute name="StateTransferTimeout">30000</attribute>

  <!-- The JGroups protocol configuration -->
  <attribute name="PartitionConfig">
```

```

        ... ..
    </attribute>
</mbean>

```

Here, we omitted the detailed JGroups protocol configuration for this channel. JGroups handles the underlying peer-to-peer communication between nodes, and its configuration is discussed in [Section 19.1, “JGroups Configuration”](#). The following list shows the available configuration attributes in the **HAPartition** MBean.

- **PartitionName** is an optional attribute to specify the name of the cluster. Its default value is **DefaultPartition**. Use the **-g** (a.k.a. **--partition**) command line switch to set this value at JBoss startup.
- **NodeAddress** is an optional attribute used to help generate a unique name for this node.
- **DeadlockDetection** is an optional boolean attribute that tells JGroups to run message deadlock detection algorithms with every request. Its default value is **false**.
- **StateTransferTimeout** is an optional attribute to specify the timeout for state replication across the cluster (in milliseconds). State replication refers to the process of obtaining initial application state from other already-running cluster members at service startup. Its default value is **30000**.
- **PartitionConfig** is an element to specify JGroup configuration options for this cluster (see [Section 19.1, “JGroups Configuration”](#)).

In order for nodes to form a cluster, they must have the exact same **PartitionName** and the **PartitionConfig** elements. Changes in either element on some but not all nodes would cause the cluster to split.

You can view the current cluster information by pointing your browser to the JMX console of any JBoss instance in the cluster (i.e., **http://hostname:8080/jmx-console/**) and then clicking on the **jboss:service=DefaultPartition** MBean (change the MBean name to reflect your partition name if you use the **-g** startup switch). A list of IP addresses for the current cluster members is shown in the **CurrentView** field.



NOTE

While it is technically possible to put a JBoss server instance into multiple HAPartitions at the same time, this practice is generally not recommended, as it increases management complexity.

13.5. JBOSS CACHE CHANNELS

JBoss Cache is a fully featured distributed cache framework that can be used in any application server environment or standalone. JBoss AS integrates JBoss Cache to provide cache services for HTTP sessions, EJB 3.0 session beans, and EJB 3.0 entity beans. Each of these cache services is defined in a separate Mbean, and each cache creates its own JGroups Channel. We will cover those MBeans when we discuss specific services in the next several sections.

13.5.1. Service Architectures

The clustering topography defined by the **HAPartition** MBean on each node is of great importance to system administrators. But for most application developers, you are probably more concerned about the cluster architecture from a client application's point of view. Two basic clustering architectures are used

with JBoss AS: client-side interceptors (a.k.a smart proxies or stubs) and external load balancers. Which architecture your application will use will depend on what type of client you have.

13.5.1.1. Client-side interceptor architecture

Most remote services provided by the JBoss application server, including JNDI, EJB, JMS, RMI and JBoss Remoting, require the client to obtain (e.g., to look up and download) a stub (or proxy) object. The stub object is generated by the server and it implements the business interface of the service. The client then makes local method calls against the stub object. The stub automatically routes the call across the network and where it is invoked against service objects managed in the server. In a clustering environment, the server-generated stub object includes an interceptor that understands how to route calls to multiple nodes in the cluster. The stub object figures out how to find the appropriate server node, marshal call parameters, un-marshal call results, and return the result to the caller client.

The stub interceptors maintain up-to-date knowledge about the cluster. For instance, they know the IP addresses of all available server nodes, the algorithm to distribute load across nodes (see next section), and how to failover the request if the target node not available. As part of handling each service request, if the cluster topology has changed the server node updates the stub interceptor with the latest changes in the cluster. For instance, if a node drops out of the cluster, each of client stub interceptor is updated with the new configuration the next time it connects to any active node in the cluster. All the manipulations done by the service stub are transparent to the client application. The client-side interceptor clustering architecture is illustrated in [Figure 13.2, “The client-side interceptor \(proxy\) architecture for clustering”](#).

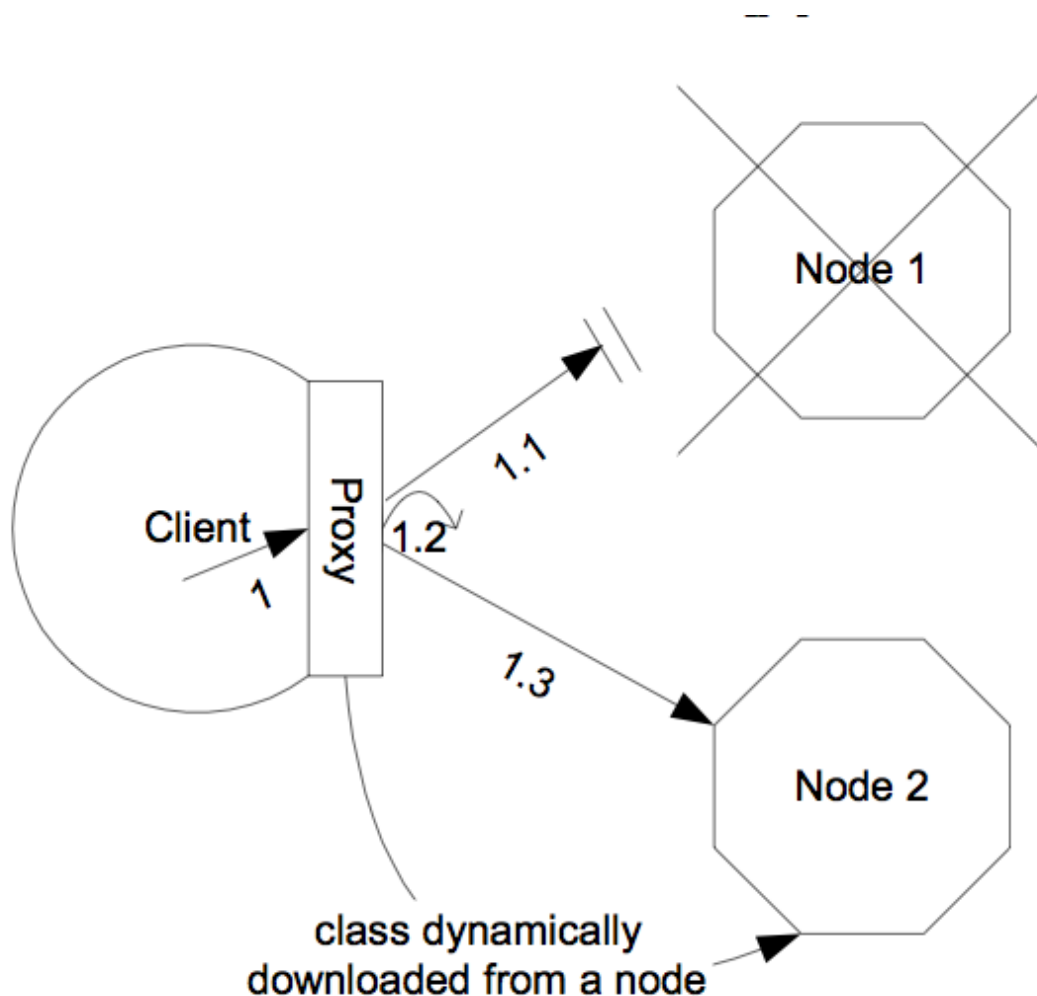


Figure 13.2. The client-side interceptor (proxy) architecture for clustering

**NOTE**

Section 15.1, “[Stateless Session Bean in EJB 2.x](#)” describes how to enable the client proxy to handle the entire cluster restart.

13.5.1.2. Load balancer

Other JBoss services, in particular the HTTP-based services, do not require the client to download anything. The client (e.g., a web browser) sends in requests and receives responses directly over the wire according to certain communication protocols (e.g., the HTTP protocol). In this case, an external load balancer is required to process all requests and dispatch them to server nodes in the cluster. The client only needs to know about how to contact the load balancer; it has no knowledge of the JBoss AS instances behind the load balancer. The load balancer is logically part of the cluster, but we refer to it as “external” because it is not running in the same process as either the client or any of the JBoss AS instances. It can be implemented either in software or hardware. There are many vendors of hardware load balancers; the `mod_jk` Apache module is an excellent example of a software load balancer. An external load balancer implements its own mechanism for understanding the cluster configuration and provides its own load balancing and failover policies. The external load balancer clustering architecture is illustrated in [Figure 13.3, “The external load balancer architecture for clustering”](#).

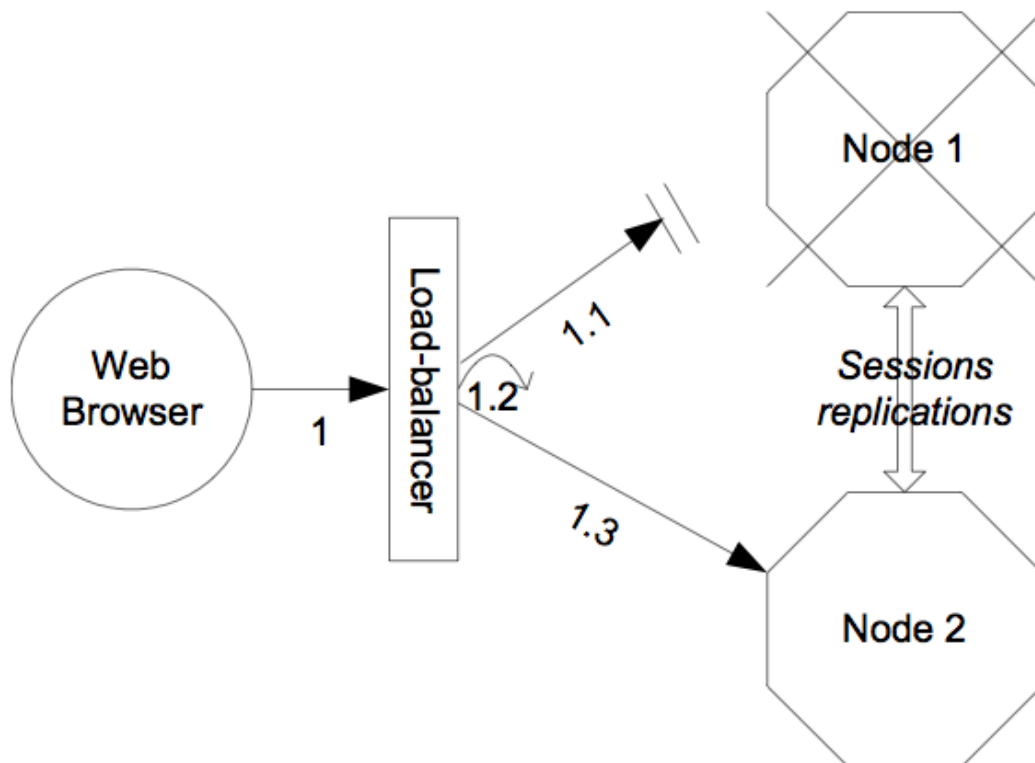


Figure 13.3. The external load balancer architecture for clustering

A potential problem with an external load balancer architecture is that the load balancer itself may be a single point of failure. It needs to be monitored closely to ensure high availability of the entire cluster's services.

13.5.2. Load-Balancing Policies

Both the JBoss client-side interceptor (stub) and load balancer use load balancing policies to determine which server node to which node a new request should be sent. In this section, let's go over the load balancing policies available in JBoss AS.

13.5.2.1. Client-side interceptor architecture

In JBoss 4.2.2, the following load balancing options are available when the client-side interceptor architecture is used. The client-side stub maintains a list of all nodes providing the target service; the job of the load balance policy is to pick a node from this list for each request.

- Round-Robin (**`org.jboss.ha.framework.interfaces.RoundRobin`**): each call is dispatched to a new node, proceeding sequentially through the list of nodes. The first target node is randomly selected from the list.
- Random-Robin (**`org.jboss.ha.framework.interfaces.RandomRobin`**): for each call the target node is randomly selected from the list.
- First Available (**`org.jboss.ha.framework.interfaces.FirstAvailable`**): one of the available target nodes is elected as the main target and is thereafter used for every call; this elected member is randomly chosen from the list of members in the cluster. When the list of target nodes changes (because a node starts or dies), the policy will choose a new target node unless the currently elected node is still available. Each client-side stub elects its own target node independently of the other stubs, so if a particular client downloads two stubs for the same target service (e.g., an EJB), each stub will independently pick its target. This is an example of a policy that provides “session affinity” or “sticky sessions”, since the target node does not change once established.
- First Available Identical All Proxies (**`org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies`**): has the same behaviour as the “First Available” policy but the elected target node is shared by all stubs in the same client-side VM that are associated with the same target service. So if a particular client downloads two stubs for the same target service (e.g. an EJB), each stub will use the same target.

Each of the above is an implementation of the `org.jboss.ha.framework.interfaces.LoadBalancePolicy` interface; users are free to write their own implementation of this simple interface if they need some special behavior. In later sections we'll see how to configure the load balance policies used by different services.

13.5.2.2. External load balancer architecture

As noted above, an external load balancer provides its own load balancing capabilities. What capabilities are supported depends on the provider of the load balancer. The only JBoss requirement is that the load balancer support “session affinity” (a.k.a. “sticky sessions”). With session affinity enabled, once the load balancer routes a request from a client to node A and the server initiates a session, all future requests associated with that session must be routed to node A, so long as node A is available.

13.5.3. Farming Deployment

The easiest way to deploy an application into the cluster is to use the farming service. That is to hot-deploy the application archive file (e.g., the EAR, WAR or SAR file) in the **`all/farm/`** directory of any of the cluster members and the application will be automatically duplicated across all nodes in the same cluster. If node joins the cluster later, it will pull in all farm deployed applications in the cluster and deploy them locally at start-up time. If you delete the application from one of the running cluster server node's **`farm/`** folder, the application will be undeployed locally and then removed from all other cluster server nodes farm folder (triggers undeployment.) You should manually delete the application from the farm folder of any server node not currently connected to the cluster.

**NOTE**

Currently, due to an implementation weakness, the farm deployment service only works for 1) archives located in the farm/ directory of the first node to join the cluster or 2) hot-deployed archives. If you first put a new application in the farm/ directory and then start the server to have it join an already running cluster, the application will not be pushed across the cluster or deployed. This is because the farm service does not know whether the application really represents a new deployment or represents an old deployment that was removed from the rest of the cluster while the newly starting node was off-line. We are working to resolve this issue.

**NOTE**

You can only put zipped archive files, not exploded directories, in the farm directory. If exploded directories are placed in farm the directory contents will be replicated around the cluster piecemeal, and it is very likely that remote nodes will begin trying to deploy things before all the pieces have arrived, leading to deployment failure.

**NOTE**

Farmed deployment is not atomic. A problem deploying, undeploying or redeploying an application on one node in the cluster will not prevent the deployment, undeployment or redeployment being done on the other nodes. There is no rollback capability. Deployment is also not staggered; it is quite likely, for example, that a redeployment will happen on all nodes in the cluster simultaneously, briefly leaving no nodes in the cluster providing service.

Farming is enabled by default in the **all** configuration in JBoss AS distributions, so you will not have to set it up yourself. The **farm-service.xml** configuration file is located in the `deploy/deploy.last` directory. If you want to enable farming in a custom configuration, simply copy the `farm-service.xml` file and copy it to the JBoss deploy directory

\$JBOSS_HOME/server/your_own_config/deploy/deploy.last. Make sure that your custom configuration has clustering enabled.

After deploying `farm-service.xml` you are ready to rumble. The required `FarmMemberService` MBean attributes for configuring a farm are listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<server>

    <mbean code="org.jboss.ha.framework.server.FarmMemberService"
          name="jboss:service=FarmMember,partition=DefaultPartition">
        ...

    <depends optional-attribute-name="ClusterPartition"
      proxy-type="attribute">
      jboss:service=${jboss.partition.name:DefaultPartition}
    </depends>
    <attribute name="ScanPeriod">5000</attribute>
    <attribute name="URLs">farm/</attribute>
    ...
  </mbean>
</server>
```


- **ClusterPartition** is a required attribute to inject the **HAPartition** service that the farm service uses for intra-cluster communication.
- **URLs** points to the directory where deployer watches for files to be deployed. This MBean will create this directory if it does not already exist. If a full URL is not provided, it is assumed that the value is a filesystem path relative to the configuration directory (e.g. **\$JBoss_HOME/server/all/**).
- **ScanPeriod** specifies the interval at which the folder must be scanned for changes.. Its default value is **5000**.

The farming service is an extension of the **URLDeploymentScanner**, which scans for hot deployments in the **deploy/** directory. So, you can use all the attributes defined in the **URLDeploymentScanner** MBean in the **FarmMemberService** MBean. In fact, the **URLs** and **ScanPeriod** attributes listed above are inherited from the **URLDeploymentScanner** MBean.

13.5.4. Distributed state replication services

In a clustered server environment, distributed state management is a key service the cluster must provide. For instance, in a stateful session bean application, the session state must be synchronized among all bean instances across all nodes, so that the client application reaches the same session state no matter which node serves the request. In an entity bean application, the bean object sometimes needs to be cached across the cluster to reduce the database load. Currently, the state replication and distributed cache services in JBoss AS are provided via three ways: the **HASessionState** Mbean, the **DistributedState** MBean and the JBoss Cache framework.

- The **HASessionState** MBean is a legacy service that provides session replication and distributed cache services for EJB 2.x stateful session beans. The MBean is defined in the **all/deploy/cluster-service.xml** file. We will show its configuration options in the EJB 2.x stateful session bean section later.
- The **DistributedState** Mbean is a legacy service built on the **HAPartition** service. It is supported for backwards compatibility reasons, but new applications should not use it; they should use the much more sophisticated JBoss Cache instead.
- As mentioned above JBoss Cache is used to provide cache services for HTTP sessions, EJB 3.0 session beans and EJB 3.0 entity beans. It is the primary distributed state management tool in JBoss AS, and is an excellent choice for any custom caching requirements your applications may have. We will cover JBoss Cache in more detail when we discuss specific services in the next several sections..

CHAPTER 14. CLUSTERED JNDI SERVICES

JNDI is one of the most important services provided by the application server. The JBoss HA-JNDI (High Availability JNDI) service brings the following features to JNDI:

- Transparent failover of naming operations. If an HA-JNDI naming Context is connected to the HA-JNDI service on a particular JBoss AS instance, and that service fails or is shut down, the HA-JNDI client can transparently fail over to another AS instance.
- Load balancing of naming operations. An HA-JNDI naming Context will automatically load balance its requests across all the HA-JNDI servers in the cluster.
- Automatic client discovery of HA-JNDI servers (using multicast).
- Unified view of JNDI trees cluster-wide. Client can connect to the HA-JNDI service running on any node in the cluster and find objects bound in JNDI on any other node. This is accomplished via two mechanisms:
- Cross-cluster lookups. A client can perform a lookup and the server side HA-JNDI service has the ability to find things bound in regular JNDI on any node in the cluster.
- A replicated cluster-wide context tree. An object bound into the HA-JNDI service will be replicated around the cluster, and a copy of that object will be available in-VM on each node in the cluster.

JNDI is a key component for many other interceptor-based clustering services: those services register themselves with the JNDI so that the client can lookup their proxies and make use of their services. HA-JNDI completes the picture by ensuring that clients have a highly-available means to look up those proxies. However, it is important to understand that using HA-JNDI (or not) has no effect whatsoever on the clustering behavior of the objects that are looked up. To illustrate:

- If an EJB is not configured as clustered, looking up the EJB via HA-JNDI does not somehow result in the addition of clustering capabilities (load balancing of EJB calls, transparent failover, state replication) to the EJB.
- If an EJB is configured as clustered, looking up the EJB via regular JNDI instead of HA-JNDI does not somehow result in the removal of the bean proxy's clustering capabilities.

14.1. HOW IT WORKS

The JBoss client-side HA-JNDI naming Context is based on the client-side interceptor architecture. The client obtains an HA-JNDI proxy object (via the InitialContext object) and invokes JNDI lookup services on the remote server through the proxy. The client specifies that it wants an HA-JNDI proxy by configuring the naming properties used by the InitialContext object. This is covered in detail in the “Client Configuration” section. Other than the need to ensure the appropriate naming properties are provided to the InitialContext, the fact that the naming Context is using HA-JNDI is completely transparent to the client.

On the server side, the HA-JNDI service maintains a cluster-wide context tree. The cluster wide tree is always available as long as there is one node left in the cluster. Each node in the cluster also maintains its own local JNDI context tree. The HA-JNDI service on that node is able to find objects bound into the local JNDI context tree. An application can bind its objects to either tree. The design rationale for this architecture is as follows:

- It avoids migration issues with applications that assume that their JNDI implementation is local. This allows clustering to work out-of-the-box with just a few tweaks of configuration files.

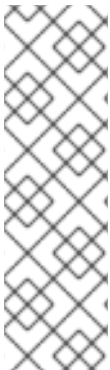
- In a homogeneous cluster, this configuration actually cuts down on the amount of network traffic. A homogenous cluster is one where the same types of objects are bound under the same names on each node.
- Designing it in this way makes the HA-JNDI service an optional service since all underlying cluster code uses a straight new **InitialContext()** to lookup or create bindings.

On the server side, a naming **Context** obtained via a call to new **InitialContext()** will be bound to the local-only, non-cluster-wide JNDI Context (this is actually basic JNDI). So, all EJB homes and such will not be bound to the cluster-wide JNDI Context, but rather, each home will be bound into the local JNDI.

When a remote client does a lookup through HA-JNDI, HA-JNDI will delegate to the local JNDI Context when it cannot find the object within the global cluster-wide Context. The detailed lookup rule is as follows.

- If the binding is available in the cluster-wide JNDI tree, return it.
- If the binding is not in the cluster-wide tree, delegate the lookup query to the local JNDI service and return the received answer if available.
- If not available, the HA-JNDI services asks all other nodes in the cluster if their local JNDI service owns such a binding and returns the answer from the set it receives.
- If no local JNDI service owns such a binding, a **NameNotFoundException** is finally raised.

In practice, objects are rarely bound in the cluster-wide JNDI tree; rather they are bound in the local JNDI tree. For example, when EJBs are deployed, their proxies are always bound in local JNDI, not HA-JNDI. So, an EJB home lookup done through HA-JNDI will always be delegated to the local JNDI instance.



NOTE

If different beans (even of the same type, but participating in different clusters) use the same JNDI name, this means that each JNDI server will have a logically different "target" bound (JNDI on node 1 will have a binding for bean A and JNDI on node 2 will have a binding, under the same name, for bean B). Consequently, if a client performs a HA-JNDI query for this name, the query will be invoked on any JNDI server of the cluster and will return the locally bound stub. Nevertheless, it may not be the correct stub that the client is expecting to receive! So, it is always best practice to ensure that across the cluster different names are used for logically different bindings.



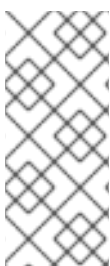
NOTE

You cannot currently use a non-JNP JNDI implementation (i.e. LDAP) for your local JNDI implementation if you want to use HA-JNDI. However, you can use JNDI federation using the ExternalContext MBean to bind non-JBoss JNDI trees into the JBoss JNDI namespace. Furthermore, nothing prevents you using one centralized JNDI server for your whole cluster and scrapping HA-JNDI and JNP.

**NOTE**

If a binding is only made available on a few nodes in the cluster (for example because a bean is only deployed on a small subset of nodes in the cluster), the probability that a lookup will hit a HA-JNDI server that does not own this binding is higher and thus the lookup will need to be forwarded to all nodes in the cluster. Consequently, the query time will be longer than if the binding would have been available locally. Moral of the story: as much as possible, cache the result of your JNDI queries in your client.

So, an EJB home lookup through HA-JNDI, will always be delegated to the local JNDI instance. If different beans (even of the same type, but participating in different clusters) use the same JNDI name, it means that each JNDI server will have a different "target" bound (JNDI on node 1 will have a binding for bean A and JNDI on node 2 will have a binding, under the same name, for bean B). Consequently, if a client performs a HA-JNDI query for this name, the query will be invoked on any JNDI server of the cluster and will return the locally bound stub. Nevertheless, it may not be the correct stub that the client is expecting to receive!

**NOTE**

You cannot currently use a non-JNP JNDI implementation (i.e. LDAP) for your local JNDI implementation if you want to use HA-JNDI. However, you can use JNDI federation using the **ExternalContext** MBean to bind non-JBoss JNDI trees into the JBoss JNDI namespace. Furthermore, nothing prevents you though of using one centralized JNDI server for your whole cluster and scrapping HA-JNDI and JNP.

**NOTE**

If a binding is only made available on a few nodes in the cluster (for example because a bean is only deployed on a small subset of nodes in the cluster), the probability to lookup a HA-JNDI server that does not own this binding is higher and the lookup will need to be forwarded to all nodes in the cluster. Consequently, the query time will be longer than if the binding would have been available locally. Moral of the story: as much as possible, cache the result of your JNDI queries in your client.

14.2. CLIENT CONFIGURATION

14.2.1. For clients running inside the application server

If you want to access HA-JNDI from inside the application server, you must explicitly get an InitialContext by passing in JNDI properties. The following code shows how to create a naming Context bound to HA-JNDI:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "localhost:1100"); // HA-JNDI port.
return new InitialContext(p);
```

The Context.PROVIDER_URL property points to the HA-JNDI service configured in the HANamingService MBean (see the section called “JBoss configuration”).

However, this does not work in all cases, especially when running a multi-homed cluster (several JBoss instances on one machine bound to different IPs). A safer method is not to specify the `Context.PROVIDER_URL` (which does not work in all scenarios) but the partition name property:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
p.put("jnp.partitionName", "DefaultPartition"); // partition name.
return new InitialContext(p);
```

Do not attempt to simplify things by placing a `jndi.properties` file in your deployment or by editing the AS's `conf/jndi.properties` file. Doing either will almost certainly break things for your application and quite possibly across the application server. If you want to externalize your client configuration, one approach is to deploy a properties file not named `jndi.properties`, and then programmatically create a `Properties` object that loads that file's contents.

14.2.2. For clients running outside the application server

The JNDI client needs to be aware of the HA-JNDI cluster. You can pass a list of JNDI servers (i.e., the nodes in the HA-JNDI cluster) to the `java.naming.provider.url` JNDI setting in the `jndi.properties` file. Each server node is identified by its IP address and the JNDI port number. The server nodes are separated by commas (see [Section 14.2.3, “JBoss configuration”](#) for how to configure the servers and ports).

```
java.naming.provider.url=server1:1100,server2:1100,server3:1100,server4:1100
```

When initialising, the JNP client code will try to get in touch with each server node from the list, one after the other, stopping as soon as one server has been reached. It will then download the HA-JNDI stub from this node.



NOTE

There is no load balancing behavior in the JNP client lookup process itself. It just goes through the provider lists and uses the first available server to obtain the stub. The HA-JNDI provider list only needs to contain a subset of HA-JNDI nodes in the cluster.

The downloaded smart proxy contains the list of currently running nodes and the logic to load balance naming requests and to fail-over to another node if necessary. Furthermore, each time a JNDI invocation is made to the server, the list of targets in the proxy interceptor is updated (only if the list has changed since the last call).

If the property string `java.naming.provider.url` is empty or if all servers it mentions are not reachable, the JNP client will try to discover a HA-JNDI server through a multicast call on the network (auto-discovery). See the section called “JBoss configuration” on how to configure auto-discovery on the JNDI server nodes. Through auto-discovery, the client might be able to get a valid HA-JNDI server node without any configuration. Of course, for auto-discovery to work, the network segment(s) between the client and the server cluster must be configured to propagate such multicast datagrams.

**NOTE**

By default the auto-discovery feature uses multicast group address 230.0.0.4 and port 1102.

In addition to the `java.naming.provider.url` property, you can specify a set of other properties. The following list shows all clustering-related client side properties you can specify when creating a new InitialContext. (All of the standard, non-clustering-related environment properties used with regular JNDI are also available.)

- **java.naming.provider.url**: Provides a list of IP addresses and port numbers for HA-JNDI provider nodes in the cluster. The client tries those providers one by one and uses the first one that responds.
- **jnp.disableDiscovery**: When set to **true**, this property disables the automatic discovery feature. Default is **false**.
- **jnp.partitionName**: In an environment where multiple HA-JNDI services bound to distinct clusters (a.k.a. partitions), are running, this property allows you to ensure that your client only accepts automatic-discovery responses from servers in the desired partition. If you do not use the automatic discovery feature (i.e. `jnp.disableDiscovery` is **true**), this property is not used. By default, this property is not set and the automatic discovery select the first HA-JNDI server that responds, irregardless of the cluster partition name.
- **jnp.discoveryTimeout**: Determines how much time the context will wait for a response to its automatic discovery packet. Default is 5000 ms.
- **jnp.discoveryGroup**: Determines which multicast group address is used for the automatic discovery. Default is 230.0.0.4. Must match the value of the `AutoDiscoveryAddress` configured on the server side HA-JNDI service.
- **jnp.discoveryPort**: Determines which multicast group port is used for the automatic discovery. Default is 1102. Must match the value of the `AutoDiscoveryPort` configured on the server side HA-JNDI service.
- **jnp.discoveryTTL**: specifies the TTL (time-to-live) for autodiscovery IP multicast packets. This value represents the number of network hops a multicast packet can be allowed to propagate before networking equipment should drop the packet. Despite its name, it does not represent a unit of time.

14.2.3. JBoss configuration

The `cluster-service.xml` file in the `all/deploy` directory includes the following MBean to enable HA-JNDI services.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
      name="jboss:service=HAJNDI">
  <depends optional-attribute-name="ClusterPartition"
    proxy-
    type="attribute">jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
</mbean>
```

You can see that this MBean depends on the **DefaultPartition** MBean defined above it (discussed

earlier in this chapter). In other configurations, you can put that element in the `jboss-service.xml` file or any other JBoss configuration files in the `/deploy` directory to enable HA-JNDI services. The available attributes for this MBean are listed below.

- **Cluster Partition** is a required attribute to inject the HAPartition service that HA-JNDI uses for intra-cluster communication.
- **BindAddress** is an optional attribute to specify the address to which the HA-JNDI server will bind waiting for JNP clients. Only useful for multi-homed computers. The default value is the value of the `jboss.bind.address` system property, or the host's default address if that property is not set. The `jboss.bind.address` system property is set if the `-b` command line switch is used when JBoss is started.
- **Port** is an optional attribute to specify the port to which the HA-JNDI server will bind waiting for JNP clients. The default value is **1100**.
- **Backlog** is an optional attribute to specify the backlog value used for the TCP server socket waiting for JNP clients. The default value is **50**.
- **RmiPort** determines which port the server should use to communicate with the downloaded stub. This attribute is optional. The default value is 1101. If no value is set, the server automatically assigns a RMI port.
- **DiscoveryDisabled** is a boolean flag that disables configuration of the auto discovery multicast listener.
- **AutoDiscoveryAddress** is an optional attribute to specify the multicast address to listen to for JNDI automatic discovery. The default value is the value of the `jboss.partition.udpGroup` system property, or 230.0.0.4 if that is not set. The `jboss.partition.udpGroup` system property is set if the `-u` command line switch is used when JBoss is started.
- **AutoDiscoveryGroup** is an optional attribute to specify the multicast group to listen to for JNDI automatic discovery.. The default value is **1102**.
- **AutoDiscoveryBindAddress** sets the interface on which HA-JNDI should listen for auto-discovery request packets. If this attribute is not specified and a **BindAddress** is specified, the **BindAddress** will be used..
- **AutoDiscoveryTTL** specifies the TTL (time-to-live) for autodiscovery IP multicast packets. This value represents the number of network hops a multicast packet can be allowed to propagate before networking equipment should drop the packet. Despite its name, it does not represent a unit of time.
- **LoadBalancePolicy** specifies the class name of the LoadBalancePolicyimplementation that should be included in the client proxy. See the earlier section on "Load-Balancing Policies" for details.
- **LookupPool** specifies the thread pool service used to control the bootstrap and auto discovery lookups.

The full default configuration of the **HANamingService** MBean is as follows.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
name="jboss:service=HAJNDI">
  <!-- We now inject the partition into the HAJNDI service instead
of requiring that the partition name be passed -->
```

```

    <depends optional-attribute-name="ClusterPartition"
    proxy-
type="attribute">jboss:service=${jboss.partition.name:DefaultPartition}
</depends>
    <!-- Bind address of bootstrap and HA-JNDI RMI endpoints -->
    <attribute name="BindAddress">${jboss.bind.address}</attribute>
    <!-- Port on which the HA-JNDI stub is made available -->
    <attribute name="Port">1100</attribute>
    <!-- RmiPort to be used by the HA-JNDI service once bound. 0 => auto. --
>
    <attribute name="RmiPort">1101</attribute>
    <!-- Accept backlog of the bootstrap socket -->
    <attribute name="Backlog">50</attribute>
    <!-- The thread pool service used to control the bootstrap and auto
discovery lookups -->
    <depends optional-attribute-name="LookupPool"
    proxy-type="attribute">jboss.system:service=ThreadPool</depends>
    <!-- A flag to disable the auto discovery via multicast -->
    <attribute name="DiscoveryDisabled">false</attribute>
    <!-- Set the auto-discovery bootstrap multicast bind address. If not
specified and a BindAddress is specified, the BindAddress will be used.
-->
    <attribute name="AutoDiscoveryBindAddress">${jboss.bind.address}
</attribute>
    <!-- Multicast Address and group port used for auto-discovery -->
    <attribute
name="AutoDiscoveryAddress">${jboss.partition.udpGroup:230.0.0.4}
</attribute>
    <attribute name="AutoDiscoveryGroup">1102</attribute>
    <!-- The TTL (time-to-live) for autodiscovery IP multicast packets -->
    <attribute name="AutoDiscoveryTTL">16</attribute>
    <!-- The load balancing policy for HA-JNDI -->
    <attribute
name="LoadBalancePolicy">org.jboss.ha.framework.interfaces.RoundRobin</att
ribute>

    <!-- Client socket factory to be used for client-server
RMI invocations during JNDI queries
<attribute name="ClientSocketFactory">custom</attribute>
-->
    <!-- Server socket factory to be used for client-server
RMI invocations during JNDI queries
<attribute name="ServerSocketFactory">custom</attribute>
-->
    </mbean>

```

It is possible to start several HA-JNDI services that use different clusters. This can be used, for example, if a node is part of many clusters. In this case, make sure that you set a different port or IP address for each services. For instance, if you wanted to hook up HA-JNDI to the example cluster you set up and change the binding port, the Mbean descriptor would look as follows.

```

<mbean code="org.jboss.ha.jndi.HANamingService"
    name="jboss:service=HAJNDI">

    <depends optional-attribute-name="ClusterPartition"

```



```
    proxy-type="attribute">jboss:service=MySpecialPartition</depends>  
    <attribute name="Port">56789</attribute>  
</mbean>
```

CHAPTER 15. CLUSTERED SESSION EJBS

Session EJBS provide remote invocation services. They are clustered based on the client-side interceptor architecture. The client application for a clustered session bean is exactly the same as the client for the non-clustered version of the session bean, except for a minor change to the `java.naming.provider.url` system property to enable HA-JNDI lookup (see previous section). No code change or re-compilation is needed on the client side. Now, let's check out how to configure clustered session beans in EJB 2.x and EJB 3.0 server applications respectively.

15.1. STATELESS SESSION BEAN IN EJB 2.X

Clustering stateless session beans is most probably the easiest case: as no state is involved, calls can be load-balanced on any participating node (i.e. any node that has this specific bean deployed) of the cluster. To make a bean clustered, you need to modify its `jboss.xml` descriptor to contain a `<clustered>` tag.

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>nextgen.StatelessSession</ejb-name>
      <jndi-name>nextgen.StatelessSession</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </home-load-balance-policy>
        <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </bean-load-balance-policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```



NOTE

The `<clustered>True</clustered>` element is really just an alias for the `<configuration-name>Clustered Stateless SessionBean</configuration-name>` element in the `conf/standard-jboss.xml` file.

In the bean configuration, only the `<clustered>` element is mandatory. It indicates that the bean needs to support clustering features. The `<cluster-config>` element is optional and the default values of its attributes are indicated in the sample configuration above. Below is a description of the attributes in the `<cluster-config>` element..

- **partition-name** specifies the name of the cluster the bean participates in. The default value is **DefaultPartition**. The default partition name can also be set system-wide using the `jboss.partition.name` system property.
- **home-load-balance-policy** indicates the class to be used by the home stub to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a **RoundRobin** fashion. You can also implement your own load-balance policy class or use the class

FirstAvailable that persists to use the first node available that it meets until it fails.

- **bean-load-balance-policy** Indicates the class to be used by the bean stub to balance calls made on the nodes of the cluster. Comments made for the **home-load-balance-policy** attribute also apply.

15.2. STATEFUL SESSION BEAN IN EJB 2.X

Clustering stateful session beans is more complex than clustering their stateless counterparts since JBoss needs to manage the state information. The state of all stateful session beans are replicated and synchronized across the cluster each time the state of a bean changes. The JBoss AS uses the **HASessionState** MBean to manage distributed session states for clustered EJB 2.x stateful session beans. In this section, we cover both the session bean configuration and the **HASessionState** MBean configuration.

15.2.1. The EJB application configuration

In the EJB application, you need to modify the **jboss.xml** descriptor file for each stateful session bean and add the **<clustered>** tag.

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>nextgen.StatefulSession</ejb-name>
      <jndi-name>nextgen.StatefulSession</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </home-load-balance-policy>
        <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.FirstAvailable
        </bean-load-balance-policy>
        <session-state-manager-jndi-name>
          /HASessionState/Default
        </session-state-manager-jndi-name>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

In the bean configuration, only the **<clustered>** tag is mandatory to indicate that the bean works in a cluster. The **<cluster-config>** element is optional and its default attribute values are indicated in the sample configuration above.

The **<session-state-manager-jndi-name>** tag is used to give the JNDI name of the **HASessionState** service to be used by this bean.

The description of the remaining tags is identical to the one for stateless session bean. Actions on the clustered stateful session bean's home interface are by default load-balanced, round-robin. Once the bean's remote stub is available to the client, calls will not be load-balanced round-robin any more and will stay "sticky" to the first node in the list.

15.2.2. Optimize state replication

As the replication process is a costly operation, you can optimise this behaviour by optionally implementing in your bean class a method with the following signature:

```
public boolean isModified ();
```

Before replicating your bean, the container will detect if your bean implements this method. If your bean does, the container calls the **isModified()** method and it only replicates the bean when the method returns **true**. If the bean has not been modified (or not enough to require replication, depending on your own preferences), you can return **false** and the replication would not occur. This feature is available on JBoss AS 3.0.1+ only.

15.2.3. The **HASessionState** service configuration

The **HASessionState** service MBean is defined in the **all/deploy/cluster-service.xml** file.

```
<mbean code="org.jboss.ha.hasessionstate.server.HASessionStateService"
  name="jboss:service=HASessionState">

  <depends>jboss:service=Naming</depends>
  <!-- We now inject the partition into the HAJNDI service instead
of requiring that the partition name be passed -->
<depends optional-attribute-name="ClusterPartition"
  proxy-type="attribute">
  jboss:service=${jboss.partition.name:DefaultPartition}
</depends>
  <!-- JNDI name under which the service is bound -->
  <attribute name="JndiName">/HASessionState/Default</attribute>
  <!-- Max delay before cleaning unreclaimed state.
Defaults to 30*60*1000 => 30 minutes -->
<attribute name="BeanCleaningDelay">0</attribute>
</mbean>
```

The configuration attributes in the **HASessionState** MBean are listed below.

- **ClusterPartition** is a required attribute to inject the **HAPartition** service that HA-JNDI uses for intra-cluster communication.
- **JndiName** is an optional attribute to specify the JNDI name under which this **HASessionState** service is bound. The default value is **/HAPartition/Default**.
- **BeanCleaningDelay** is an optional attribute to specify the number of milliseconds after which the **HASessionState** service can clean a state that has not been modified. If a node, owning a bean, crashes, its brother node will take ownership of this bean. Nevertheless, the container cache of the brother node will not know about it (because it has never seen it before) and will never delete according to the cleaning settings of the bean. That is why the **HASessionState** service needs to do this cleanup sometimes. The default value is **30*60*1000** milliseconds (i.e., 30 minutes).

15.2.4. Handling Cluster Restart

We have covered the HA smart client architecture in the section called “Client-side interceptor architecture”. The default HA smart proxy client can only failover as long as one node in the cluster exists. If there is a complete cluster shutdown, the proxy becomes orphaned and loses knowledge of the available nodes in the cluster. There is no way for the proxy to recover from this. The proxy needs to look up a fresh set of targets out of JNDI/HAJNDI when the nodes are restarted.

The 3.2.7+/4.0.2+ releases contain a `RetryInterceptor` that can be added to the proxy client side interceptor stack to allow for a transparent recovery from such a restart failure. To enable it for an EJB, setup an invoker-proxy-binding that includes the `RetryInterceptor`. Below is an example `jboss.xml` configuration.

```
<jboss>
<session>
  <ejb-name>nextgen_RetryInterceptorStatelessSession</ejb-name>
  <invoker-bindings>
    <invoker>
      <invoker-proxy-binding-name>
        clustered-retry-stateless-rmi-invoker
      </invoker-proxy-binding-name>
      <jndi-name>
        nextgen_RetryInterceptorStatelessSession
      </jndi-name>
    </invoker>
  </invoker-bindings>
  <clustered>true</clustered>
</session>

<invoker-proxy-binding>
  <name>clustered-retry-stateless-rmi-invoker</name>
  <invoker-mbean>jboss:service=invoker,type=jrmpha</invoker-mbean>
  <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
  <proxy-factory-config>
    <client-interceptors>
      <home>
        <interceptor>
          org.jboss.proxy.ejb.HomeInterceptor
        </interceptor>
      </home>
      <interceptor>
        org.jboss.proxy.SecurityInterceptor
      </interceptor>
      <interceptor>
        org.jboss.proxy.TransactionInterceptor
      </interceptor>
      <interceptor>
        org.jboss.proxy.ejb.RetryInterceptor
      </interceptor>
      <interceptor>
        org.jboss.invocation.InvokerInterceptor
      </interceptor>
    </client-interceptors>
  </proxy-factory-config>
  <bean>
    <interceptor>
      org.jboss.proxy.ejb.StatelessSessionInterceptor
    </interceptor>
  </bean>
</invoker-proxy-binding>
```

```
    org.jboss.proxy.SecurityInterceptor
  </interceptor>
  <interceptor>
org.jboss.proxy.TransactionInterceptor
  </interceptor>
  <interceptor>
org.jboss.proxy.ejb.RetryInterceptor
  </interceptor>
  <interceptor>
    org.jboss.invocation.InvokerInterceptor
  </interceptor>
</bean>
  </client-interceptors>
  </proxy-factory-config>
</invoker-proxy-binding>
```

15.2.5. JNDI Lookup Process

In order to recover the HA proxy, the `RetryInterceptor` does a lookup in JNDI. This means that internally it creates a new `InitialContext` and does a JNDI lookup. But, for that lookup to succeed, the `InitialContext` needs to be configured properly to find your naming server. The `RetryInterceptor` will go through the following steps in attempting to determine the proper naming environment properties:

1. It will check its own static `retryEnv` field. This field can be set by client code via a call to `RetryInterceptor.setRetryEnv(Properties)`. This approach to configuration has two downsides: first, it reduces portability by introducing JBoss-specific calls to the client code; and second, since a static field is used only a single configuration per JVM is possible.
2. If the `retryEnv` field is null, it will check for any environment properties bound to a `ThreadLocal` by the `org.jboss.naming.NamingContextFactory` class. To use this class as your naming context factory, in your `jni.properties` set property `java.naming.factory.initial=org.jboss.naming.NamingContextFactory`. The advantage of this approach is use of `org.jboss.naming.NamingContextFactory` is simply a configuration option in your `jni.properties` file, and thus your java code is unaffected. The downside is the naming properties are stored in a `ThreadLocal` and thus are only visible to the thread that originally created an `InitialContext`.
3. If neither of the above approaches yield a set of naming environment properties, a default `InitialContext` is used. If the attempt to contact a naming server is unsuccessful, by default the `InitialContext` will attempt to fall back on multicast discovery to find an HA-JNDI naming server. See the section on “ClusteredJNDI Services” for more on multicast discovery of HA-JNDI.

15.2.6. SingleRetryInterceptor

The `RetryInterceptor` is useful in many use cases, but a disadvantage it has is that it will continue attempting to re-lookup the HA proxy in JNDI until it succeeds. If for some reason it cannot succeed, this process could go on forever, and thus the EJB call that triggered the `RetryInterceptor` will never return. For many client applications, this possibility is unacceptable. As a result, JBoss doesn't make the `RetryInterceptor` part of its default client interceptor stacks for clustered EJBs.

In the 4.0.4.RC1 release, a new flavor of retry interceptor was introduced, the `org.jboss.proxy.ejb.SingleRetryInterceptor`. This version works like the `RetryInterceptor`, but only makes a single attempt to re-lookup the HA proxy in JNDI. If this attempt fails, the EJB call will fail just as if no retry interceptor was used. Beginning with 4.0.4.CR2, the `SingleRetryInterceptor` is part of the default client interceptor stacks for clustered EJBs.

The downside of the `SingleRetryInterceptor` is that if the retry attempt is made during a portion of a cluster restart where no servers are available, the retry will fail and no further attempts will be made.

15.3. STATELESS SESSION BEAN IN EJB 3.0

To cluster a stateless session bean in EJB 3.0, all you need to do is to annotate the bean class with the `@Clustered` annotation. You can pass in the load balance policy and cluster partition as parameters to the annotation. The default load balance policy is

`org.jboss.ha.framework.interfaces.RandomRobin` and the default cluster is `DefaultPartition`. Below is the definition of the `@Clustered` annotation.

```
public @interface Clustered {
    Class loadBalancePolicy() default LoadBalancePolicy.class;
    String partition() default "${jboss.partition.name:DefaultPartition}";
}
```

Here is an example of a clustered EJB 3.0 stateless session bean implementation.

```
@Stateless
@Clustered
public class MyBean implements MySessionInt {

    public void test() {
        // Do something cool
    }
}
```

The `@Clustered` annotation can also be omitted and the clustering configuration applied in `jboss.xml`:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>NonAnnotationStateful</ejb-name>
      <clustered>true</clustered>
      <cluster-config>
        <partition-name>FooPartition</partition-name>
        <load-balance-policy>
          org.jboss.ha.framework.interfaces.RandomRobin
        </load-balance-policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

15.4. STATEFUL SESSION BEANS IN EJB 3.0

To cluster stateful session beans in EJB 3.0, you need to tag the bean implementation class with the `@Cluster` annotation, just as we did with the EJB 3.0 stateless session bean earlier. The `@org.jboss.ejb3.annotation.cache.tree.CacheConfig` annotation can also be applied to the bean to specify caching behavior. Below is the definition of the `@CacheConfig` annotation:

```
public @interface CacheConfig
{
    String name() default "jboss.cache:service=EJB3SFSBClusteredCache";
    int maxSize() default 10000;
    long idleTimeoutSeconds() default 300;
    boolean replicationIsPassivation() default true;
    long removalTimeoutSeconds() default 0;
}
```

- **name** specifies the object name of the JBoss Cache Mbean that should be used for caching the bean (see below for more on this Mbean).
- **maxSize** specifies the maximum number of beans that can be cached before the cache should start passivating beans, using an LRU algorithm.
- **idleTimeoutSeconds** specifies the max period of time a bean can go unused before the cache should passivate it (irregardless of whether maxSize beans are cached.)
- **removalTimeoutSeconds** specifies the max period of time a bean can go unused before the cache should remove it altogether.
- **replicationIsPassivation** specifies whether the cache should consider a replication as being equivalent to a passivation, and invoke any `@PrePassivate` and `@PostActivate` callbacks on the bean. By default true, since replication involves serializing the bean, and preparing for and recovering from serialization is a common reason for implementing the callback methods.

Here is an example of a clustered EJB 3.0 stateful session bean implementation.

```
@Stateful
@Clustered
@CacheConfig(maxSize=5000,removalTimeoutSeconds=18000)
public class MyBean implements MySessionInt {

    private int state = 0;

    public void increment() {
        System.out.println("counter: " + (state++));
    }
}
```

As with stateless beans, the `@Clustered` annotation can also be omitted and the clustering configuration applied in `jboss.xml`; see the example above.

NOTE

The `@CacheConfig` annotation for EJB 3.0 stateful session beans for a non-clustered environment does not define the name property (there is no need for a common cache in such an environment). Below is an example implementation of a non-clustered EJB 3.0 stateful session bean:

```
@Stateful
@Clustered
@CacheConfig(maxSize=5000,removalTimeoutSeconds=18000)
public class MyBean implements MySessionInt {

    private int state = 0;

    public void increment() { System.out.println("counter: " +
        (state++)); }
}
```

As with EJB 2.0 clustered SFSBs, JBoss provides a mechanism whereby a bean implementation can expose a method the container can invoke to check whether the bean's state is not dirty after a request and doesn't need to be replicated. With EJB3, the mechanism is a little more formal; instead of just exposing a method with a known signature, an EJB3 SFSB must implement the `org.jboss.ejb3.cache.Optimized` interface:

```
public interface Optimized {
    boolean isModified();
}
```

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The related MBean service is defined in the **`ejb3-clustered-sfsbcache-service.xml`** file in the **`deploy`** directory. The contents of the file are as follows.

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
    name="jboss.cache:service=EJB3SFSBClusteredCache">

    <attribute name="ClusterName">
      ${jboss.partition.name:DefaultPartition}-SFSBCache
    </attribute>
    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
    <attribute name="CacheMode">REPL_ASYNC</attribute>

    <!-- We want to activate/inactivate regions as beans are deployed -->
    <attribute name="UseRegionBasedMarshalling">true</attribute>
    <!-- Must match the value of "useRegionBasedMarshalling" -->
    <attribute name="InactiveOnStartup">true</attribute>

    <attribute name="ClusterConfig">
      ...
    </attribute>

    <!-- The max amount of time (in milliseconds) we wait until the
    initial state (ie. the contents of the cache) are retrieved from
```

```

existing members. -->
<attribute name="InitialStateRetrievalTimeout">17500</attribute>

<!-- Number of milliseconds to wait until all responses for a
synchronous call have been received.
-->
<attribute name="SyncReplTimeout">17500</attribute>

<!-- Max number of milliseconds to wait for a lock acquisition -->
<attribute name="LockAcquisitionTimeout">15000</attribute>

<!-- Name of the eviction policy class. -->
<attribute name="EvictionPolicyClass">
  org.jboss.cache.eviction.LRUPolicy
</attribute>

<!-- Specific eviction policy configurations. This is LRU -->
<attribute name="EvictionPolicyConfig">
  <config>
    <attribute name="wakeUpIntervalSeconds">5</attribute>
    <name>statefulClustered</name>
    <!-- So default region would never timeout -->
    <region name="/_default_">
      <attribute name="maxNodes">0</attribute>
      <attribute name="timeToIdleSeconds">0</attribute>
    </region>
  </config>
</attribute>

<!-- Store passivated sessions to the file system -->
<attribute name="CacheLoaderConfiguration">
<config>

  <passivation>true</passivation>
<shared>false</shared>

  <cacheloader>
    <class>org.jboss.cache.loader.FileCacheLoader</class>
    <!-- Passivate to the server data dir -->
    <properties>
      location=${jboss.server.data.dir}${/}sfbs
    </properties>
    <async>false</async>
    <fetchPersistentState>true</fetchPersistentState>
    <ignoreModifications>false</ignoreModifications>
  </cacheloader>

  </config>
</attribute>
</mbean>
</server>

```

The configuration attributes in this MBean are essentially the same as the attributes in the standard JBoss Cache **TreeCache** MBean discussed in [Chapter 19, JBossCache and JGroups Services](#). Again, we omitted the JGroups configurations in the **ClusterConfig** attribute (see more in [Section 19.1, “JGroups Configuration”](#)). Two noteworthy items:

- The cache is configured to support eviction. The EJB3 SFSB container uses the JBoss Cache eviction mechanism to manage SFSB passivation. When beans are deployed, the EJB container will programatically add eviction regions to the cache, one region per bean type.
- A JBoss Cache CacheLoader is also configured; again to support SFSB passivation. When beans are evicted from the cache, the cache loader passivates them to a persistent store; in this case to the filesystem in the `$JBOSS_HOME/server/all/data/sfsb` directory. JBoss Cache supports a variety of different CacheLoader implementations that know how to store data to different persistent store types; see the JBoss Cache documentation for details. However, if you change the CacheLoaderConfiguration, be sure that you do not use a shared store (e.g., a single schema in a shared database.) Each node in the cluster must have its own persistent store, otherwise as nodes independently passivate and activate clustered beans, they will corrupt each others data.

CHAPTER 16. CLUSTERED ENTITY EJBS

In a JBoss AS cluster, the entity bean instance caches need to be kept in sync across all nodes. If an entity bean provides remote services, the service methods need to be load balanced as well.

To use a clustered entity bean, the application does not need to do anything special, except for looking up EJB 2.x remote bean references from the clustered HA-JNDI.

16.1. ENTITY BEAN IN EJB 2.X

First of all, it is worth noting that clustering 2.x entity beans is a bad thing to do. It exposes elements that generally are too fine grained for use as remote objects to clustered remote objects and introduces data synchronization problems that are non-trivial. Do NOT use EJB 2.x entity bean clustering unless you fit into the special case situation of read-only, or one read-write node with read-only nodes synched with the cache invalidation services.

To cluster EJB 2.x entity beans, you need to add the **<clustered>** element to the application's **jboss.xml** descriptor file. Below is a typical **jboss.xml** file.

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>nextgen.EnterpriseEntity</ejb-name>
      <jndi-name>nextgen.EnterpriseEntity</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </home-load-balance-policy>
        <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.FirstAvailable
        </bean-load-balance-policy>
      </cluster-config>
    </entity>
  </enterprise-beans>
</jboss>
```

The EJB 2.x entity beans are clustered for load balanced remote invocations. All the bean instances are synchronized to have the same contents on all nodes.

However, clustered EJB 2.x Entity Beans do not have a distributed locking mechanism or a distributed cache. They can only be synchronized by using row-level locking at the database level (see **<row-lock>** in the CMP specification) or by setting the Transaction Isolation Level of your JDBC driver to be **TRANSACTION_SERIALIZABLE**. Because there is no supported distributed locking mechanism or distributed cache Entity Beans use Commit Option "B" by default (See **standardjboss.xml** and the container configurations Clustered CMP 2.x EntityBean, Clustered CMP EntityBean, or Clustered BMP EntityBean). It is not recommended that you use Commit Option "A" unless your Entity Bean is read-only. (There are some design patterns that allow you to use Commit Option "A" with read-mostly beans. You can also take a look at the Seppuku pattern <http://dima.dhs.org/misc/readOnlyUpdates.html>. JBoss may incorporate this pattern into later versions.)



NOTE

If you are using Bean Managed Persistence (BMP), you are going to have to implement synchronization on your own. The MVCSOFT CMP 2.0 persistence engine (see <http://www.jboss.org/jbossgroup/partners.jsp>) provides different kinds of optimistic locking strategies that can work in a JBoss cluster.

16.2. ENTITY BEAN IN EJB 3.0

In EJB 3.0, the entity beans primarily serve as a persistence data model. They do not provide remote services. Hence, the entity bean clustering service in EJB 3.0 primarily deals with distributed caching and replication, instead of load balancing.

16.2.1. Configure the distributed cache

To avoid round trips to the database, you can use a cache for your entities. JBoss EJB 3.0 entity beans are implemented by Hibernate, which has support for a second-level cache. The Hibernate setup used for the JBoss EJB 3.0 implementation uses JBoss Cache as its underlying second-level cache implementation. The second-level cache provides the following functionalities.

- If you persist a cache enabled entity bean instance to the database via the entity manager the entity will be inserted into the cache.
- If you update an entity bean instance and save the changes to the database via the entity manager the entity will be updated in the cache.
- If you remove an entity bean instance from the database via the entity manager the entity will be removed from the cache.
- If loading a cached entity from the database via the entity manager, and that entity does not exist in the database, it will be inserted into the cache.

The JBoss Cache service for EJB 3.0 entity beans is configured in a **TreeCache** MBean in the **deploy/ejb3-entity-cache-service.xml** file. The name of the cache MBean service is **jboss.cache:service=EJB3EntityTreeCache**. Below are the contents of the **ejb3-entity-cache-service.xml** file in the standard JBoss distribution. Again, we omitted the JGroups configuration element **ClusterConfig**.

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
    name="jboss.cache:service=EJB3EntityTreeCache">

    <depends>jboss:service=Naming</depends>
    <depends>jboss:service=TransactionManager</depends>

    <!-- Name of cluster. Needs to be the same on all nodes in the clusters,
         in order to find each other -->
    <attribute name="ClusterName">
      ${jboss.partition.name:DefaultPartition}-EntityCache
    </attribute>

    <!-- Configure the TransactionManager -->
    <attribute name="TransactionManagerLookupClass">
      org.jboss.cache.JBossTransactionManagerLookup
```

```

</attribute>

<attribute name="IsolationLevel">REPEATABLE_READ</attribute>
<attribute name="CacheMode">REPL_SYNC</attribute>

<!-- Must be true if any entity deployment uses a scoped classloader -->
<attribute name="UseRegionBasedMarshalling">true</attribute>
<!-- Must match the value of "useRegionBasedMarshalling" -->
<attribute name="InactiveOnStartup">true</attribute>

<attribute name="ClusterConfig">
    ...
</attribute>

<attribute name="InitialStateRetrievalTimeout">17500</attribute>
<attribute name="SyncReplTimeout">17500</attribute>
<attribute name="LockAcquisitionTimeout">15000</attribute>

<attribute name="EvictionPolicyClass">
org.jboss.cache.eviction.LRUPolicy
</attribute>

<!-- Specific eviction policy configurations. This is LRU -->
<attribute name="EvictionPolicyConfig">
<config>
<attribute name="wakeUpIntervalSeconds">5</attribute>
<!-- Cache wide default -->
<region name="/_default_">
<attribute name="maxNodes">5000</attribute>
<attribute name="timeToLiveSeconds">1000</attribute>
</region>
</config>
</attribute>
</mbean>
</server>

```

This is a replicated cache, so, if running within a cluster, and the cache is updated, changes to the entries in one node will be replicated to the corresponding entries in the other nodes in the cluster.

JBoss Cache allows you to specify timeouts to cached entities. Entities not accessed within a certain amount of time are dropped from the cache in order to save memory. The above configuration sets up a default configuration region that says that at most the cache will hold 5000 nodes, after which nodes will start being evicted from memory, least-recently used nodes last. Also, if any node has not been accessed within the last 1000 seconds, it will be evicted from memory. In general, a node in the cache represents a cached item (entity, collection, or query result set), although there are also a few other node that are used for internal purposes. If the above values of 5000 maxNodes and 1000 idle seconds are invalid for your application(s), you can change the cache-wide defaults. You can also add separate eviction regions for each of your entities; more on this below.

Now, we have JBoss Cache configured to support distributed caching of EJB 3.0 entity beans. We still have to configure individual entity beans to use the cache service.

16.2.2. Configure the entity beans for cache

You define your entity bean classes the normal way. Future versions of JBoss EJB 3.0 will support

annotating entities and their relationship collections as cached, but for now you have to configure the underlying hibernate engine directly. Take a look at the **persistence.xml** file, which configures the caching options for hibernate via its optional **property** elements. The following element in **persistence.xml** defines that caching should be enabled:

```
<!-- Clustered cache with TreeCache -->
<property name="cache.provider_class">
    org.jboss.ejb3.entity.TreeCacheProviderHook
</property>
```

The following property element defines the object name of the cache to be used, i.e., the name of the TreeCache MBean shown above.

```
<property name="treecache.mbean.object_name">
    jboss.cache:service=EJB3EntityTreeCache
</property>
```

Finally, you should give a “region_prefix” to this configuration. This ensures that all cached items associated with this persistence.xml are properly grouped together in JBoss Cache. The jboss.cache:service=EJB3EntityTreeCache cache is a shared resource, potentially used by multiple persistence units. The items cached in that shared cache need to be properly grouped to allow the cache to properly manage classloading. `<property name="hibernate.cache.region_prefix" value="myprefix"/>`

If you do not provide a region prefix, JBoss will automatically provide one for you, building it up from the name of the EAR (if any) and the name of the JAR that includes the persistence.xml. For example, a persistence.xml packaged in foo.ear, bar.jar would be given “foo_ear,bar_jar” as its region prefix. This is not a particularly friendly region prefix if you need to use it to set up specialized eviction regions (see below), so specifying your own region prefix is recommended.

Next we need to configure what entities be cached. The default is to not cache anything, even with the settings shown above. We use the **@org.hibernate.annotations.Cache** annotation to tag entity beans that needs to be cached.

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Account implements Serializable {
    // ...
}
```

A very simplified rule of thumb is that you will typically want to do caching for objects that rarely change, and which are frequently read. You can fine tune the cache for each entity bean in the **ejb3-entity-cache-service.xml** configuration file. For instance, you can specify the size of the cache. If there are too many objects in the cache, the cache could evict oldest objects (or least used objects, depending on configuration) to make room for new objects. Assuming the region_prefix specified in **persistence.xml** was myprefix, the default name of the cache region for the **com.mycompany.entities.Account** entity bean **/myprefix/com/mycompany/entities/Account**.

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
    name="jboss.cache:service=EJB3EntityTreeCache">
    ...
  <attribute name="EvictionPolicyConfig">
    <config>
```

```

        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <region name="/_default_">
            <attribute name="maxNodes">5000</attribute>
            <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
        <!-- Separate eviction rules for Account entities -->
        <region name="/myprefix/com/mycompany/entities/Account">
            <attribute name="maxNodes">10000</attribute>
            <attribute name="timeToLiveSeconds">5000</attribute>
        </region>
        ... ..
    </config>
</attribute>
</mbean>
</server>

```

If you do not specify a cache region for an entity bean class, all instances of this class will be cached in the `/_default` region as defined above. The `@Cache` annotation exposes an optional attribute “region” that lets you specify the cache region where an entity is to be stored, rather than having it be automatically be created from the fully-qualified class name of the entity class.

```

@Entity
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL,
      region="Account")
public class Account implements Serializable {
    // ... ..
}

```

The eviction configuration would then become:

```

<server>
  <mbean code="org.jboss.cache.TreeCache"
        name="jboss.cache:service=EJB3EntityTreeCache">
    ... ..
  <attribute name="EvictionPolicyConfig">
    <config>
      <attribute name="wakeUpIntervalSeconds">5</attribute>
      <region name="/_default_">
        <attribute name="maxNodes">5000</attribute>
        <attribute name="timeToLiveSeconds">1000</attribute>
      </region>
      <!-- Separate eviction rules for Account entities -->
      <region name="/myprefix/Account">
        <attribute name="maxNodes">10000</attribute>
        <attribute name="timeToLiveSeconds">5000</attribute>
      </region>
      ... ..
    </config>
  </attribute>
</mbean>
</server>

```

16.2.3. Query result caching

The EJB3 Query API also provides means for you to save in the second-level cache the results (i.e., collections of primary keys of entity beans, or collections of scalar values) of specified queries. Here we show a simple example of annotating a bean with a named query, also providing the Hibernate-specific hints that tells Hibernate to cache the query.

First, in persistence.xml you need to tell Hibernate to enable query caching:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

Next, you create a named query associated with an entity, and tell Hibernate you want to cache the results of that query:

```
@Entity
@Cache (usage=CacheConcurrencyStrategy.TRANSACTIONAL,
region="Account")
@NamedQueries({
    @NamedQuery(name="account.bybranch",
        query="select acct from Account as acct where acct.branch = ?1",
        hints={@QueryHint(name="org.hibernate.cacheable",value="true")})
})
public class Account implements Serializable {
    // ...
}
```

The @NamedQueries, @NamedQuery and @QueryHint annotations are all in the javax.persistence package. See the Hibernate and EJB3 documentation for more on how to use EJB3 queries and on how to instruct EJB3 to cache queries.

By default, Hibernate stores query results in JBoss Cache in a region named {region_prefix}/org/hibernate/cache/StandardQueryCache. Based on this, you can set up separate eviction handling for your query results. So, if the region prefix were set to myprefix in persistence.xml, you could, for example, create this sort of eviction handling:

```
<server>
    <mbean code="org.jboss.cache.TreeCache"
        name="jboss.cache:service=EJB3EntityTreeCache">
        ...
        <attribute name="EvictionPolicyConfig">
            <config>
                <attribute name="wakeUpIntervalSeconds">5</attribute>
                <region name="/_default_">
                    <attribute name="maxNodes">5000</attribute>
                    <attribute name="timeToLiveSeconds">1000</attribute>
                </region>
                <!-- Separate eviction rules for Account entities -->
                <region name="/myprefix/Account">
                    <attribute name="maxNodes">10000</attribute>
                    <attribute name="timeToLiveSeconds">5000</attribute>
                </region>
                <!-- Cache queries for 10 minutes -->
                <region name="/myprefix/org/hibernate/cache/StandardQueryCache">
                    <attribute name="maxNodes">100</attribute>
                    <attribute name="timeToLiveSeconds">600</attribute>
                </region>
            </config>
        </attribute>
    </mbean>
</server>
```

```

        </region>
        ... ..
    </config>
</attribute>
</mbean>
</server>

```

The `@NamedQuery.hints` attribute shown above takes an array of vendor-specific `@QueryHints` as a value. Hibernate accepts the “org.hibernate.cacheRegion” query hint, where the value is the name of a cache region to use instead of the default `/org/hibernate/cache/StandardQueryCache`. For example:

```

@Entity
@Cache (usage=CacheConcurrencyStrategy.TRANSACTIONAL,
region="Account")
@NamedQueries({
    @NamedQuery(name="account.bybranch",
        query="select acct from Account as acct where acct.branch = ?1",
        hints={@QueryHint(name="org.hibernate.cacheable",value="true"),
        @QueryHint(name="org.hibernate.cacheRegion,value="Queries")
    })
})
public class Account implements Serializable {
    // ... ..
}

```

The related eviction configuration:

```

<server>
  <mbean code="org.jboss.cache.TreeCache"
        name="jboss.cache:service=EJB3EntityTreeCache">
    ... ..
    <attribute name="EvictionPolicyConfig">
      <config>
        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <region name="/_default_">
          <attribute name="maxNodes">5000</attribute>
          <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
        <!-- Separate eviction rules for Account entities -->
        <region name="/myprefix/Account">
          <attribute name="maxNodes">10000</attribute>
          <attribute name="timeToLiveSeconds">5000</attribute>
        </region>
        <!-- Cache queries for 10 minutes -->
        <region name="/myprefix/Queries">
          <attribute name="maxNodes">100</attribute>
          <attribute name="timeToLiveSeconds">600</attribute>
        </region>
        ... ..
      </config>
    </attribute>
  </mbean>
</server>

```

CHAPTER 17. HTTP SERVICES

HTTP session replication is used to replicate the state associated with your web clients on other nodes of a cluster. Thus, in the event one of your node crashes, another node in the cluster will be able to recover. Two distinct functions must be performed:

- Session state replication
- Load-balancing of incoming invocations

State replication is directly handled by JBoss. When you run JBoss in the **all** configuration, session state replication is enabled by default. Just configure your web application as distributable in its **web.xml** (see below), deploy it, and its session state is automatically replicated across all JBoss instances in the cluster.

However, load-balancing is a different story; it is not handled by JBoss itself and requires an external load balancer. This function could be provided by specialized hardware switches or routers (Cisco LoadDirector for example) or by specialized software running on commodity hardware. As a very common scenario, we will demonstrate how to set up a software load balancer using Apache httpd and mod_jk.



NOTE

A load-balancer tracks HTTP requests and, depending on the session to which the request is linked, it dispatches the request to the appropriate node. This is called load-balancing with sticky-sessions: once a session is created on a node, every future request will also be processed by that same node. Using a load-balancer that supports sticky-sessions but not configuring your web application for session replication allows you to scale very well by avoiding the cost of session state replication: each query will always be handled by the same node. But in case a node dies, the state of all client sessions hosted by this node (the shopping carts, for example) will be lost and the clients will most probably need to login on another node and restart with a new session. In many situations, it is acceptable not to replicate HTTP sessions because all critical state is stored in a database. In other situations, losing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

17.1. CONFIGURING LOAD BALANCING USING APACHE AND MOD_JK

Apache is a well-known web server which can be extended by plugging in modules. One of these modules, mod_jk has been specifically designed to allow the forwarding of requests from Apache to a Servlet container. Furthermore, it is also able to load-balance HTTP calls to a set of Servlet containers while maintaining sticky sessions, which is what is most interesting for us in this section.

17.2. DOWNLOAD THE SOFTWARE

First of all, make sure that you have Apache installed. You can download Apache directly from Apache web site at <http://httpd.apache.org/>. Its installation is pretty straightforward and requires no specific configuration. As several versions of Apache exist, we advise you to use version 2.0.x. We will consider, for the next sections, that you have installed Apache in the **APACHE_HOME** directory.

Next, download mod_jk binaries. Several versions of mod_jk exist as well. We strongly advise you to use mod_jk 1.2.x, as both mod_jk and mod_jk2 are deprecated, unsupported and no further developments are going on in the community. The mod_jk 1.2.x binary can be downloaded from

<http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/>. Rename the downloaded file to `mod_jk.so` and copy it under `APACHE_HOME/modules/`.

17.3. CONFIGURE APACHE TO LOAD MOD_JK

Modify `APACHE_HOME/conf/httpd.conf` and add a single line at the end of the file:

```
# Include mod_jk's specific configuration file
Include conf/mod-jk.conf
```

Next, create a new file named `APACHE_HOME/conf/mod-jk.conf`:

```
# Load mod_jk module
# Specify the filename of the mod_jk lib
LoadModule jk_module modules/mod_jk.so

# Where to find workers.properties
JkWorkersFile conf/workers.properties

# Where to put jk logs
JkLogFile logs/mod_jk.log

# Set the jk log level [debug/error/info]
JkLogLevel info

# Select the log format
JkLogStampFormat "[%a %b %d %H:%M:%S %Y]"

# JkOptions indicates to send SSK KEY SIZE
JkOptions +ForwardKeySize +ForwardURISCompat -ForwardDirectories

# JkRequestLogFormat
JkRequestLogFormat "%w %V %T"

# Mount your applications
JkMount /application/* loadbalancer

# You can use external file for mount points.
# It will be checked for updates each 60 seconds.
# The format of the file is: /url=worker
# /examples/*=loadbalancer
JkMountFile conf/uriworkermap.properties

# Add shared memory.
# This directive is present with 1.2.10 and
# later versions of mod_jk, and is needed for
# for load balancing to work properly
JkShmFile logs/jk.shm

# Add jkstatus for managing runtime data
<Location /jkstatus/>
    JkMount status
    Order deny,allow
```

```

        Deny from all
        Allow from 127.0.0.1
    </Location>

```

Please note that two settings are very important:

- The **LoadModule** directive must reference the mod_jk library you have downloaded in the previous section. You must indicate the exact same name with the "modules" file path prefix.
- The **JkMount** directive tells Apache which URLs it should forward to the mod_jk module (and, in turn, to the Servlet containers). In the above file, all requests with URL path **/application/*** are sent to the mod_jk load-balancer. This way, you can configure Apache to server static contents (or PHP contents) directly and only use the loadbalancer for Java applications. If you only use mod_jk as a loadbalancer, you can also forward all URLs (i.e., **/***) to mod_jk.

In addition to the **JkMount** directive, you can also use the **JkMountFile** directive to specify a mount points configuration file, which contains multiple Tomcat forwarding URL mappings. You just need to create a **uriworkermap.properties** file in the **APACHE_HOME/conf** directory. The format of the file is **/url=worker_name**. To get things started, paste the following example into the file you created:

```

# Simple worker configuration file

# Mount the Servlet context to the ajp13 worker
/jmx-console=loadbalancer
/jmx-console/*=loadbalancer
/web-console=loadbalancer
/web-console/*=loadbalancer

```

This will configure mod_jk to forward requests to **/jmx-console** and **/web-console** to Tomcat.

You will most probably not change the other settings in **mod_jk.conf**. They are used to tell mod_jk where to put its logging file, which logging level to use and so on.

17.4. CONFIGURE WORKER NODES IN MOD_JK

Next, you need to configure mod_jk workers file **conf/workers.properties**. This file specifies where the different Servlet containers are located and how calls should be load-balanced across them. The configuration file contains one section for each target servlet container and one global section. For a two nodes setup, the file could look like this:

```

# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status

# Define Node1
# modify the host as your host IP or DNS name.
worker.node1.port=8009
worker.node1.host=node1.mydomain.com
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.cachesize=10

# Define Node2
# modify the host as your host IP or DNS name.
worker.node2.port=8009

```

```
worker.node2.host= node2.mydomain.com
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.cachesize=10

# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node1,node2
worker.loadbalancer.sticky_session=1
#worker.list=loadbalancer

# Status worker for managing load balancer
worker.status.type=status
```

Basically, the above file configures `mod_jk` to perform weighted round-robin load balancing with sticky sessions between two servlet containers (JBoss Tomcat) `node1` and `node2` listening on port 8009.

In the **`works.properties`** file, each node is defined using the **`worker.XXX`** naming convention where **`XXX`** represents an arbitrary name you choose for each of the target Servlet containers. For each worker, you must specify the host name (or IP address) and the port number of the AJP13 connector running in the Servlet container.

The **`lbfactor`** attribute is the load-balancing factor for this specific worker. It is used to define the priority (or weight) a node should have over other nodes. The higher this number is for a given worker relative to the other workers, the more HTTP requests the worker will receive. This setting can be used to differentiate servers with different processing power.

The **`cachesize`** attribute defines the size of the thread pools associated to the Servlet container (i.e. the number of concurrent requests it will forward to the Servlet container). Make sure this number does not outnumber the number of threads configured on the AJP13 connector of the Servlet container. Please review <http://jakarta.apache.org/tomcat/connectors-doc/config/workers.html> for comments on **`cachesize`** for Apache 1.3.x.

The last part of the **`conf/workers.properties`** file defines the `loadbalancer` worker. The only thing you must change is the **`worker.loadbalancer.balanced_workers`** line: it must list all workers previously defined in the same file: load-balancing will happen over these workers.

The **`sticky_session`** property specifies the cluster behavior for HTTP sessions. If you specify **`worker.loadbalancer.sticky_session=0`**, each request will be load balanced between `node1` and `node2`; i.e., different requests for the same session will go to different servers. But when a user opens a session on one server, it is always necessary to always forward this user's requests to the same server, as long as that server is available. This is called a "sticky session", as the client is always using the same server he reached on his first request. To enable session stickiness, you need to set **`worker.loadbalancer.sticky_session`** to 1.



NOTE

A non-loadbalanced setup with a single node requires a **`worker.list=node1`** entry.

17.5. CONFIGURING JBOSS TO WORK WITH MOD_JK

Finally, we must configure the JBoss Tomcat instances on all clustered nodes so that they can expect requests forwarded from the `mod_jk` loadbalancer.

On each clustered JBoss node, we have to name the node according to the name specified in **workers.properties**. For instance, on JBoss instance node1, edit the **JBOSS_HOME/server/all/deploy/jboss-web.deployer/server.xml** file (replace **/all** with your own server name if necessary). Locate the **<Engine>** element and add an attribute **jvmRoute**:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
...
</Engine>
```

You also need to be sure the AJP connector in server.xml is enabled (i.e., uncommented). It is enabled by default.

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

Then, for each JBoss Tomcat instance in the cluster, we need to tell it that **mod_jk** is in use, so it can properly manage the **jvmRoute** appended to its session cookies so that **mod_jk** can properly route incoming requests. Edit the **JBOSS_HOME/server/all/deploy/jboss-web.deployer/META-INF/jboss-service.xml** file (replace **/all** with your own server name). Locate the **<attribute>** element with a name of **UseJK**, and set its value to **true**:

```
<attribute name="UseJK">true</attribute>
```

At this point, you have a fully working Apache+mod_jk load-balancer setup that will balance call to the Servlet containers of your cluster while taking care of session stickiness (clients will always use the same Servlet container).



NOTE

For more updated information on using mod_jk 1.2 with JBoss Tomcat, please refer to the JBoss wiki page at http://wiki.jboss.org/wiki/Wiki.jsp?page=UsingMod_jk1.2WithJBoss.

17.6. CONFIGURING HTTP SESSION STATE REPLICATION

The preceding discussion has been focused on using **mod_jk** as a load balancer. The content of the remainder our discussion of clustering HTTP services in JBoss AS applies no matter what load balancer is used.

In [Section 17.4, “Configure worker nodes in mod_jk”](#), we covered how to use sticky sessions to make sure that a client in a session always hits the same server node in order to maintain the session state. However, sticky sessions by themselves are not an ideal solution. If a node goes down, all its session data is lost. A better and more reliable solution is to replicate session data across the nodes in the cluster. This way, the client can hit any server node and obtain the same session state.

The **jboss.cache.service=TomcatClusteringCache** MBean makes use of JBoss Cache to provide HTTP session replication services to the JBoss Tomcat cluster. This MBean is defined in the **deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml** file.



NOTE

Before AS 4.2.0, the location of the HTTP session cache configuration file was **deploy/tc5-cluster.sar/META-INF/jboss-service.xml**. Prior to AS 4.0.4 CR2, the file was named **deploy/tc5-cluster-service.xml**.

Below is a typical **deploy/jbossweb-cluster.sar/META-INF/jboss-service.xml** file. The configuration attributes in the **TomcatClusteringCache** MBean are very similar to those in the JBoss AS cache configuration.

```
<mbean code="org.jboss.cache.aop.TreeCacheAop"
  name="jboss.cache:service=TomcatClusteringCache">

  <depends>jboss:service=Naming</depends>
  <depends>jboss:service=TransactionManager</depends>
  <depends>jboss.aop:service=AspectDeployer</depends>

  <attribute name="TransactionManagerLookupClass">
    org.jboss.cache.BatchModeTransactionManagerLookup
  </attribute>

  <attribute name="IsolationLevel">REPEATABLE_READ</attribute>

  <attribute name="CacheMode">REPL_ASYNC</attribute>

  <attribute name="ClusterName">
    Tomcat-{$jboss.partition.name:Cluster}
  </attribute>

  <attribute name="UseMarshalling">false</attribute>

  <attribute name="InactiveOnStartup">false</attribute>

  <attribute name="ClusterConfig">
    ... ..
  </attribute>

  <attribute name="LockAcquisitionTimeout">15000</attribute>
  <attribute name="SyncReplTimeout">20000</attribute>
</mbean>
```

Note that the value of the mbean element's code attribute is `org.jboss.cache.aop.TreeCacheAop`, which is different from the other JBoss Cache Mbeans used in JBoss AS. This is because FIELD granularity HTTP session replication (covered below) needs the added features of the **TreeCacheAop** (a.k.a. **PojoCache**) class.

The details of all the configuration options for a `TreeCache` MBean are covered in the JBoss Cache documentation. Below, we will just discuss several attributes that are most relevant to the HTTP cluster session replication.

- **TransactionManagerLookupClass** sets the transaction manager factory. The default value is **org.jboss.cache.BatchModeTransactionManagerLookup**. It tells the cache NOT to participate in JTA-specific transactions. Instead, the cache manages its own transactions. Please do not change this.

- **CacheMode** controls how the cache is replicated. The valid values are **REPL_SYNC** and **REPL_ASYNC**. With either setting the client request thread updates the local cache with the current session contents and then sends a message to the caches on the other members of the cluster, telling them to make the same change. With **REPL_ASYNC** (the default) the request thread returns as soon as the update message has been put on the network. With **REPL_SYNC**, the request thread blocks until it gets a reply message from all cluster members, informing it that the update was successfully applied. Using synchronous replication makes sure changes are applied around the cluster before the web request completes. However, synchronous replication is much slower.
- **ClusterName** specifies the name of the cluster that the cache works within. The default cluster name is the word "Tomcat-" appended by the current JBoss partition name. All the nodes must use the same cluster name.
- The **UseMarshalling** and **InactiveOnStartup** attributes must have the same value. They must be **true** if **FIELD** level session replication is needed (see later). Otherwise, they are default to **false**.
- **ClusterConfig** configures the underlying JGroups stack. Please refer to [Section 19.1, "JGroups Configuration"](#) for more information.
- **LockAcquisitionTimeout** sets the maximum number of milliseconds to wait for a lock acquisition when trying to lock a cache node. The default value is 15000.
- **SyncReplTimeout** sets the maximum number of milliseconds to wait for a response from all nodes in the cluster when a synchronous replication message is sent out. The default value is 20000; should be a few seconds longer than **LockAcquisitionTimeout**.

17.7. ENABLING SESSION REPLICATION IN YOUR APPLICATION

To enable clustering of your web application you must tag it as distributable in the **web.xml** descriptor. Here's an example:

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

You can further configure session replication using the **replication-config** element in the **jboss-web.xml** file. Here is an example:

```
<jboss-web>
  <replication-config>
    <replication-trigger>SET_AND_NON_PRIMITIVE_GET</replication-
trigger>
    <replication-granularity>SESSION</replication-granularity>
    <replication-field-batch-mode>true</replication-field-batch-mode>
  </replication-config>
</jboss-web>
```

-

The **replication-trigger** element determines what triggers a session replication (i.e. when is a session is considered **dirty** and in need of replication). It has 4 options:

- **SET**: With this policy, the session is considered dirty only when an attribute is set in the session (i.e., `HttpSession.setAttribute()` is invoked.) If your application always writes changed values back into the session, this option will be most optimal in terms of performance. The downside of SET is that if an object is retrieved from the session and modified without being written back into the session, the session manager will not know the attribute is dirty and the change to that object may not be replicated.
- **SET_AND_GET**: With this policy, any attribute that is get or set will be marked as dirty. If an object is retrieved from the session and modified without being written back into the session, the change to that object will be replicated. The downside of SET_AND_GET is that it can have significant performance implications, since even reading immutable objects from the session (e.g., strings, numbers) will mark the read attributes as needing to be replicated.
- **SET_AND_NON_PRIMITIVE_GET**: This policy is similar to the SET_AND_GET policy except that get operations that return attribute values with primitive types do not mark the attribute as dirty. Primitive system types (i.e., String, Integer, Long, etc.) are immutable, so there is no reason to mark an attribute with such a type as dirty just because it has been read. If a get operation returns a value of a non-primitive type, the session manager has no simple way to know whether the object is mutable, so it assumes it is and marks the attribute as dirty. This setting avoids the downside of SET while reducing the performance impact of SET_AND_GET. It is the default setting.
- **ACCESS**: This option causes the session to be marked as dirty whenever it is accessed. Since a the session is accessed during each HTTP request, it will be replicated with each request. The purpose of ACCESS is to ensure session last-access timestamps are kept in sync around the cluster.. Since with the other replication-trigger options the time stamp may not be updated in other clustering nodes because of no replication, the session in other nodes may expire before the active node if the HTTP request does not retrieve or modify any session attributes. When this option is set, the session timestamps will be synchronized throughout the cluster nodes. Note that use of this option can have a significant performance impact, so use it with caution. With the other replication-trigger options, if a session has gone 80% of its expiration interval without being replicated, as a safeguard its timestamp will be replicated no matter what. So, ACCESS is only useful in special circumstances where the above safeguard is considered inadequate.

The **replication-granularity** element controls the size of the replication units. The supported values are:

- **ATTRIBUTE**: Replication is only for the dirty attributes in the session plus some session data, like the last-accessed timestamp. For sessions that carry large amounts of data, this option can increase replication performance. However, attributes will be separately serialized, so if there are any shared references between objects stored in the attributes, those shared references may be broken on remote nodes. For example, say a Person object stored under key "husband" has a reference to an Address, while another Person object stored under key "wife" has a reference to that same Address object. When the "husband" and "wife" attributes are separately deserialized on the remote nodes, each Person object will now have a reference to its own Address object; the Address object will no longer be shared.
- **SESSION**: The entire session object is replicated if any attribute is dirty. The entire session is serialized in one unit, so shared object references are maintained on remote nodes. This is the default setting.

- **FIELD:** Replication is only for individual changed data fields inside session attribute objects. Shared object references will be preserved across the cluster. Potentially most performant, but requires changes to your application (this will be discussed later).

The **replication-field-batch-mode** element indicates whether you want all replication messages associated with a request to be batched into one message. Only applicable if replication-granularity is FIELD. Default is **true**.

If your sessions are generally small, SESSION is the better policy. If your session is larger and some parts are infrequently accessed, ATTRIBUTE replication will be more effective. If your application has very big data objects in session attributes and only fields in those objects are frequently modified, the FIELD policy would be the best. In the next section, we will discuss exactly how the FIELD level replication works.

17.8. USING FIELD LEVEL REPLICATION

FIELD-level replication only replicates modified data fields inside objects stored in the session. Its use could potentially drastically reduce the data traffic between clustered nodes, and hence improve the performance of the whole cluster. To use FIELD-level replication, you have to first prepare (i.e., bytecode enhance) your Java class to allow the session cache to detect when fields in cached objects have been changed and need to be replicated.

The first step in doing this is to identify the classes that need to be prepared. This is done via annotations. For example:

```
@org.jboss.cache.aop.AopMarker
public class Address
{
    ...
}
```

If you annotate a class with InstanceAopMarker instead, then all of its subclasses will be automatically annotated as well. Similarly, you can annotate an interface with InstanceofAopMarker and all of its implementing classes will be annotated. For example:

```
@org.jboss.cache.aop.InstanceOfAopMarker
public class Person
{
    ...
}
then when you have a sub-class like
public class Student extends Person
{
    ...
}
```

There will be no need to annotate **Student**. It will be annotated automatically because it is a sub-class of **Person**. Jboss AS 4.2 requires JDK 5 at runtime, but some users may still need to build their projects using JDK 1.4. In this case, annotating classes can be done via JDK 1.4 style annotations embedded in JavaDocs. For example:

```
/*
 * My usual comments here first.
 * @@org.jboss.web.tomcat.tc5.session.AopMarker
 */
```

```
public class Address
{
    ...
}
```

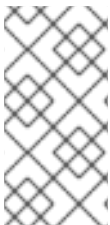
The analogue for **@InstanceAopMarker** is:

```
/*
 *
 * @@org.jboss.web.tomcat.tc5.session.InstanceOfAopMarker
 */
public class Person
{
    ...
}
```

Once you have annotated your classes, you will need to perform a pre-processing step to bytecode enhance your classes for use by TreeCacheAop. You need to use the JBoss AOP pre-compiler **annotationc** and post-compiler **aopc** to process the above source code before and after they are compiled by the Java compiler. The annotationc step is only need if the JDK 1.4 style annotations are used; if JDK 5 annotations are used it is not necessary. Here is an example on how to invoke those commands from command line.

```
$ annotationc [classpath] [source files or directories]
$ javac -cp [classpath] [source files or directories]
$ aopc [classpath] [class files or directories]
```

Please see the JBoss AOP documentation for the usage of the pre- and post-compiler. The JBoss AOP project also provides easy to use ANT tasks to help integrate those steps into your application build process.



NOTE

You can see a complete example on how to build, deploy, and validate a FIELD-level replicated web application from this page: http://wiki.jboss.org/wiki/Wiki.jsp?page=Http_session_field_level_example. The example bundles the pre- and post-compile tools so you do not need to download JBoss AOP separately.

When you deploy the web application into JBoss AS, make sure that the following configurations are correct:

- In the server's **deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml** file, the **inactiveOnStartup** and **useMarshalling** attributes must both be **true**.
- In the application's **jboss-web.xml** file, the **replication-granularity** attribute must be **FIELD**.

Finally, let's see an example on how to use FIELD-level replication on those data classes. Notice that there is no need to call **session.setAttribute()** after you make changes to the data object, and all changes to the fields are automatically replicated across the cluster.

```
// Do this only once. So this can be in init(), e.g.
if(firstTime)
{
```

```

    Person joe = new Person("Joe", 40);
    Person mary = new Person("Mary", 30);
    Address addr = new Address();
    addr.setZip(94086);

    joe.setAddress(addr);
    mary.setAddress(addr); // joe and mary share the same address!

    session.setAttribute("joe", joe); // that's it.
    session.setAttribute("mary", mary); // that's it.
}

Person mary = (Person)session.getAttribute("mary");
mary.getAddress().setZip(95123); // this will update and replicate the zip
code.

```

Besides plain objects, you can also use regular Java collections of those objects as session attributes. JBoss cache automatically figures out how to handle those collections and replicate field changes in their member objects.

17.9. MONITORING SESSION REPLICATION

If you have deployed and accessed your application, go to the `jboss.cache:service=TomcatClusteringCache` MBean and invoke the `printDetails` operation. You should see output resembling the following.

```

/JSESSION

/localhost

/quote

/FB04767C454BAB3B2E462A27CB571330
VERSION: 6
FB04767C454BAB3B2E462A27CB571330:
org.jboss.invocation.MarshalledValue@1f13a81c

/AxCI80vt5VQTfNyYy9Bomw**
VERSION: 4
AxCI80vt5VQTfNyYy9Bomw**: org.jboss.invocation.MarshalledValue@e076e4c8

```

This output shows two separate web sessions, in one application named *quote*, that are being shared via JBossCache. This example uses a **replication-granularity** of **session**. Had **ATTRIBUTE** level replication been used, there would be additional entries showing each replicated session attribute. In either case, the replicated values are stored in an opaque **MarshalledValue** container. There aren't currently any tools that allow you to inspect the contents of the replicated session values. If you do not see any output, either the application was not correctly marked as **distributable** or you haven't accessed a part of application that places values in the HTTP session. The `org.jboss.cache` and `org.jboss.web` logging categories provide additional insight into session replication useful for debugging purposes.

17.10. USING CLUSTERED SINGLE SIGN ON

JBoss supports clustered single sign-on, allowing a user to authenticate to one web application on a

JBoss server and to be recognized on all web applications, on that same machine or on another node in the cluster, that are deployed on the same virtual host. Authentication replication is handled by the same JBoss Cache Mbean that is used by the HTTP session replication service. Although session replication does not need to be explicitly enabled for the applications in question, the **jboss-web-cluster.sar** file needs to be deployed.

To enable single sign-on, you must add the **ClusteredSingleSignOn** valve to the appropriate **Host** elements of the tomcat **server.xml** file. The valve configuration is shown here:

```
<Valve className="org.jboss.web.tomcat.tc5.sso.ClusteredSingleSignOn" />
```

17.11. CLUSTERED SESSION NOTIFICATION POLICY

This section introduces the concept of a clustered session notification policy for determining whether the servlet spec notifications related to session events are allowed to be emitted on the local cluster node. Such notifications are emitted to implementations of the `javax.servlet.http.HttpSessionListener`, `javax.servlet.http.HttpSessionAttributeListener`, and `javax.servlet.http.HttpSessionBindingListener` interfaces.

This new notification policy concept is defined in the **org.jboss.web.tomcat.service.session.notification** Java package, and all classes referenced below are a part of this package.

While it is possible to implement **ClusteredSessionNotificationPolicy** for case-specific customization, JBoss provides **LegacyClusteredSessionNotificationPolicy**, which implements the behavior in previous JBoss versions (the default), and **IgnoreUndeployLegacyClusteredSessionNotificationPolicy**, which implements the same behavior except for undeployment situations in which no **HttpSessionListener** and **HttpSessionAttributeListener** notifications are sent.

It is possible to change from the default **LegacyClusteredSessionNotificationPolicy** to **IgnoreUndeployLegacyClusteredSessionNotificationPolicy** or to a custom **ClusteredSessionNotificationPolicy** implementation by performing one of the following steps:

- Using the system property 'jboss.web.clustered.session.notification.policy' and setting it equal to the class name of the policy you want to choose. This will change the policy for all web applications deployed in the application server.
- Using jboss-web.xml to set the policy per application using the following format:

```
<replication-config>
  <session-notification-policy>
    CLASS-NAME-HERE
  </session-notification-policy>
</replication-config>
```

NOTE

The session-notification-policy element shown above is not part of the official DTD for a JBoss 4.2+ jboss-web.xml (see http://www.jboss.org/j2ee/dtd/jboss-web_4_2.dtd). However, the internal jboss-web.xml parsing logic will properly handle the element in JBoss AS 4.2.4 and later as well as in EAP 4.2.0.GA_CP05 and 4.3.0.GA_CP03 and later.

CHAPTER 18. CLUSTERED SINGLETON SERVICES

A clustered singleton service (also known as an HA singleton) is a service that is deployed on multiple nodes in a cluster, but is providing its service on only one of the nodes. The node running the singleton service is typically called the master node. When the master fails or is shut down, another master is selected from the remaining nodes and the service is restarted on the new master. Thus, other than a brief interval when one master has stopped and another has yet to take over, the service is always being provided by one but only one node.

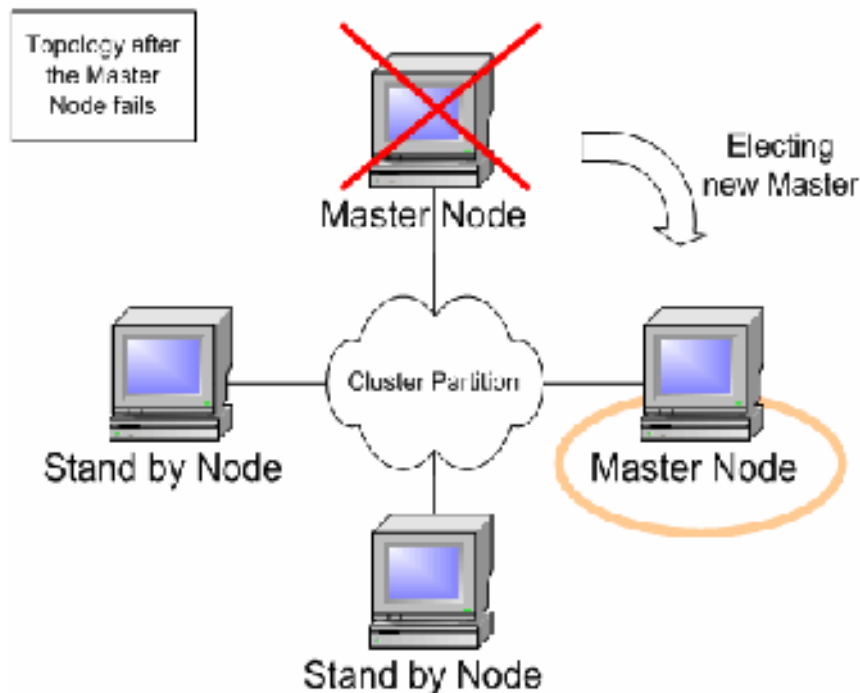


Figure 18.1. Topology after the Master Node fails

The JBoss Application Server (AS) provides support for a number of strategies for helping you deploy clustered singleton services. In this section we will explore the different strategies. All of the strategies are built on top of the **HAPartition** service described in the introduction. They rely on the **HAPartition** to provide notifications when different nodes in the cluster start and stop; based on those notifications each node in the cluster can independently (but consistently) determine if it is now the master node and needs to begin providing a service.

18.1. HASINGLETONDEPLOYER SERVICE

The simplest and most commonly used strategy for deploying an HA singleton is to take an ordinary deployment (war, ear, jar, whatever you would normally put in **deploy**) and deploy it in the **\$JBOSS_HOME/server/all/deploy-hasingleton** directory instead of in **deploy**. The **deploy-hasingleton** directory does not lie under **deploy** or **farm**, so its contents are not automatically deployed when an AS instance starts. Instead, deploying the contents of this directory is the responsibility of a special service, the **jboss.ha:service=HASingletonDeployer** MBean (which itself is deployed via the **deploy/deploy-hasingleton-service.xml** file.) The **HASingletonDeployer** service is itself an HA Singleton, one whose provided service when it becomes master is to deploy the contents of **deploy-hasingleton** and whose service when it stops being the master (typically at server shutdown) is to undeploy the contents of **deploy-hasingleton**.

So, by placing your deployments in **deploy-hasingleton** you know that they will be deployed only on the master node in the cluster. If the master node cleanly shuts down, they will be cleanly undeployed as part of shutdown. If the master node fails or is shut down, they will be deployed on whatever node takes

over as master.

Using `deploy-hasingleton` is very simple, but it does have two drawbacks:

- There is no hot-deployment feature for services in **`deploy-hasingleton`**. Redeploying a service that has been deployed to **`deploy-hasingleton`** requires a server restart.
- If the master node fails and another node takes over as master, your singleton service needs to go through the entire deployment process before it will be providing services. Depending on how complex the deployment of your service is and what sorts of startup activities it engages in, this could take a while, during which time the service is not being provided.

18.2. MBEAN DEPLOYMENTS USING HASINGLETONCONTROLLER

If your service is an MBean (i.e., not a J2EE deployment like an ear or war or jar), you can deploy it along with a service called an `HASingletonController` in order to turn it into an HA singleton. It is the job of the `HASingletonController` to work with the `HAPartition` service to monitor the cluster and determine if it is now the master node for its service. If it determines it has become the master node, it invokes a method on your service telling it to begin providing service. If it determines it is no longer the master node, it invokes a method on your service telling it to stop providing service. Let's walk through an illustration.

First, we have an MBean service that we want to make an HA singleton. The only thing special about it is it needs to expose in its MBean interface a method that can be called when it should begin providing service, and another that can be called when it should stop providing service:

```
public class HASingletonExample
implements HASingletonExampleMBean {

    private boolean isMasterNode = false;

    public void startSingleton() {
        isMasterNode = true;
    }
    .
    public boolean isMasterNode() {
        return isMasterNode;
    }

    public void stopSingleton() {
        isMasterNode = false;
    }
}
```

We used “`startSingleton`” and “`stopSingleton`” in the above example, but you could name the methods anything.

Next, we deploy our service, along with an `HASingletonController` to control it, most likely packaged in a .sar file, with the following **`META-INF/jboss-service.xml`**:

```
<server>
  <!-- This MBean is an example of a clustered singleton -->
  <mbean code="org.jboss.ha.examples.HASingletonExample"
    name="jboss:service=HASingletonExample"/>
```



```

<!-- This HASingletonController manages the cluster Singleton -->
<mbean code="org.jboss.ha.singleton.HASingletonController"
name="jboss:service=ExampleHASingletonController">

  <!-- Inject a ref to the HAPartition -->
  <depends optional-attribute-name="ClusterPartition" proxy-
type="attribute">
    jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
  <!-- Inject a ref to the service being controlled -->
  <depends optional-attribute-name="TargetName">
    jboss:service=HASingletonExample
  </depends>
  <!-- Methods to invoke when become master / stop being master -->
  <attribute name="TargetStartMethod">startSingleton</attribute>
  <attribute name="TargetStopMethod">stopSingleton</attribute>
</mbean>
</server>

```

Voila! A clustered singleton service.

The obvious downside to this approach is it only works for MBeans. Upsides are that the above example can be placed in **deploy** or **farm** and thus can be hot deployed and farmed deployed. Also, if our example service had complex, time-consuming startup requirements, those could potentially be implemented in `create()` or `start()` methods. JBoss will invoke `create()` and `start()` as soon as the service is deployed; it doesn't wait until the node becomes the master node. So, the service could be primed and ready to go, just waiting for the controller to implement `startSingleton()` at which point it can immediately provide service.

The `jboss.ha:service=HASingletonDeployer` service discussed above is itself an interesting example of using an `HASingletonController`. Here is its deployment descriptor (extracted from the **deploy/deploy-hasingleton-service.xml** file):

```

<mbean code="org.jboss.ha.singleton.HASingletonController"
name="jboss.ha:service=HASingletonDeployer">
  <depends optional-attribute-name="ClusterPartition" proxy-
type="attribute">
    jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
  <depends optional-attribute-name="TargetName">
    jboss.system:service=MainDeployer
  </depends>
  <attribute name="TargetStartMethod">deploy</attribute>
  <attribute name="TargetStartMethodArgument">
    ${jboss.server.home.url}/deploy-hasingleton
  </attribute>
  <attribute name="TargetStopMethod">undeploy</attribute>
  <attribute name="TargetStopMethodArgument">
    ${jboss.server.home.url}/deploy-hasingleton
  </attribute>
</mbean>

```

A few interesting things here. First the service being controlled is the **MainDeployer** service, which is the core deployment service in JBoss. That is, it's a service that wasn't written with an intent that it be

controlled by an **HASingletonController**. But it still works! Second, the target start and stop methods are “deploy” and “undeploy”. No requirement that they have particular names, or even that they logically have “start” and “stop” functionality. Here the functionality of the invoked methods is more like “do” and “undo”. Finally, note the “**TargetStart(Stop)MethodArgument**” attributes. Your singleton service's start/stop methods can take an argument, in this case the location of the directory the **MainDeployer** should deploy/undeploy.

18.3. HASINGLETON DEPLOYMENTS USING A BARRIER

Services deployed normally inside deploy or farm that want to be started/stopped whenever the content of deploy-hasingleton gets deployed/undeployed, (i.e., whenever the current node becomes the master), need only specify a dependency on the Barrier mbean:

```
<depends>jboss.ha:service=HASingletonDeployer,type=Barrier</depends>
```

The way it works is that a BarrierController is deployed along with the jboss.ha:service=HASingletonDeployer MBean and listens for JMX notifications from it. A BarrierController is a relatively simple Mbean that can subscribe to receive any JMX notification in the system. It uses the received notifications to control the lifecycle of a dynamically created Mbean called the Barrier. The Barrier is instantiated, registered and brought to the CREATE state when the BarrierController is deployed. After that, the BarrierController starts and stops the Barrier when matching JMX notifications are received. Thus, other services need only depend on the Barrier MBean using the usual <depends> tag, and they will be started and stopped in tandem with the Barrier. When the BarrierController is undeployed the Barrier is destroyed too.

This provides an alternative to the deploy-hasingleton approach in that we can use farming to distribute the service, while content in deploy-hasingleton must be copied manually on all nodes.

On the other hand, the barrier-dependent service will be instantiated/created (i.e., any create() method invoked) on all nodes, but only started on the master node. This is different with the deploy-hasingleton approach that will only deploy (instantiate/create/start) the contents of the deploy-hasingleton directory on one of the nodes.

So services depending on the barrier will need to make sure they do minimal or no work inside their create() step, rather they should use start() to do the work.



NOTE

The Barrier controls the start/stop of dependent services, but not their destruction, which happens only when the **BarrierController** is itself destroyed/undeployed. Thus using the **Barrier** to control services that need to be “destroyed” as part of their normal “undeploy” operation (like, for example, an **EJBContainer**) will not have the desired effect.

18.4. DETERMINING THE MASTER NODE

The various clustered singleton management strategies all depend on the fact that each node in the cluster can independently react to changes in cluster membership and correctly decide whether it is now the “master node”. How is this done?

Prior to JBoss AS 4.2.0, the methodology for this was fixed and simple. For each member of the cluster, the HAPartition mbean maintains an attribute called the CurrentView, which is basically an ordered list of the current members of the cluster. As nodes join and leave the cluster, JGroups ensures that each

surviving member of the cluster gets an updated view. You can see the current view by going into the JMX console, and looking at the `CurrentView` attribute in the `jboss:service=DefaultPartition` mbean. Every member of the cluster will have the same view, with the members in the same order.

Let's say, for example, that we have a 4 node cluster, nodes A through D, and the current view can be expressed as {A, B, C, D}. Generally speaking, the order of nodes in the view will reflect the order in which they joined the cluster (although this is not always the case, and should not be assumed to be the case.)

To further our example, let's say there is a singleton service (i.e., an **HASingletonController**) named `Foo` that's deployed around the cluster, except, for whatever reason, on B. The **HAPartition** service maintains across the cluster a registry of what services are deployed where, in view order. So, on every node in the cluster, the **HAPartition** service knows that the view with respect to the `Foo` service is {A, C, D} (no B).

Whenever there is a change in the cluster topology of the `Foo` service, the **HAPartition** service invokes a callback on `Foo` notifying it of the new topology. So, for example, when `Foo` started on D, the `Foo` service running on A, C and D all got callbacks telling them the new view for `Foo` was {A, C, D}. That callback gives each node enough information to independently decide if it is now the master. The `Foo` service on each node does this by checking if they are the first member of the view – if they are, they are the master; if not, they're not. Simple as that.

If A were to fail or shutdown, `Foo` on C and D would get a callback with a new view for `Foo` of {C, D}. C would then become the master. If A restarted, A, C and D would get a callback with a new view for `Foo` of {C, D, A}. C would remain the master – there's nothing magic about A that would cause it to become the master again just because it was before.

CHAPTER 19. JBOSSCACHE AND JGROUPS SERVICES

JGroups and JBossCache provide the underlying communication, node replication and caching services, for JBoss AS clusters. Those services are configured as MBeans. There is a set of JBossCache and JGroups MBeans for each type of clustering applications (e.g., the Stateful Session EJBs, HTTP session replication etc.).

The JBoss AS ships with a reasonable set of default JGroups and JBossCache MBean configurations. Most applications just work out of the box with the default MBean configurations. You only need to tweak them when you are deploying an application that has special network or performance requirements.

19.1. JGROUPS CONFIGURATION

The JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster. It is built on top a stack of network communication protocols that provide transport, discovery, reliability and failure detection, and cluster membership management services. [Figure 19.1, “Protocol stack in JGroups”](#) shows the protocol stack in JGroups.

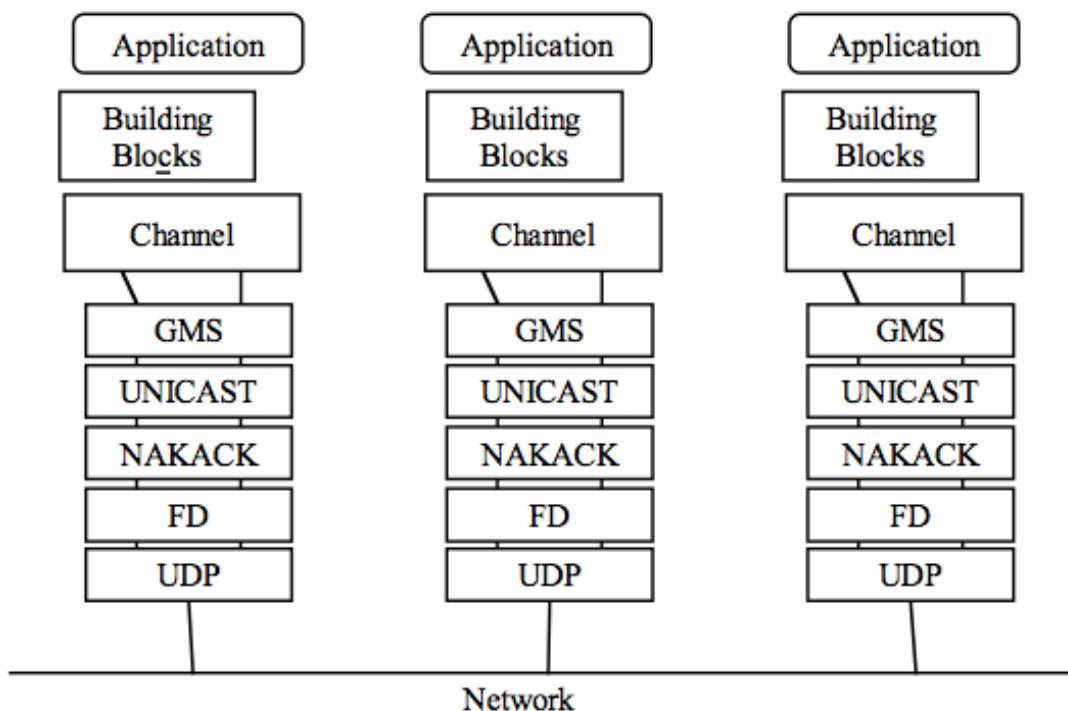


Figure 19.1. Protocol stack in JGroups

JGroups configurations often appear as a nested attribute in cluster related MBean services, such as the **PartitionConfig** attribute in the **ClusterPartition** MBean or the **ClusterConfig** attribute in the **TreeCache** MBean. You can configure the behavior and properties of each protocol in JGroups via those MBean attributes. Below is an example JGroups configuration in the **ClusterPartition** MBean.

```

<mbean code="org.jboss.ha.framework.server.ClusterPartition"
  name="jboss:service=${jboss.partition.name:DefaultPartition}">

  ... ..

  <attribute name="PartitionConfig">
    <Config>

```

```

<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
    mcast_port="${jboss.hapartition.mcast_port:45566}"
    tos="8"
    ucast_recv_buf_size="20000000"
    ucast_send_buf_size="640000"
    mcast_recv_buf_size="25000000"
    mcast_send_buf_size="640000"
    loopback="false"
    discard_incompatible_packets="true"
    enable_bundling="false"
    max_bundle_size="64000"
    max_bundle_timeout="30"
    use_incoming_packet_handler="true"
    use_outgoing_packet_handler="false"
    ip_ttl="${jgroups.udp.ip_ttl:2}"
    down_thread="false" up_thread="false"/>

<PING timeout="2000"
    down_thread="false" up_thread="false" num_initial_members="3"/>

<MERGE2 max_interval="100000"
    down_thread="false" up_thread="false" min_interval="20000"/>
<FD_SOCK down_thread="false" up_thread="false"/>

<FD timeout="10000" max_tries="5"
    down_thread="false" up_thread="false" shun="true"/>
<VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
<pbcast.NAKACK max_xmit_size="60000"
    use_mcast_xmit="false" gc_lag="0"
    retransmit_timeout="300,600,1200,2400,4800"
    down_thread="false" up_thread="false"
    discard_delivered_msgs="true"/>
<UNICAST timeout="300,600,1200,2400,3600"
    down_thread="false" up_thread="false"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
    down_thread="false" up_thread="false"
    max_bytes="400000"/>
<pbcast.GMS print_local_addr="true" join_timeout="3000"
    down_thread="false" up_thread="false"
    join_retry_timeout="2000" shun="true"
    view_bundling="true"/>
<FRAG2 frag_size="60000" down_thread="false" up_thread="false"/>
<pbcast.STATE_TRANSFER down_thread="false"
    up_thread="false" use_flush="false"/>
</Config>
</attribute>
</mbean>

```

All the JGroups configuration data is contained in the <Config> element under the JGroups config MBean attribute. This information is used to configure a JGroups Channel; the Channel is conceptually similar to a socket, and manages communication between peers in a cluster. Each element inside the <Config> element defines a particular JGroups Protocol; each Protocol performs one function, and the combination of those functions is what defines the characteristics of the overall Channel. In the next several sections, we will dig into the commonly used protocols and their options and explain exactly what they mean.

19.2. COMMON CONFIGURATION PROPERTIES

The following common properties are exposed by all of the JGroups protocols discussed below:

- **down_thread** whether the protocol should create an internal queue and a queue processing thread (aka the `down_thread`) for messages passed down from higher layers. The higher layer could be another protocol higher in the stack, or the application itself, if the protocol is the top one on the stack. If true (the default), when a message is passed down from a higher layer, the calling thread places the message in the protocol's queue, and then returns immediately. The protocol's `down_thread` is responsible for reading messages off the queue, doing whatever protocol-specific processing is required, and passing the message on to the next protocol in the stack.
- **up_thread** is conceptually similar to `down_thread`, but here the queue and thread are for messages received from lower layers in the protocol stack.

Generally speaking, **up_thread** and **down_thread** should be set to false.

19.3. TRANSPORT PROTOCOLS

The transport protocols send messages from one cluster node to another (unicast) or from cluster node to all other nodes in the cluster (mcast). JGroups supports UDP, TCP, and TUNNEL as transport protocols.



NOTE

The **UDP**, **TCP**, and **TUNNEL** elements are mutually exclusive. You can only have one transport protocol in each JGroups **Config** element

19.3.1. UDP configuration

UDP is the preferred protocol for JGroups. UDP uses multicast or multiple unicasts to send and receive messages. If you choose UDP as the transport protocol for your cluster service, you need to configure it in the **UDP** sub-element in the JGroups **Config** element. Here is an example.

```
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
    mcast_port="${jboss.hapartition.mcast_port:45566}"
    tos="8"
    ucast_rcv_buf_size="20000000"
    ucast_send_buf_size="640000"
    mcast_rcv_buf_size="25000000"
    mcast_send_buf_size="640000"
    loopback="false"
    discard_incompatible_packets="true"
    enable_bundling="false"
    max_bundle_size="64000"
    max_bundle_timeout="30"
    use_incoming_packet_handler="true"
    use_outgoing_packet_handler="false"
    ip_ttl="${jgroups.udp.ip_ttl:2}"
    down_thread="false" up_thread="false"/>
```

The available attributes in the above JGroups configuration are listed below.

- **ip_mcast** specifies whether or not to use IP multicasting. The default is **true**. If set to false, it will send n unicast packets rather than 1 multicast packet. Either way, packets are UDP datagrams.
- **mcast_addr** specifies the multicast address (class D) for joining a group (i.e., the cluster). If omitted, the default is **228.8.8.8**.
- **mcast_port** specifies the multicast port number. If omitted, the default is **45566**.
- **bind_addr** specifies the interface on which to receive and send multicasts (uses the - **Djgroups.bind_address** system property, if present). If you have a multihomed machine, set the **bind_addr** attribute or system property to the appropriate NIC IP address. By default, system property setting takes priority over XML attribute unless -Djgroups.ignore.bind_addr system property is set.
- **receive_on_all_interfaces** specifies whether this node should listen on all interfaces for multicasts. The default is **false**. It overrides the **bind_addr** property for receiving multicasts. However, **bind_addr** (if set) is still used to send multicasts.
- **send_on_all_interfaces** specifies whether this node send UDP packets via all the NICs if you have a multi NIC machine. This means that the same multicast message is sent N times, so use with care.
- **receive_interfaces** specifies a list of interfaces to receive multicasts on. The multicast receive socket will listen on all of these interfaces. This is a comma-separated list of IP addresses or interface names. E.g. "**192.168.5.1,eth1,127.0.0.1**".
- **ip_ttl** specifies time-to-live for IP Multicast packets. TTL is the commonly used term in multicast networking, but is actually something of a misnomer, since the value here refers to how many network hops a packet will be allowed to travel before networking equipment will drop it.
- **use_incoming_packet_handler** specifies whether to use a separate thread to process incoming messages. Sometimes receivers are overloaded (they have to handle de-serialization etc). Packet handler is a separate thread taking care of de-serialization, receiver thread(s) simply put packet in queue and return immediately. Setting this to true adds one more thread. The default is **true**.
- **use_outgoing_packet_handler** specifies whether to use a separate thread to process outgoing messages. The default is false.
- **enable_bundling** specifies whether to enable message bundling. If it is **true**, the node would queue outgoing messages until **max_bundle_size** bytes have accumulated, or **max_bundle_time** milliseconds have elapsed, whichever occurs first. Then bundle queued messages into a large message and send it. The messages are unbundled at the receiver. The default is **false**.
- **loopback** specifies whether to loop outgoing message back up the stack. In **unicast** mode, the messages are sent to self. In **mcast** mode, a copy of the mcast message is sent. The default is **false**.
- **discard_incompatibe_packets** specifies whether to discard packets from different JGroups versions. Each message in the cluster is tagged with a JGroups version. When a message from a different version of JGroups is received, it will be discarded if set to true, otherwise a warning will be logged. The default is **false**.

- **mcast_send_buf_size**, **mcast_rcv_buf_size**, **ucast_send_buf_size**, **ucast_rcv_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.
- **tos** specifies traffic class for sending unicast and multicast datagrams.



NOTE

On Windows 2000 machines, because of the media sense feature being broken with multicast (even after disabling media sense), you need to set the UDP protocol's **loopback** attribute to **true**.

19.3.2. TCP configuration

Alternatively, a JGroups-based cluster can also work over TCP connections. Compared with UDP, TCP generates more network traffic when the cluster size increases. TCP is fundamentally a unicast protocol. To send multicast messages, JGroups uses multiple TCP unicasts. To use TCP as a transport protocol, you should define a **TCP** element in the JGroups **Config** element. Here is an example of the **TCP** element.

```
<TCP start_port="7800"
      bind_addr="192.168.5.1"
      loopback="true"
      down_thread="false" up_thread="false"/>
```

Below are the attributes available in the **TCP** element.

- **bind_addr** specifies the binding address. It can also be set with the - **Djgroups.bind_address** command line option at server startup.
- **start_port**, **end_port** define the range of TCP ports the server should bind to. The server socket is bound to the first available port from **start_port**. If no available port is found (e.g., because of a firewall) before the **end_port**, the server throws an exception. If no **end_port** is provided or **end_port < start_port** then there is no upper limit on the port range. If **start_port == end_port**, then we force JGroups to use the given port (start fails if port is not available). The default is 7800. If set to 0, then the operating system will pick a port. Please, bear in mind that setting it to 0 will work only if we use MPING or TCPGOSSIP as discovery protocol because **TCCPING** requires listing the nodes and their corresponding ports.
- **loopback** specifies whether to loop outgoing message back up the stack. In **unicast** mode, the messages are sent to self. In **mcast** mode, a copy of the mcast message is sent. The default is false.
- **rcv_buf_size**, **send_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.
- **conn_expire_time** specifies the time (in milliseconds) after which a connection can be closed by the reaper if no traffic has been received.
- **reaper_interval** specifies interval (in milliseconds) to run the reaper. If both values are 0, no reaping will be done. If either value is > 0, reaping will be enabled. By default, **reaper_interval** is 0, which means no reaper.

- **sock_conn_timeout** specifies max time in millis for a socket creation. When doing the initial discovery, and a peer hangs, don't wait forever but go on after the timeout to ping other members. Reduces chances of *not* finding any members at all. The default is 2000.
- **use_send_queues** specifies whether to use separate send queues for each connection. This prevents blocking on write if the peer hangs. The default is true.
- **external_addr** specifies external IP address to broadcast to other group members (if different to local address). This is useful when you have use (Network Address Translation) NAT, e.g. a node on a private network, behind a firewall, but you can only route to it via an externally visible address, which is different from the local address it is bound to. Therefore, the node can be configured to broadcast its external address, while still able to bind to the local one. This avoids having to use the TUNNEL protocol, (and hence a requirement for a central gossip router) because nodes outside the firewall can still route to the node inside the firewall, but only on its external address. Without setting the `external_addr`, the node behind the firewall will broadcast its private address to the other nodes which will not be able to route to it.
- **skip_suspected_members** specifies whether unicast messages should not be sent to suspected members. The default is true.
- **tcp_nodelay** specifies TCP_NODELAY. TCP by default nags messages, that is, conceptually, smaller messages are bundled into larger ones. If we want to invoke synchronous cluster method calls, then we need to disable nagling in addition to disabling message bundling (by setting **enable_bundling** to false). Nagling is disabled by setting **tcp_nodelay** to true. The default is false.

19.3.3. TUNNEL configuration

The TUNNEL protocol uses an external router to send messages. The external router is known as a **GossipRouter**. Each node has to register with the router. All messages are sent to the router and forwarded on to their destinations. The TUNNEL approach can be used to setup communication with nodes behind firewalls. A node can establish a TCP connection to the GossipRouter through the firewall (you can use port 80). The same connection is used by the router to send messages to nodes behind the firewall as most firewalls do not permit outside hosts to initiate a TCP connection to a host inside the firewall. The TUNNEL configuration is defined in the TUNNEL element in the JGroups Config element. Here is an example..

```
<TUNNEL router_port="12001"
  router_host="192.168.5.1"
  down_thread="false" up_thread="false"/>
```

The available attributes in the **TUNNEL** element are listed below.

- **router_host** specifies the host on which the GossipRouter is running.
- **router_port** specifies the port on which the GossipRouter is listening.
- **loopback** specifies whether to loop messages back up the stack. The default is **true**.

19.4. DISCOVERY PROTOCOLS

The cluster needs to maintain a list of current member nodes at all times so that the load balancer and client interceptor know how to route their requests. Discovery protocols are used to discover active nodes in the cluster and detect the oldest member of the cluster, which is the coordinator. All initial nodes

are discovered when the cluster starts up. When a new node joins the cluster later, it is only discovered after the group membership protocol (GMS, see [Section 19.7.1, “Group Membership”](#)) admits it into the group.

Since the discovery protocols sit on top of the transport protocol, you can choose to use different discovery protocols based on your transport protocol. These are also configured as sub-elements in the JGroups MBean **Config** element.

19.4.1. PING

PING is a discovery protocol that works by either multicasting PING requests to an IP multicast address or connecting to a gossip router. As such, PING normally sits on top of the UDP or TUNNEL transport protocols. Each node responds with a packet {C, A}, where C=coordinator's address and A=own address. After timeout milliseconds or num_initial_members replies, the joiner determines the coordinator from the responses, and sends a JOIN request to it (handled by). If nobody responds, we assume we are the first member of a group.

Here is an example PING configuration for IP multicast.

```
<PING timeout="2000"
      num_initial_members="2"
      down_thread="false" up_thread="false"/>
```

Here is another example PING configuration for contacting a Gossip Router.

```
<PING gossip_host="localhost"
      gossip_port="1234"
      timeout="3000"
      num_initial_members="3"
      down_thread="false" up_thread="false"/>
```

The available attributes in the **PING** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **gossip_host** specifies the host on which the GossipRouter is running.
- **gossip_port** specifies the port on which the GossipRouter is listening on.
- **gossip_refresh** specifies the interval (in milliseconds) for the lease from the GossipRouter. The default is 20000.
- **initial_hosts** is a comma-separated list of addresses (e.g., **host1[12345], host2[23456]**), which are pinged for discovery.

If both **gossip_host** and **gossip_port** are defined, the cluster uses the GossipRouter for the initial discovery. If the **initial_hosts** is specified, the cluster pings that static list of addresses for discovery. Otherwise, the cluster uses IP multicasting for discovery.



NOTE

The discovery phase returns when the **timeout** ms have elapsed or the **num_initial_members** responses have been received.

19.4.2. TCPGOSSIP

The TCPGOSSIP protocol only works with a GossipRouter. It works essentially the same way as the PING protocol configuration with valid **gossip_host** and **gossip_port** attributes. It works on top of both UDP and TCP transport protocols. Here is an example.

```
<TCPGOSSIP timeout="2000"
  initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"
  num_initial_members="3"
  down_thread="false" up_thread="false"/>
```

The available attributes in the **TCPGOSSIP** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-separated list of addresses (e.g., **host1[12345],host2[23456]**) for GossipRouters to register with.

19.4.3. TCPPING

The TCPPING protocol takes a set of known members and ping them for discovery. This is essentially a static configuration. It works on top of TCP. Here is an example of the **TCPPING** configuration element in the JGroups **Config** element.

```
<TCPPING timeout="2000"
  initial_hosts="hosta[2300],hostb[3400],hostc[4500]"
  port_range="3"
  num_initial_members="3"
  down_thread="false" up_thread="false"/>
```

The available attributes in the **TCPPING** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-separated list of addresses (e.g., **host1[12345],host2[23456]**) for pinging.
- **port_range** specifies the number of consecutive ports to be probed when getting the initial membership, starting with the port specified in the **initial_hosts** parameter. Given the current values of **port_range** and **initial_hosts** above, the TCPPING layer will try to connect to

hosta:2300, hosta:2301, hosta:2302, hostb:3400, hostb:3401, hostb:3402, hostc:4500, hostc:4501, hostc:4502. The configuration options allows for multiple nodes on the same host to be pinged.

19.4.4. MPING

MPING uses IP multicast to discover the initial membership. It can be used with all transports, but usually this is used in combination with TCP. TCP usually requires TCPPING, which has to list all group members explicitly, but MPING doesn't have this requirement. The typical use case for this is when we want TCP as transport, but multicasting for discovery so we don't have to define a static list of initial hosts in TCPPING or require external Gossip Router.

```
<MPING timeout="2000"
  bind_to_all_interfaces="true"
  mcast_addr="228.8.8.8"
  mcast_port="7500"
  ip_ttl="8"
  num_initial_members="3"
  down_thread="false" up_thread="false"/>
```

The available attributes in the **MPING** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2..
- **bind_addr** specifies the interface on which to send and receive multicast packets.
- **bind_to_all_interfaces** overrides the **bind_addr** and uses all interfaces in multihome nodes.
- **mcast_addr**, **mcast_port**, **ip_ttl** attributes are the same as related attributes in the UDP protocol configuration.

19.5. FAILURE DETECTION PROTOCOLS

The failure detection protocols are used to detect failed nodes. Once a failed node is detected, a suspect verification phase can occur after which, if the node is still considered dead, the cluster updates its view so that the load balancer and client interceptors know to avoid the dead node. The failure detection protocols are configured as sub-elements in the JGroups MBean **Config** element.

19.5.1. FD

FD is a failure detection protocol based on heartbeat messages. This protocol requires each node to periodically send are-you-alive messages to its neighbour. If the neighbour fails to respond, the calling node sends a SUSPECT message to the cluster. The current group coordinator can optionally double check whether the suspected node is indeed dead after which, if the node is still considered dead, updates the cluster's view. Here is an example FD configuration.

```
<FD timeout="2000"
  max_tries="3"
  shun="true"
  down_thread="false" up_thread="false"/>
```

The available attributes in the **FD** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for the responses to the are-you-alive messages. The default is 3000.
- **max_tries** specifies the number of missed are-you-alive messages from a node before the node is suspected. The default is 2.
- **shun** specifies whether a failed node will be shunned. Once shunned, the node will be expelled from the cluster even if it comes back later. The shunned node would have to re-join the cluster through the discovery process. JGroups allows to configure itself such that shunning leads to automatic rejoins and state transfer, which is the default behaviour within JBoss Application Server.



NOTE

Regular traffic from a node counts as if it is a live. So, the are-you-alive messages are only sent when there is no regular traffic to the node for sometime.

19.5.2. FD_SOCKET

FD_SOCKET is a failure detection protocol based on a ring of TCP sockets created between group members. Each member in a group connects to its neighbor (last member connects to first) thus forming a ring. Member B is suspected when its neighbor A detects abnormally closed TCP socket (presumably due to a node B crash). However, if a member B is about to leave gracefully, it lets its neighbor A know, so that it does not become suspected. The simplest FD_SOCKET configuration does not take any attribute. You can just declare an empty **FD_SOCKET** element in JGroups's **Config** element.

```
<FD_SOCKET_down_thread="false" up_thread="false"/>
```

There available attributes in the **FD_SOCKET** element are listed below.

- **bind_addr** specifies the interface to which the server socket should bind to. If -Djgroups.bind_address system property is defined, XML value will be ignore. This behaviour can be reversed setting -Djgroups.ignore.bind_addr=true system property.

19.5.3. VERIFY_SUSPECT

This protocol verifies whether a suspected member is really dead by pinging that member once again. This verification is performed by the coordinator of the cluster. The suspected member is dropped from the cluster group if confirmed to be dead. The aim of this protocol is to minimize false suspicions. Here's an example.

```
<VERIFY_SUSPECT timeout="1500"
  down_thread="false" up_thread="false"/>
```

The available attributes in the **FD_SOCKET** element are listed below.

- **timeout** specifies how long to wait for a response from the suspected member before considering it dead.

19.5.4. FD versus FD_SOCKET

FD and FD_SOCKET, each taken individually, do not provide a solid failure detection layer. Let's look at the differences between these failure detection protocols to understand how they complement each other:

- *FD*
 - An overloaded machine might be slow in sending are-you-alive responses.
 - A member will be suspected when suspended in a debugger/profiler.
 - Low timeouts lead to higher probability of false suspicions and higher network traffic.
 - High timeouts will not detect and remove crashed members for some time.
- *FD_SOCKET*:
 - Suspended in a debugger is no problem because the TCP connection is still open.
 - High load no problem either for the same reason.
 - Members will only be suspected when TCP connection breaks
 - So hung members will not be detected.
 - Also, a crashed switch will not be detected until the connection runs into the TCP timeout (between 2-20 minutes, depending on TCP/IP stack implementation).

The aim of a failure detection layer is to report real failures and therefore avoid false suspicions. There are two solutions:

1. By default, JGroups configures the FD_SOCKET socket with KEEP_ALIVE, which means that TCP sends a heartbeat on socket on which no traffic has been received in 2 hours. If a host crashed (or an intermediate switch or router crashed) without closing the TCP connection properly, we would detect this after 2 hours (plus a few minutes). This is of course better than never closing the connection (if KEEP_ALIVE is off), but may not be of much help. So, the first solution would be to lower the timeout value for KEEP_ALIVE. This can only be done for the entire kernel in most operating systems, so if this is lowered to 15 minutes, this will affect all TCP sockets.
2. The second solution is to combine FD_SOCKET and FD; the timeout in FD can be set such that it is much lower than the TCP timeout, and this can be configured individually per process. FD_SOCKET will already generate a suspect message if the socket was closed abnormally. However, in the case of a crashed switch or host, FD will make sure the socket is eventually closed and the suspect message generated. Example:

```
<FD_SOCKET down_thread="false" up_thread="false"/>
<FD timeout="10000" max_tries="5" shun="true"
down_thread="false" up_thread="false" />
```

This suspects a member when the socket to the neighbor has been closed abnormally (e.g. process crash, because the OS closes all sockets). However, if a host or switch crashes, then the sockets won't be closed, therefore, as a second line of defense, FD will suspect the neighbor after 50 seconds. Note that with this example, if you have your system stopped in a breakpoint in the debugger, the node you're debugging will be suspected after ca 50 seconds.

A combination of FD and FD_SOCKET provides a solid failure detection layer and for this reason, such technique is used across JGroups configurations included within JBoss Application Server.

19.6. RELIABLE DELIVERY PROTOCOLS

Reliable delivery protocols within the JGroups stack ensure that data packets are actually delivered in the right order (FIFO) to the destination node. The basis for reliable message delivery is positive and negative delivery acknowledgments (ACK and NAK). In the ACK mode, the sender resends the message until the acknowledgment is received from the receiver. In the NAK mode, the receiver requests retransmission when it discovers a gap.

19.6.1. UNICAST

The UNICAST protocol is used for unicast messages. It uses ACK. It is configured as a sub-element under the JGroups Config element. If the JGroups stack is configured with TCP transport protocol, UNICAST is not necessary because TCP itself guarantees FIFO delivery of unicast messages. Here is an example configuration for the **UNICAST** protocol.

```
<UNICAST timeout="100,200,400,800"
down_thread="false" up_thread="false"/>
```

There is only one configurable attribute in the **UNICAST** element.

- **timeout** specifies the retransmission timeout (in milliseconds). For instance, if the timeout is "100,200,400,800", the sender resends the message if it hasn't received an ACK after 100 ms the first time, and the second time it waits for 200 ms before resending, and so on.

19.6.2. NAKACK

The NAKACK protocol is used for multicast messages. It uses NAK. Under this protocol, each message is tagged with a sequence number. The receiver keeps track of the sequence numbers and deliver the messages in order. When a gap in the sequence number is detected, the receiver asks the sender to retransmit the missing message. The NAKACK protocol is configured as the **pbcast.NAKACK** sub-element under the JGroups **Config** element. Here is an example configuration.

```
<pbcast.NAKACK max_xmit_size="60000" use_mcast_xmit="false"

  retransmit_timeout="300,600,1200,2400,4800" gc_lag="0"
  discard_delivered_msgs="true"
  down_thread="false" up_thread="false"/>
```

The configurable attributes in the **pbcast.NAKACK** element are as follows.

- **retransmit_timeout** specifies the retransmission timeout (in milliseconds). It is the same as the **timeout** attribute in the UNICAST protocol.
- **use_mcast_xmit** determines whether the sender should send the retransmission to the entire cluster rather than just the node requesting it. This is useful when the sender drops the packet -- so we do not need to retransmit for each node.
- **max_xmit_size** specifies maximum size for a bundled retransmission, if multiple packets are reported missing.
- **discard_delivered_msgs** specifies whether to discard delivery messages on the receiver nodes. By default, we save all delivered messages. However, if we only ask the sender to resend their messages, we can enable this option and discard delivered messages.

- **gc_lag** specifies the number of messages garbage collection lags behind.

19.7. OTHER CONFIGURATION OPTIONS

In addition to the protocol stacks, you can also configure JGroups network services in the **Config** element.

19.7.1. Group Membership

The group membership service in the JGroups stack maintains a list of active nodes. It handles the requests to join and leave the cluster. It also handles the SUSPECT messages sent by failure detection protocols. All nodes in the cluster, as well as the load balancer and client side interceptors, are notified if the group membership changes. The group membership service is configured in the **pbcast.GMS** sub-element under the JGroups **Config** element. Here is an example configuration.

```
<pbcast.GMS print_local_addr="true"
  join_timeout="3000"
  down_thread="false" up_thread="false"
  join_retry_timeout="2000"
  shun="true"
  view_bundling="true"/>
```

The configurable attributes in the **pbcast.GMS** element are as follows.

- **join_timeout** specifies the maximum number of milliseconds to wait for a new node JOIN request to succeed. Retry afterwards.
- **join_retry_timeout** specifies the maximum number of milliseconds to wait after a failed JOIN to re-submit it.
- **print_local_addr** specifies whether to dump the node's own address to the output when started.
- **shun** specifies whether a node should shun itself if it receives a cluster view that it is not a member node.
- **disable_initial_coord** specifies whether to prevent this node as the cluster coordinator.
- **view_bundling** specifies whether multiple JOIN or LEAVE request arriving at the same time are bundled and handled together at the same time, only sending out 1 new view / bundle. This is more efficient than handling each request separately.

19.7.2. Flow Control

The flow control service tries to adapt the sending data rate and the receiving data among nodes. If a sender node is too fast, it might overwhelm the receiver node and result in dropped packets that have to be retransmitted. In JGroups, the flow control is implemented via a credit-based system. The sender and receiver nodes have the same number of credits (bytes) to start with. The sender subtracts credits by the number of bytes in messages it sends. The receiver accumulates credits for the bytes in the messages it receives. When the sender's credit drops to a threshold, the receiver sends some credit to the sender. If the sender's credit is used up, the sender blocks until it receives credits from the receiver. The flow control service is configured in the **FC** sub-element under the JGroups **Config** element. Here is an example configuration.


```
<FC max_credits="1000000"
  down_thread="false" up_thread="false"
  min_threshold="0.10"/>
```

The configurable attributes in the **FC** element are as follows.

- **max_credits** specifies the maximum number of credits (in bytes). This value should be smaller than the JVM heap size.
- **min_credits** specifies the threshold credit on the sender, below which the receiver should send in more credits.
- **min_threshold** specifies percentage value of the threshold. It overrides the **min_credits** attribute.
- **min_block_time** specifies the max time (in ms) a sender blocks. If a sender is blocking, and no credits have been received after 5 seconds, then it sends an explicit credit request to the receivers whose credits are currently below 0, until it receives credits from all members whose credits are below 0.



NOTE

Applications that use synchronous group RPC calls primarily do not require FC protocol in their JGroups protocol stack because synchronous communication, where the thread that makes the call blocks waiting for responses from all the members of the group, already slows overall rate of calls. Even though TCP provides flow control by itself, FC is still required in TCP based JGroups stacks because of group communication, where we essentially have to send group messages at the highest speed the slowest receiver can keep up with. TCP flow control only takes into account individual node communications and has not a notion of who's the slowest in the group, which is why FC is required.

19.7.2.1. Why is FC needed on top of TCP ? TCP has its own flow control !

The reason is group communication, where we essentially have to send group messages at the highest speed the slowest receiver can keep up with. Let's say we have a cluster {A,B,C,D}. D is slow (maybe overloaded), the rest is fast. When A sends a group message, it establishes TCP connections A-A (conceptually), A-B, A-C and A-D (if they don't yet exist). So let's say A sends 100 million messages to the cluster. Because TCP's flow control only applies to A-B, A-C and A-D, but not to A-{B,C,D}, where {B,C,D} is the group, it is possible that A, B and C receive the 100M, but D only received 1M messages. (BTW: this is also the reason why we need NAKACK, although TCP does its own retransmission).

Now JGroups has to buffer all messages in memory for the case when the original sender S dies and a node asks for retransmission of a message of S. Because all members buffer all messages they received, they need to purge stable messages (= messages seen by everyone) every now and then. This is done by the STABLE protocol, which can be configured to run the stability protocol round time based (e.g. every 50s) or size based (whenever 400K data has been received).

In the above case, the slow node D will prevent the group from purging messages above 1M, so every member will buffer 99M messages ! This in most cases leads to OOM exceptions. Note that - although the sliding window protocol in TCP will cause writes to block if the window is full - we assume in the above case that this is still much faster for A-B and A-C than for A-D.

So, in summary, we need to send messages at a rate the slowest receiver (D) can handle.

19.7.2.2. So do I always need FC?

This depends on how the application uses the JGroups channel. Referring to the example above, if there was something about the application that would naturally cause A to slow down its rate of sending because D wasn't keeping up, then FC would not be needed.

A good example of such an application is one that makes synchronous group RPC calls (typically using a JGroups `RpcDispatcher`.) By synchronous, we mean the thread that makes the call blocks waiting for responses from all the members of the group. In that kind of application, the threads on A that are making calls would block waiting for responses from D, thus naturally slowing the overall rate of calls.

A JBoss Cache cluster configured for `REPL_SYNC` is a good example of an application that makes synchronous group RPC calls. If a channel is only used for a cache configured for `REPL_SYNC`, we recommend you remove FC from its protocol stack.

And, of course, if your cluster only consists of two nodes, including FC in a TCP-based protocol stack is unnecessary. There is no group beyond the single peer-to-peer relationship, and TCP's internal flow control will handle that just fine.

Another case where FC may not be needed is for a channel used by a JBoss Cache configured for buddy replication and a single buddy. Such a channel will in many respects act like a two node cluster, where messages are only exchanged with one other node, the buddy. (There may be other messages related to data gravitation that go to all members, but in a properly engineered buddy replication use case these should be infrequent. But if you remove FC be sure to load test your application.)

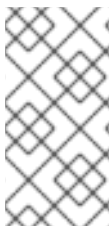
19.7.3. Fragmentation

This protocol fragments messages larger than certain size. Unfragments at the receiver's side. It works for both unicast and multicast messages. It is configured in the `FRAG2` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<FRAG2 frag_size="60000" down_thread="false" up_thread="false"/>
```

The configurable attributes in the `FRAG2` element are as follows.

- **frag_size** specifies the max frag size in bytes. Messages larger than that are fragmented.



NOTE

TCP protocol already provides fragmentation but a fragmentation JGroups protocol is still needed if FC is used. The reason for this is that if you send a message larger than `FC.max_bytes`, FC protocol would block. So, `frag_size` within `FRAG2` needs to be set to always be less than `FC.max_bytes`.

19.7.4. State Transfer

The state transfer service transfers the state from an existing node (i.e., the cluster coordinator) to a newly joining node. It is configured in the **`pbcaster.STATE_TRANSFER`** sub-element under the JGroups **`Config`** element. It does not have any configurable attribute. Here is an example configuration.

```
<pbcaster.STATE_TRANSFER down_thread="false" up_thread="false"/>
```

19.7.5. Distributed Garbage Collection

In a JGroups cluster, all nodes have to store all messages received for potential retransmission in case

of a failure. However, if we store all messages forever, we will run out of memory. So, the distributed garbage collection service in JGroups periodically purges messages that have been seen by all nodes from the memory in each node. The distributed garbage collection service is configured in the **pbcast.STABLE** sub-element under the JGroups **Config** element. Here is an example configuration.

```
<pbcast.STABLE stability_delay="1000"
  desired_avg_gossip="5000"
  down_thread="false" up_thread="false"
  max_bytes="400000"/>
```

The configurable attributes in the **pbcast.STABLE** element are as follows.

- **desired_avg_gossip** specifies intervals (in milliseconds) of garbage collection runs. Value **0** disables this service.
- **max_bytes** specifies the maximum number of bytes received before the cluster triggers a garbage collection run. Value **0** disables this service.
- **stability_delay** specifies delay before we send STABILITY msg (give others a chance to send first). If used together with **max_bytes**, this attribute should be set to a small number.



NOTE

Set the **max_bytes** attribute when you have a high traffic cluster.

19.7.6. Merging

When a network error occurs, the cluster might be partitioned into several different partitions. JGroups has a MERGE service that allows the coordinators in partitions to communicate with each other and form a single cluster back again. The flow control service is configured in the **MERGE2** sub-element under the JGroups **Config** element. Here is an example configuration.

```
<MERGE2 max_interval="10000"
  min_interval="2000"
  down_thread="false" up_thread="false"/>
```

The configurable attributes in the **FC** element are as follows.

- **max_interval** specifies the maximum number of milliseconds to send out a MERGE message.
- **min_interval** specifies the minimum number of milliseconds to send out a MERGE message.

JGroups chooses a random value between **min_interval** and **max_interval** to send out the MERGE message.



NOTE

The cluster states are not merged in a merger. This has to be done by the application. If **MERGE2** is used in conjunction with TCPPING, the **initial_hosts** attribute must contain all the nodes that could potentially be merged back, in order for the merge process to work properly. Otherwise, the merge process would not merge all the nodes even though shunning is disabled. Alternatively use MPING, which is commonly used with TCP to provide multicast member discovery capabilities, instead of TCPPING to avoid having to specify all the nodes.

19.7.7. Binding JGroups Channels to a particular interface

In the Transport Protocols section above, we briefly touched on how the interface to which JGroups will bind sockets is configured. Let's get into this topic in more depth:

First, it's important to understand that the value set in any `bind_addr` element in an XML configuration file will be ignored by JGroups if it finds that system property `jgroups.bind_addr` (or a deprecated earlier name for the same thing, **`bind.address`**) has been set. The system property trumps XML. If JBoss AS is started with the `-b` (a.k.a. `--host`) switch, the AS will set **`jgroups.bind_addr`** to the specified value.

Beginning with AS 4.2.0, for security reasons the AS will bind most services to localhost if `-b` is not set. The effect of this is that in most cases users are going to be setting `-b` and thus `jgroups.bind_addr` is going to be set and any XML setting will be ignored.

So, what are *best practices* for managing how JGroups binds to interfaces?

- Binding JGroups to the same interface as other services. Simple, just use `-b`:

```
./run.sh -b 192.168.1.100 -c all
```

- Binding services (e.g., JBoss Web) to one interface, but use a different one for JGroups:

```
./run.sh -b 10.0.0.100 -Djgroups.bind_addr=192.168.1.100 -c all
```

Specifically setting the system property overrides the `-b` value. This is a common usage pattern; put client traffic on one network, with intra-cluster traffic on another.

- Binding services (e.g., JBoss Web) to all interfaces. This can be done like this:

```
./run.sh -b 0.0.0.0 -c all
```

However, doing this will not cause JGroups to bind to all interfaces! Instead, JGroups will bind to the machine's default interface. See the Transport Protocols section for how to tell JGroups to receive or send on all interfaces, if that is what you really want.

- Binding services (e.g., JBoss Web) to all interfaces, but specify the JGroups interface:

```
./run.sh -b 0.0.0.0 -Djgroups.bind_addr=192.168.1.100 -c all
```

Again, specifically setting the system property overrides the `-b` value.

- Using different interfaces for different channels:

```
./run.sh -b 10.0.0.100 -Djgroups.ignore_bind_addr=true -c all
```

This setting tells JGroups to ignore the **`jgroups.bind_addr`** system property, and instead use whatever is specified in XML. You would need to edit the various XML configuration files to set the **`bind_addr`** to the desired interfaces.

19.7.8. Isolating JGroups Channels

Within JBoss AS, there are a number of services that independently create JGroups channels -- 3 different JBoss Cache services (used for HttpSession replication, EJB3 SFSB replication and EJB3 entity replication) along with the general purpose clustering service called HAPartition that underlies most

other JBossHA services.

It is critical that these channels only communicate with their intended peers; not with the channels used by other services and not with channels for the same service opened on machines not meant to be part of the group. Nodes improperly communicating with each other is one of the most common issues users have with JBoss AS clustering.

Whom a JGroups channel will communicate with is defined by its group name, multicast address, and multicast port, so isolating JGroups channels comes down to ensuring different channels use different values for the group name, multicast address and multicast port.

To isolate JGroups channels for different services on the same set of AS instances from each other, you **MUST** change the group name and the multicast port. In other words, each channel must have its own set of values.

For example, say we have a production cluster of 3 machines, each of which has an HAPartition deployed along with a JBoss Cache used for web session clustering. The HAPartition channels should not communicate with the JBoss Cache channels. They should use a different group name and multicast port. They can use the same multicast address, although they don't need to.

To isolate JGroups channels for the same service from other instances of the service on the network, you **MUST** change ALL three values. Each channel must have its own group name, multicast address, and multicast port.

For example, say we have a production cluster of 3 machines, each of which has an HAPartition deployed. On the same network there is also a QA cluster of 3 machines, which also has an HAPartition deployed. The HAPartition group name, multicast address, and multicast port for the production machines must be different from those used on the QA machines.

19.7.9. Changing the Group Name

The group name for a JGroups channel is configured via the service that starts the channel. Unfortunately, different services use different attribute names for configuring this. For HAPartition and related services configured in the `deploy/cluster-service.xml` file, this is configured via a `PartitionName` attribute. For JBoss Cache services, the name of the attribute is `ClusterName`.

Starting with JBoss AS 4.0.4, for the HAPartition and all the standard JBoss Cache services, we make it easy for you to create unique groups names simply by using the `-g` (a.k.a. `--partition`) switch when starting JBoss:

```
./run.sh -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the `jboss.partition.name` system property, which is used as a component in the configuration of the group name in all the standard clustering configuration files. For example,

```
<attribute name="ClusterName">Tomcat-${jboss.partition.name:Cluster}</attribute>
```

19.7.10. Changing the multicast address and port

The `-u` (a.k.a. `--udp`) command line switch may be used to control the multicast address used by the JGroups channels opened by all standard AS services.

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the `jboss.partition.udpGroup` system property, which you can see referenced in all of the standard protocol stack configs in JBoss AS:

```
<Config>
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
....
```

Unfortunately, setting the multicast ports is not so simple. As described above, by default there are four separate JGroups channels in the standard JBoss AS all configuration, and each should be given a unique port. There are no command line switches to set these, but the standard configuration files do use system properties to set them. So, they can be configured from the command line by using `-D`. For example,

```
/run.sh -u 230.1.2.3 -g QAPartition
-Djboss.messaging.controlchanneludpport=21234
-Djboss.messaging.datachanneludpport=22345
-Djboss.hapartition.mcast_port=12345 -
Djboss.webpartition.mcast_port=23456 -
Djboss.ejb3entitypartition.mcast_port=34567 -
Djboss.ejb3sfsbpartition.mcast_port=45678 -b 192.168.1.100 -c all
```



NOTE

The ports in the above example are randomly chosen for demonstration purposes only.

Why isn't it sufficient to change the group name?

If channels with different group names share the same multicast address and port, the lower level JGroups protocols in each channel will see, process and eventually discard messages intended for the other group. This will at a minimum hurt performance and can lead to anomalous behavior.

Why do I need to change the multicast port if I change the address?

It should be sufficient to just change the address, but there is a problem on several operating systems whereby packets addressed to a particular multicast port are delivered to all listeners on that port, regardless of the multicast address they are listening on. This problem has been fixed for Linux, where it was most prevalent, as of 4.2 CP06 and 4.3 CP04, but if you have problems when just changing the address, you should change the ports as well.

19.7.11. JGroups Troubleshooting

Nodes do not form a cluster

Make sure your machine is set up correctly for IP multicast. There are 2 test programs that can be used to detect this: `McastReceiverTest` and `McastSenderTest`. Go to the **`$JBOSS_HOME/server/all/lib`** directory and start `McastReceiverTest`, for example:

```
java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -mcast_addr
224.10.10.10 -port 5555
```

Then in another window start **`McastSenderTest`**:

```
java -cp jgroups.jar org.jgroups.tests.McastSenderTest -mcast_addr
224.10.10.10 -port 5555
```

If you want to bind to a specific network interface card (NIC), use **-bind_addr 192.168.0.2**, where 192.168.0.2 is the IP address of the NIC to which you want to bind. Use this parameter in both the sender and the receiver.

You should be able to type in the **McastSenderTest** window and see the output in the **McastReceiverTest** window. If not, try to use **-ttl 32** in the sender. If this still fails, consult a system administrator to help you setup IP multicast correctly, and ask the admin to make sure that multicast will work on the interface you have chosen or, if the machines have multiple interfaces, ask to be told the correct interface. Once you know multicast is working properly on each machine in your cluster, you can repeat the above test to test the network, putting the sender on one machine and the receiver on another.

19.7.12. Causes of missing heartbeats in FD

Sometimes a member is suspected by FD because a heartbeat ack has not been received for some time *T* (defined by *timeout* and *max_tries*). This can have multiple reasons, e.g. in a cluster of A,B,C,D; C can be suspected if (note that A pings B, B pings C, C pings D and D pings A):

- B or C are running at 100% CPU for more than *T* seconds. So even if C sends a heartbeat ack to B, B may not be able to process it because it is at 100%
- B or C are garbage collecting, same as above.
- A combination of the 2 cases above
- The network loses packets. This usually happens when there is a lot of traffic on the network, and the switch starts dropping packets (usually broadcasts first, then IP multicasts, TCP packets last).
- B or C are processing a callback. Let's say C received a remote method call over its channel and takes *T*+1 seconds to process it. During this time, C will not process any other messages, including heartbeats, and therefore B will not receive the heartbeat ack and will suspect C.

PART IV. LEGACY EJB SUPPORT

CHAPTER 20. EJBS ON JBOSS

The EJB Container Configuration and Architecture

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

20.1. THE EJB CLIENT SIDE VIEW

We will begin our tour of the EJB container by looking at the client view of an EJB through the home and remote proxies. It is the responsibility of the container provider to generate the **javax.ejb.EJBHome** and **javax.ejb.EJBObject** for an EJB implementation. A client never references an EJB bean instance directly, but rather references the **EJBHome** which implements the bean home interface, and the **EJBObject** which implements the bean remote interface. [Figure 20.1, “The composition of an EJBHome proxy in JBoss.”](#) shows the composition of an EJB home proxy and its relation to the EJB deployment.

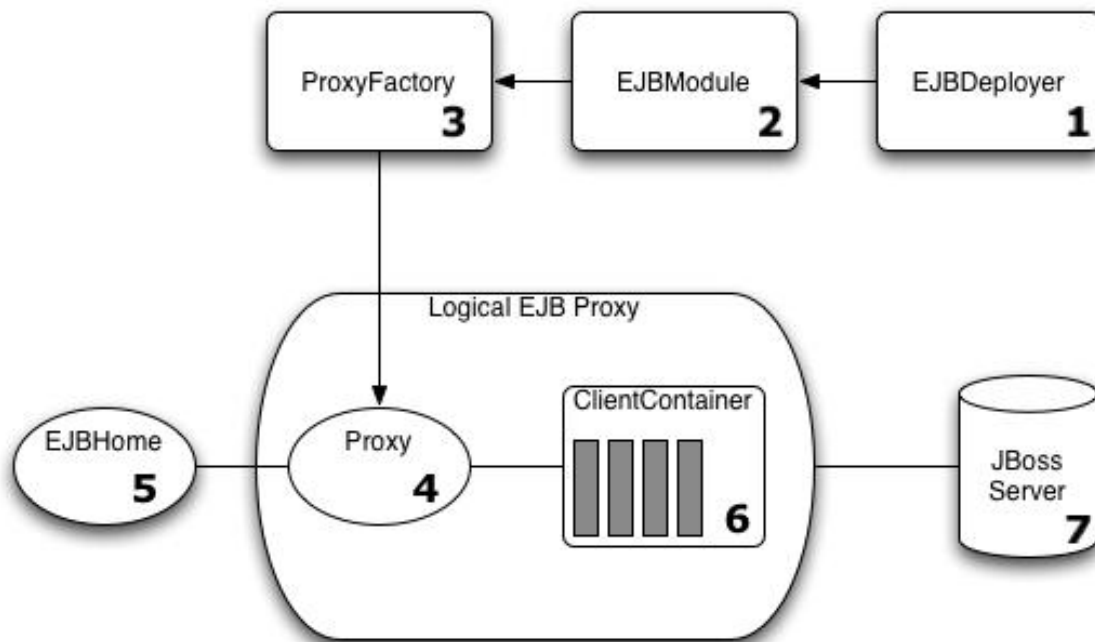


Figure 20.1. The composition of an EJBHome proxy in JBoss.

The numbered items in the figure are:

1. The **EJBDeployer** (**org.jboss.ejb.EJBDeployer**) is invoked to deploy an EJB JAR. An **EJBModule** (**org.jboss.ejb.EJBModule**) is created to encapsulate the deployment metadata.
2. The create phase of the **EJBModule** life cycle creates an **EJBProxyFactory** (**org.jboss.ejb.EJBProxyFactory**) that manages the creation of EJB home and remote interface proxies based on the **EJBModuleInvoker-proxy-bindings** metadata. There can be multiple proxy factories associated with an EJB and we will look at how this is defined shortly.
3. The **ProxyFactory** constructs the logical proxies and binds the homes into JNDI. A logical proxy is composed of a dynamic **Proxy** (**java.lang.reflect.Proxy**), the home interfaces of the EJB that the proxy exposes, the **ProxyHandler**

(**java.lang.reflect.InvocationHandler**) implementation in the form of the **ClientContainer** (**org.jboss.proxy.ClientContainer**), and the client side interceptors.

4. The proxy created by the **EJBProxyFactory** is a standard dynamic proxy. It is a serializable object that proxies the EJB home and remote interfaces as defined in the **EJBModule** metadata. The proxy translates requests made through the strongly typed EJB interfaces into a detyped invocation using the **ClientContainer** handler associated with the proxy. It is the dynamic proxy instance that is bound into JNDI as the EJB home interface that clients lookup. When a client does a lookup of an EJB home, the home proxy is transported into the client VM along with the **ClientContainer** and its interceptors. The use of dynamic proxies avoids the EJB specific compilation step required by many other EJB containers.
5. The EJB home interface is declared in the **ejb-jar.xml** descriptor and available from the **EJBModule** metadata. A key property of dynamic proxies is that they are seen to implement the interfaces they expose. This is true in the sense of Java's strong type system. A proxy can be cast to any of the home interfaces and reflection on the proxy provides the full details of the interfaces it proxies.
6. The proxy delegates calls made through any of its interfaces to the **ClientContainer** handler. The single method required of the handler is: **public Object invoke(Object proxy, Method m, Object[] args) throws Throwable**. The **EJBProxyFactory** creates a **ClientContainer** and assigns this as the **ProxyHandler**. The **ClientContainer**'s state consists of an **InvocationContext** (**org.jboss.invocation.InvocationContext**) and a chain of interceptors (**org.jboss.proxy.Interceptor**). The **InvocationContext** contains:
 - the JMX **ObjectName** of the EJB container MBean the **Proxy** is associated with
 - the **javax.ejb.EJBMetaData** for the EJB
 - the JNDI name of the EJB home interface
 - the transport specific invoker (**org.jboss.invocation.Invoker**)

The interceptor chain consists of the functional units that make up the EJB home or remote interface behavior. This is a configurable aspect of an EJB as we will see when we discuss the **jboss.xml** descriptor, and the interceptor makeup is contained in the **EJBModule** metadata. Interceptors (**org.jboss.proxy.Interceptor**) handle the different EJB types, security, transactions and transport. You can add your own interceptors as well.

7. The transport specific invoker associated with the proxy has an association to the server side detached invoker that handles the transport details of the EJB method invocation. The detached invoker is a JBoss server side component.

The configuration of the client side interceptors is done using the **jboss.xmlclient-interceptors** element. When the **ClientContainer** **invoke** method is called it creates an untyped **Invocation** (**org.jboss.invocation.Invocation**) to encapsulate request. This is then passed through the interceptor chain. The last interceptor in the chain will be the transport handler that knows how to send the request to the server and obtain the reply, taking care of the transport specific details.

As an example of the client interceptor configuration usage, consider the default stateless session bean configuration found in the **server/production/standardjboss.xml** descriptor. [Example 20.1, "The client-interceptors from the Standard Stateless SessionBean configuration."](#) shows the

stateless-rmi-invoker client interceptors configuration referenced by the Standard Stateless SessionBean.

Example 20.1. The client-interceptors from the Standard Stateless SessionBean configuration.

```
<invoker-proxy-binding>
  <name>stateless-rmi-invoker</name>
  <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
  <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
    <proxy-factory-config>
      <client-interceptors>
        <home>

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor call-by-value="false">
    org.jboss.invocation.InvokerInterceptor
  </interceptor>
  <interceptor call-by-value="true">
    org.jboss.invocation.MarshallingInvokerInterceptor
  </interceptor>
</home>
</bean>

<interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor call-by-value="false">
    org.jboss.invocation.InvokerInterceptor
  </interceptor>
  <interceptor call-by-value="true">
    org.jboss.invocation.MarshallingInvokerInterceptor
  </interceptor>
</bean>
</client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>

<container-configuration>
  <container-name>Standard Stateless SessionBean</container-name>
  <call-logging>>false</call-logging>
  <invoker-proxy-binding-name>stateless-rmi-invoker</invoker-proxy-binding-name>
  <!-- ... -->
</container-configuration>
```

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

- **org.jboss.proxy.ejb.HomeInterceptor**: handles the **getHomeHandle**, **getEJBMetaData**, and **remove** methods of the **EJBHome** interface locally in the client VM. Any other methods are propagated to the next interceptor.
- **org.jboss.proxy.ejb.StatelessSessionInterceptor**: handles the **toString**, **equals**, **hashCode**, **getHandle**, **getEJBHome** and **isIdentical** methods of the **EJBObject** interface locally in the client VM. Any other methods are propagated to the next interceptor.
- **org.jboss.proxy.SecurityInterceptor**: associates the current security context with the method invocation for use by other interceptors or the server.
- **org.jboss.proxy.TransactionInterceptor**: associates any active transaction with the invocation method invocation for use by other interceptors.
- **org.jboss.invocation.InvokerInterceptor**: encapsulates the dispatch of the method invocation to the transport specific invoker. It knows if the client is executing in the same VM as the server and will optimally route the invocation to a by reference invoker in this situation. When the client is external to the server VM, this interceptor delegates the invocation to the transport invoker associated with the invocation context. In the case of the [Example 20.1, “The client-interceptors from the Standard Stateless SessionBean configuration.”](#) configuration, this would be the invoker stub associated with the **jboss:service=invoker, type=jrmp**, the **JRMPInvoker** service.

org.jboss.invocation.MarshallingInvokerInterceptor: extends the **InvokerInterceptor** to not optimize in-VM invocations. This is used to force **call-by-value** semantics for method calls.

20.1.1. Specifying the EJB Proxy Configuration

To specify the EJB invocation transport and the client proxy interceptor stack, you need to define an **invoker-proxy-binding** in either the EJB JAR **META-INF/jboss.xml** **descriptor**, or the server **standardjboss.xml** descriptor. There are several default **invoker-proxy-bindings** defined in the **standardjboss.xml** descriptor for the various default EJB container configurations and the standard RMI/JRMP and RMI/IIOP transport protocols. The current default proxy configurations are:

- **entity-rmi-invoker**: a RMI/JRMP configuration for entity beans
- **clustered-entity-rmi-invoker**: a RMI/JRMP configuration for clustered entity beans
- **stateless-rmi-invoker**: a RMI/JRMP configuration for stateless session beans
- **clustered-stateless-rmi-invoker**: a RMI/JRMP configuration for clustered stateless session beans
- **stateful-rmi-invoker**: a RMI/JRMP configuration for clustered stateful session beans
- **clustered-stateful-rmi-invoker**: a RMI/JRMP configuration for clustered stateful session beans
- **message-driven-bean**: a JMS invoker for message driven beans
- **singleton-message-driven-bean**: a JMS invoker for singleton message driven beans
- **message-inflow-driven-bean**: a JMS invoker for message inflow driven beans

- **jms-message-inflow-driven-bean**: a JMS inflow invoker for standard message driven beans
- **iiop**: a RMI/IIOP for use with session and entity beans.

To introduce a new protocol binding, or customize the proxy factory, or the client side interceptor stack, requires defining a new **invoker-proxy-binding**. The full **invoker-proxy-binding** DTD fragment for the specification of the proxy configuration is given in [Figure 20.2, “The invoker-proxy-binding schema”](#).

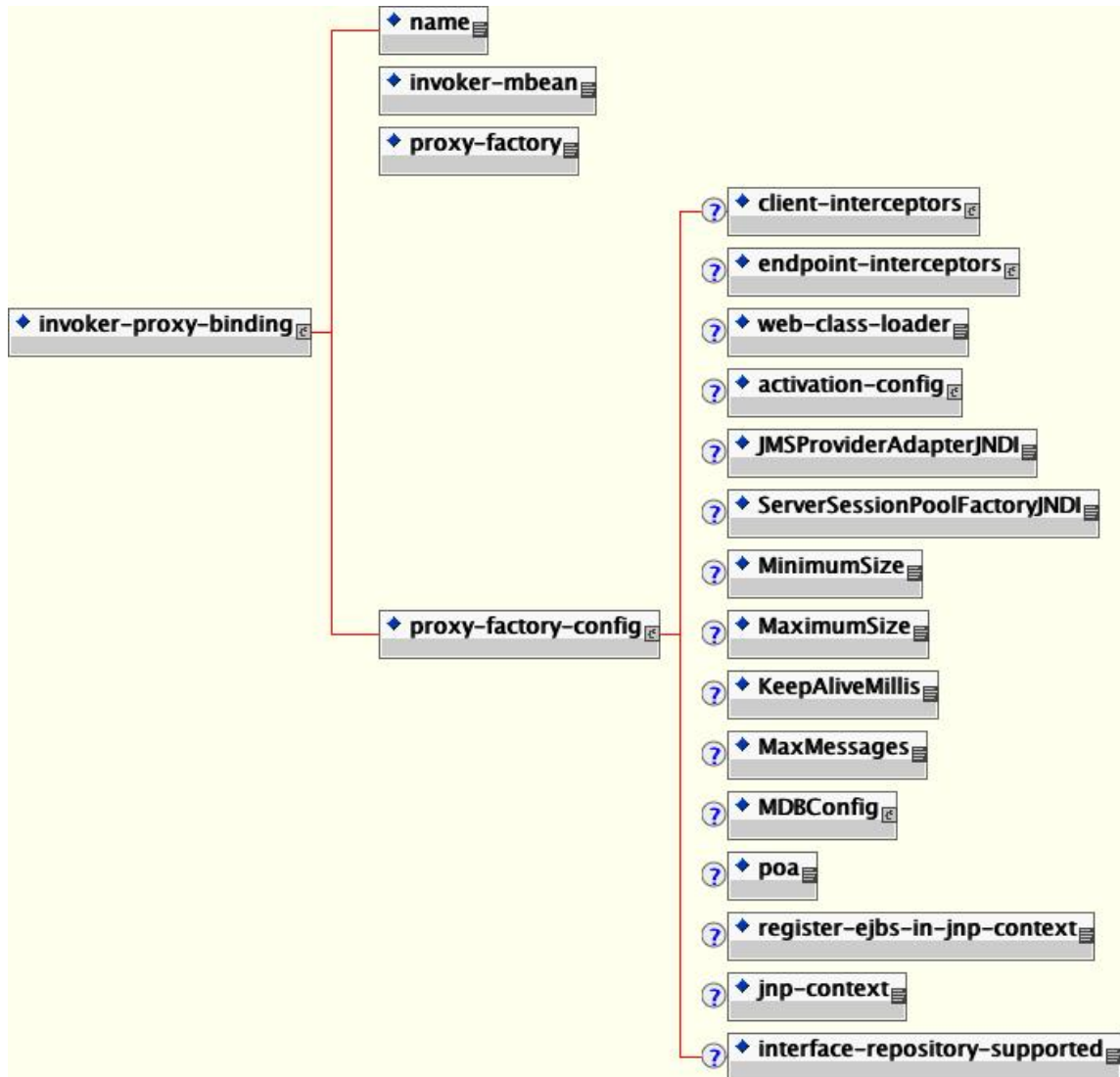


Figure 20.2. The invoker-proxy-binding schema

The **invoker-proxy-binding** child elements are:

- **name**: The **name** element gives a unique name for the **invoker-proxy-binding**. The name is used to reference the binding from the EJB container configuration when setting the default proxy binding as well as the EJB deployment level to specify addition proxy bindings. You will see how this is done when we look at the **jboss.xml** elements that control the server side EJB container configuration.
- **invoker-mbean**: The **invoker-mbean** element gives the JMX **ObjectName** string of the detached invoker MBean service the proxy invoker will be associated with.
- **proxy-factory**: The **proxy-factory** element specifies the fully qualified class name of the proxy factory, which must implement the **org.jboss.ejb.EJBProxyFactory** interface. The **EJBProxyFactory** handles the configuration of the proxy and the association of the protocol

specific invoker and context. The current JBoss implementations of the **EJBProxyFactory** interface include:

- **org.jboss.proxy.ejb.ProxyFactory**: The RMI/JRMP specific factory.
- **org.jboss.proxy.ejb.ProxyFactoryHA**: The cluster RMI/JRMP specific factory.
- **org.jboss.ejb.plugins.jms.JMSContainerInvoker**: The JMS specific factory.
- **org.jboss.proxy.ejb.IORFactory**: The RMI/IIOP specific factory.
- **proxy-factory-config**: The **proxy-factory-config** element specifies additional information for the **proxy-factory** implementation. Unfortunately, its currently an unstructured collection of elements. Only a few of the elements apply to each type of proxy factory. The child elements break down into the three invocation protocols: RMI/RJMP, RMI/IIOP and JMS.

For the RMI/JRMP specific proxy factories, **org.jboss.proxy.ejb.ProxyFactory** and **org.jboss.proxy.ejb.ProxyFactoryHA** the following elements apply:

- **client-interceptors**: The **client-interceptors** define the home, remote and optionally the multi-valued proxy interceptor stacks.
- **web-class-loader**: The web class loader defines the instance of the **org.jboss.web.WebClassLoader** that should be associated with the proxy for dynamic class loading.

The following **proxy-factory-config** is for an entity bean accessed over RMI.

```
<proxy-factory-config>
  <client-interceptors>
    <home>
      <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

    <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor call-by-value="false">
        org.jboss.invocation.InvokerInterceptor
      </interceptor>
      <interceptor call-by-value="true">
        org.jboss.invocation.MarshallingInvokerInterceptor
      </interceptor>
    </home>
  </client-interceptors>
  <bean>

  <interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

  <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
    <interceptor call-by-value="false">
      org.jboss.invocation.InvokerInterceptor
    </interceptor>
    <interceptor call-by-value="true">
      org.jboss.invocation.MarshallingInvokerInterceptor
    </interceptor>
  </bean>
</list-entity>
```

```

<interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
    <interceptor call-by-value="false">
        org.jboss.invocation.InvokerInterceptor
    </interceptor>
    <interceptor call-by-value="true">
        org.jboss.invocation.MarshallingInvokerInterceptor
    </interceptor>
</list-entity>
</client-interceptors>
</proxy-factory-config>

```

For the RMI/IIOP specific proxy factory, **org.jboss.proxy.ejb.IORFactory**, the following elements apply:

- **web-class-loader**: The web class loader defines the instance of the **org.jboss.web.WebClassLoader** that should be associated with the proxy for dynamic class loading.
- **poa**: The portable object adapter usage. Valid values are **per-servant** and **shared**.
- **register-ejbs-in-jnp-context**: A flag indicating if the EJBs should be register in JNDI.
- **jnp-context**: The JNDI context in which to register EJBs.
- **interface-repository-supported**: This indicates whether or not a deployed EJB has its own CORBA interface repository.

The following shows a **proxy-factory-config** for EJBs accessed over IIOP.

```

<proxy-factory-config>
    <web-class-loader>org.jboss.iiop.WebCL</web-class-loader>
    <poa>per-servant</poa>
    <register-ejbs-in-jnp-context>true</register-ejbs-in-jnp-context>
    <jnp-context>iiop</jnp-context>
</proxy-factory-config>

```

For the JMS specific proxy factory, **org.jboss.ejb.plugins.jms.JMSContainerInvoker**, the following elements apply:

- **MinimumSize**: This specifies the minimum pool size for MDBs processing . This defaults to 1.
- **MaximumSize**: This specifies the upper limit to the number of concurrent MDBs that will be allowed for the JMS destination. This defaults to 15.
- **MaxMessages**: This 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. This value should not be modified from the default unless your JMS provider indicates this is supported.

- **KeepAliveMillis**: This specifies the keep alive time interval in milliseconds for sessions in the session pool. The default is 30000 (30 seconds).
- **MDBConfig**: Configuration for the MDB JMS connection behavior. Among the elements supported are:
 - **ReconnectIntervalSec**: The time to wait (in seconds) before trying to recover the connection to the JMS server.
 - **DeliveryActive**: Whether or not the MDB is active at startup. The default is true.
 - **DLQConfig**: Configuration for an MDB's dead letter queue, used when messages are redelivered too many times.
 - **JMSProviderAdapterJNDI**: The JNDI name of the JMS provider adapter in the `java:/` namespace. This is mandatory for an MDB and must implement `org.jboss.jms.jndi.JMSProviderAdapter`.
 - **ServerSessionPoolFactoryJNDI**: The JNDI name of the session pool in the `java:/` namespace of the JMS provider's session pool factory. This is mandatory for an MDB and must implement `org.jboss.jms.asf.ServerSessionPoolFactory`.

Example 20.2, “A sample JMSContainerInvoker proxy-factory-config” gives a sample **proxy-factory-config** fragment taken from the `standardjboss.xml` descriptor.

Example 20.2. A sample JMSContainerInvoker proxy-factory-config

```
<proxy-factory-config>
  <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>

  <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
    <MinimumSize>1</MinimumSize>
    <MaximumSize>15</MaximumSize>
    <KeepAliveMillis>30000</KeepAliveMillis>
    <MaxMessages>1</MaxMessages>
    <MDBConfig>
      <ReconnectIntervalSec>10</ReconnectIntervalSec>
      <DLQConfig>
        <DestinationQueue>queue/DLQ</DestinationQueue>
        <MaxTimesRedelivered>10</MaxTimesRedelivered>
        <TimeToLive>0</TimeToLive>
      </DLQConfig>
    </MDBConfig>
  </proxy-factory-config>
```

20.2. THE EJB SERVER SIDE VIEW

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

20.2.1. Detached Invokers - The Transport Middlemen

We looked at the detached invoker architecture in the context of exposing RMI compatible interfaces of MBean services earlier. Here we will look at how detached invokers are used to expose the EJB container home and bean interfaces to clients. The generic view of the invoker architecture is presented in Figure 20.3, “The transport invoker server side architecture”.

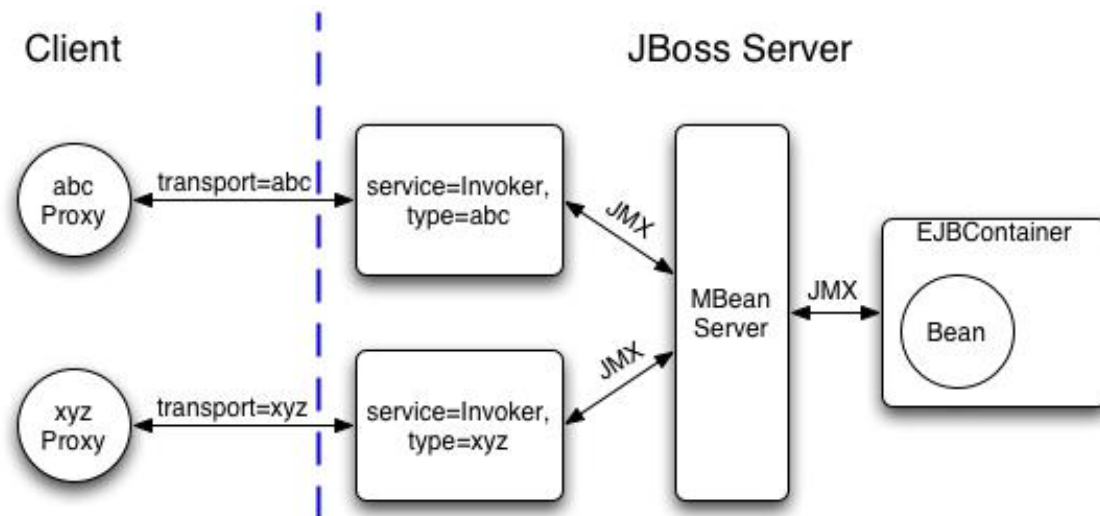


Figure 20.3. The transport invoker server side architecture

For each type of home proxy there is a binding to an invoker and its associated transport protocol. A container may have multiple invocation protocols active simultaneously. In the `jboss.xml` file, an **invoker-proxy-binding-name** maps to an **invoker-proxy-binding/name** element. At the **container-configuration** level this specifies the default invoker that will be used for EJBs deployed to the container. At the bean level, the **invoker-bindings** specify one or more invokers to use with the EJB container MBean.

When one specifies multiple invokers for a given EJB deployment, the home proxy must be given a unique JNDI binding location. This is specified by the **invoker/jndi-name** element value. Another issue when multiple invokers exist for an EJB is how to handle remote homes or interfaces obtained when the EJB calls other beans. Any such interfaces need to use the same invoker used to call the outer EJB in order for the resulting remote homes and interfaces to be compatible with the proxy the client has initiated the call through. The **invoker/ejb-ref** elements allow one to map from a protocol independent ENC **ejb-ref** to the home proxy binding for **ejb-ref** target EJB home that matches the referencing invoker type.

An example of using a custom **JRMPInvoker** MBean that enables compressed sockets for session beans can be found in the `org.jboss.test.jrmp` package of the testsuite. The following example illustrates the custom **JRMPInvoker** configuration and its mapping to a stateless session bean.

```
<server>
  <mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
  name="jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory"
  >
    <attribute name="RMIObjectPort">4445</attribute>
    <attribute name="RMIClientSocketFactory">
      org.jboss.test.jrmp.ejb.CompressionClientSocketFactory
    </attribute>
    <attribute name="RMIServerSocketFactory">
      org.jboss.test.jrmp.ejb.CompressionServerSocketFactory
```

```
        </attribute>
    </mbean>
</server>
```

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

```
<?xml version="1.0"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<!-- The jboss.xml descriptor for the jrmp-comp.jar ejb unit -->
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>StatelessSession</ejb-name>
            <configuration-name>Standard Stateless
SessionBean</configuration-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-compression-invoker
                    </invoker-proxy-binding-name>
                    <jndi-name>jrmp-compressed/StatelessSession</jndi-
name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>

    <invoker-proxy-bindings>
        <invoker-proxy-binding>
            <name>stateless-compression-invoker</name>
            <invoker-mbean>

jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory
            </invoker-mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-
factory>
            <proxy-factory-config>
                <client-interceptors>
                    <home>

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </home>
                <bean>
                    <interceptor>

org.jboss.proxy.ejb.StatelessSessionInterceptor
                    </interceptor>
```

```

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </bean>
    </client-interceptors>
  </proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>

```

The **StatelessSession** EJB **invoker-bindings** settings specify that the **stateless-compression-invoker** will be used with the home interface bound under the JNDI name **jrmc-compressed/StatelessSession**. The **stateless-compression-invoker** is linked to the custom JRMP invoker we just declared.

The following example, **org.jboss.test.hello** testsuite package, is an example of using the **HttpInvoker** to configure a stateless session bean to use the RMI/HTTP protocol.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBoss 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldViaHTTP</ejb-name>
      <jndi-name>helloworld/HelloHTTP</jndi-name>
      <invoker-bindings>
        <invoker>
          <invoker-proxy-binding-name>
            stateless-http-invoker
          </invoker-proxy-binding-name>
        </invoker>
      </invoker-bindings>
    </session>
  </enterprise-beans>
  <invoker-proxy-bindings>
    <!-- A custom invoker for RMI/HTTP -->
    <invoker-proxy-binding>
      <name>stateless-http-invoker</name>
      <invoker-mbean>jboss:service=invoker,type=http</invoker-mbean>
      <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-
factory>
      <proxy-factory-config>
        <client-interceptors>
          <home>

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

```

```
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </home>
    <bean>
        <interceptor>
            org.jboss.proxy.ejb.StatelessSessionInterceptor
        </interceptor>
    </bean>
</client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>
```

Here a custom invoker-proxy-binding named **stateless-http-invoker** is defined. It uses the **HttpInvoker** MBean as the detached invoker. The **jboss:service=invoker,type=http** name is the default name of the **HttpInvoker** MBean as found in the **http-invoker.sar/META-INF/jboss-service.xml** descriptor, and its service descriptor fragment is show here:

```
<!-- The HTTP invoker service configuration -->
<mbean code="org.jboss.invocation.http.server.HttpInvoker"
    name="jboss:service=invoker,type=http">
    <!-- Use a URL of the form
http://<hostname>:8080/invoker/EJBInvokerServlet
    where <hostname> is InetAddress.getHostname value on which the
server
    is running. -->
    <attribute name="InvokerURLPrefix">http://</attribute>
    <attribute
name="InvokerURLSuffix">:8080/invoker/EJBInvokerServlet</attribute>
    <attribute name="UseHostName">true</attribute>
</mbean>
```

The client proxy posts the EJB invocation content to the **EJBInvokerServlet** URL specified in the **HttpInvoker** service configuration.

20.2.2. The HA JRMPInvoker - Clustered RMI/JRMP Transport

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

20.2.3. The HA HttpInvoker - Clustered RMI/HTTP Transport

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

To enable HA-RMI/HTTP you need to configure the invokers for the EJB container. This is done through either a **jboss.xml** descriptor, or the **standardjboss.xml** descriptor. [Example 20.3, “A jboss.xml stateless session configuration for HA-RMI/HTTP”](#) shows is an example of a stateless session configuration taken from the **org.jboss.test.hello** testsuite package.

Example 20.3. A jboss.xml stateless session configuration for HA-RMI/HTTP

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldViaClusteredHTTP</ejb-name>
      <jndi-name>helloworld/HelloHA-HTTP</jndi-name>
      <invoker-bindings>
        <invoker>
          <invoker-proxy-binding-name>
            stateless-httpHA-invoker
          </invoker-proxy-binding-name>
        </invoker>
      </invoker-bindings>
      <clustered>true</clustered>
    </session>
  </enterprise-beans>
  <invoker-proxy-bindings>
    <invoker-proxy-binding>
      <name>stateless-httpHA-invoker</name>
      <invoker-mbean>jboss:service=invoker,type=httpHA</invoker-
mbean>
      <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-
factory>
      <proxy-factory-config>
        <client-interceptors>
          <home>

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </home>
          <bean>
            <interceptor>

org.jboss.proxy.ejb.StatelessSessionInterceptor
            </interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
```

```

        </bean>
    </client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>

```

The **stateless-httpHA-invoker** invoker-proxy-binding references the **jboss:service=invoker, type=httpHA** invoker service. This service would be configured as shown below.

```

<mbean code="org.jboss.invocation.http.server.HttpInvokerHA"
      name="jboss:service=invoker,type=httpHA">
  <!-- Use a URL of the form
        http://<hostname>:8080/invoker/EJBInvokerHAServlet
        where <hostname> is InetAddress.getHostname value on which the
server
        is running.
  -->
  <attribute name="InvokerURLPrefix">http://</attribute>
  <attribute
name="InvokerURLSuffix">:8080/invoker/EJBInvokerHAServlet</attribute>
  <attribute name="UseHostName">true</attribute>
</mbean>

```

The URL used by the invoker proxy is the **EJBInvokerHAServlet** mapping as deployed on the cluster node. The **HttpInvokerHA** instances across the cluster form a collection of candidate http URLs that are made available to the client side proxy for failover and/or load balancing.

20.3. THE EJB CONTAINER

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

20.3.1. EJBDeployer MBean

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

- **VerifyDeployments**: a boolean flag indicating if the EJB verifier should be run. This validates that the EJBs in a deployment unit conform to the EJB 2.1 specification. Setting this to true is useful for ensuring your deployments are valid.
- **VerifierVerbose**: A boolean that controls the verbosity of any verification failures/warnings that result from the verification process.
- **StrictVerifier**: A boolean that enables/disables strict verification. When strict verification is enable an EJB will deploy only if verifier reports no errors.
- **CallByValue**: a boolean flag that indicates call by value semantics should be used by default.

- **ValidateDTDs**: a boolean flag that indicates if the `ejb-jar.xml` and `jboss.xml` descriptors should be validated against their declared DTDs. Setting this to true is useful for ensuring your deployment descriptors are valid.
- **MetricsEnabled**: a boolean flag that controls whether container interceptors marked with an `metricsEnabled=true` attribute should be included in the configuration. This allows one to define a container interceptor configuration that includes metrics type interceptors that can be toggled on and off.
- **WebServiceName**: The JMX ObjectName string of the web service MBean that provides support for the dynamic class loading of EJB classes.
- **TransactionManagerServiceName**: The JMX ObjectName string of the JTA transaction manager service. This must have an attribute named **TransactionManager** that returns that `javax.transaction.TransactionManager` instance.

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

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

20.3.1.1. Verifying EJB deployments

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

Probably the most problematic aspect of writing EJBs is the fact that there is a disconnection between the bean implementation and its remote and home interfaces, as well as its deployment descriptor configuration. It is easy to have these separate elements get out of synch. One tool that helps eliminate this problem is XDoclet. It allows you to use custom JavaDoc-like tags in the EJB bean implementation class to generate the related bean interfaces, deployment descriptors and related objects. See the XDoclet home page, <http://sourceforge.net/projects/xdoclet> for additional details.

20.3.1.2. Deploying EJBs Into Containers

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

- Create subclass of `org.jboss.ejb.Container` depending on the type of the EJB: stateless,

stateful, BMP entity, CMP entity, or message driven. The container is assigned a unique **ClassLoader** from which it can load local resources. The uniqueness of the **ClassLoader** is also used to isolate the standard `java:comp` JNDI namespace from other J2EE components.

- Set all container configurable attributes from a merge of the `jboss.xml` and `standardjboss.xml` descriptors.
- Create and add the container interceptors as configured for the container.
- 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.

20.3.1.3. Container configuration information

JBoss externalizes most if not all of the setup of the EJB containers using an XML file that conforms to the `jboss_4_0.dtd`. The section DTD that relates to container configuration information is shown in [Figure 20.4, “The jboss_4_0 DTD elements related to container configuration.”](#).

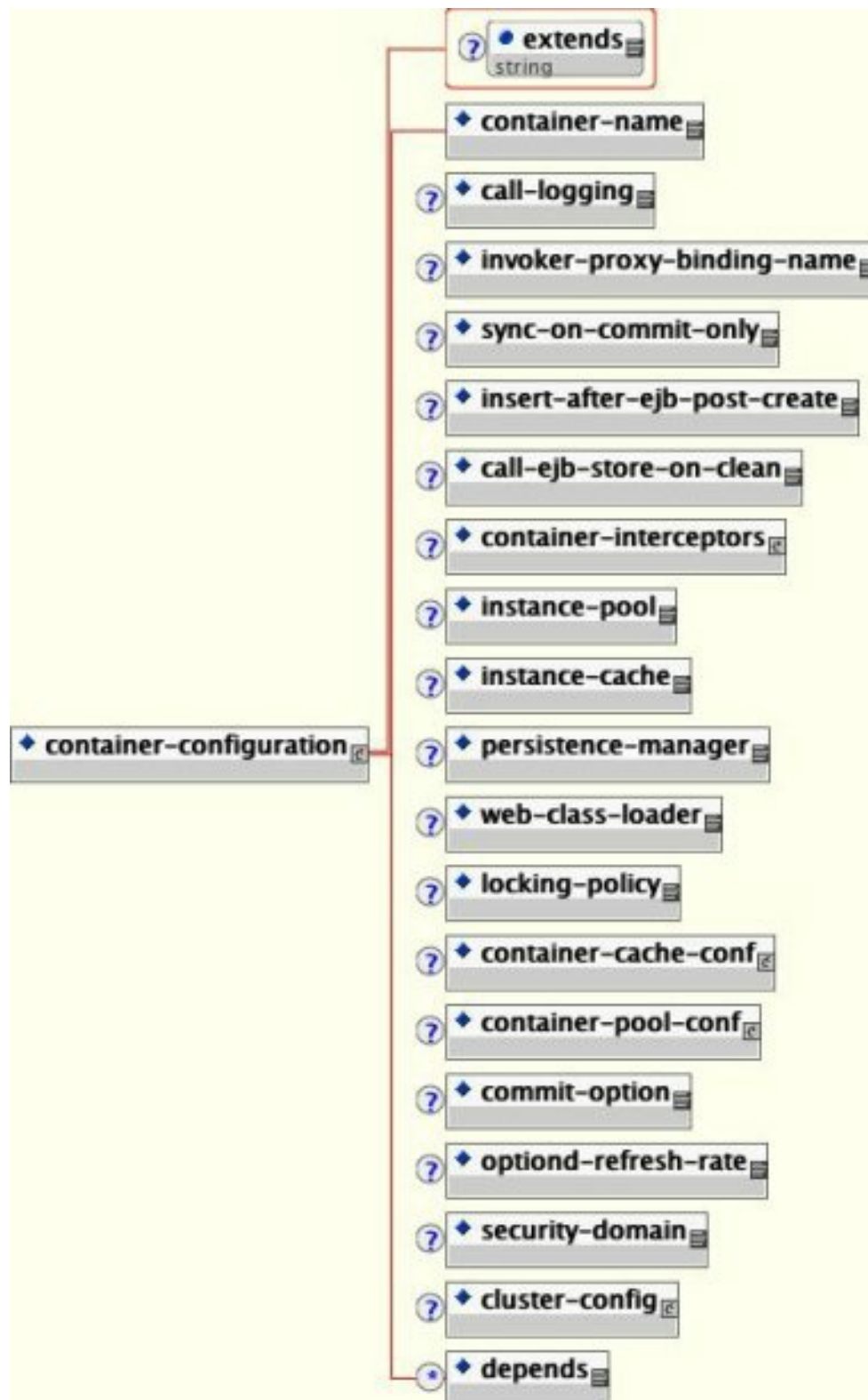


Figure 20.4. The `jboss_4_0` DTD elements related to container configuration.

The **container-configuration** 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 the default invoker type, the container interceptor makeup, instance caches/pools and their sizes, persistence manager, security, and so on. Because this is a large amount of information that requires a detailed understanding of the JBoss container architecture, JBoss ships with a standard configuration for the four types of EJBs. This configuration file is called **standardjboss.xml** and it is located in the `conf` directory of any configuration file set that uses EJBs. The following is a sample of **container-configuration** from **standardjboss.xml**.

```

<container-configuration>
  <container-name>Standard CMP 2.x EntityBean</container-name>
  <call-logging>false</call-logging>
  <invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-
name>
  <sync-on-commit-only>false</sync-on-commit-only>
  <insert-after-ejb-post-create>false</insert-after-ejb-post-create>
  <call-ejb-store-on-clean>true</call-ejb-store-on-clean>
  <container-interceptors>

<interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</intercep
tor>
  <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
  <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>

<interceptor>org.jboss.ejb.plugins.CallValidationInterceptor</interceptor>
  <interceptor metricsEnabled="true">
    org.jboss.ejb.plugins.MetricsInterceptor
  </interceptor>

<interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</intercepto
r>
  <interceptor>
org.jboss.resource.connectionmanager.CachedConnectionInterceptor
  </interceptor>

<interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</inter
ceptor>

<interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</inter
ceptor>
  </container-interceptors>
  <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-
pool>
  <instance-
cache>org.jboss.ejb.plugins.InvalidableEntityInstanceCache</instance-
cache>
  <persistence-
manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-
manager>
  <locking-
policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</locking-
policy>
  <container-cache-conf>
    <cache-
policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-
policy>

```

```

    <cache-policy-conf>
      <min-capacity>50</min-capacity>
      <max-capacity>1000000</max-capacity>
      <overager-period>300</overager-period>
      <max-bean-age>600</max-bean-age>
      <resizer-period>400</resizer-period>
      <max-cache-miss-period>60</max-cache-miss-period>
      <min-cache-miss-period>1</min-cache-miss-period>
      <cache-load-factor>0.75</cache-load-factor>
    </cache-policy-conf>
  </container-cache-conf>
  <container-pool-conf>
    <MaximumSize>100</MaximumSize>
  </container-pool-conf>
  <commit-option>B</commit-option>
</container-configuration>

```

These two examples 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 file set directory. The second is at the EJB JAR level. By placing a **jboss.xml** file in the EJB JAR **META-INF** directory, you can specify either overrides for container configurations in the **standardjboss.xml** file, or entirely new named container configurations. This provides great flexibility in the configuration of containers. As you have seen, all container configuration attributes have been externalized and as such are easily modifiable. Knowledgeable developers can even implement specialized container components, such as instance pools or caches, and easily integrate them with the standard container configurations to optimize behavior for a particular application or environment.

How an EJB deployment chooses its container configuration is based on the explicit or implicit **jboss/enterprise-beans/<type>/configuration-name** element. The **configuration-name** element is a link to a **container-configurations/container-configuration** element. It specifies which container configuration to use for the referring EJB. The link is from a **configuration-name** element to a **container-name** element.

You are able to specify container configurations per class of EJB by including a **container-configuration** element in the EJB definition. Typically one does not define completely new container configurations, although this is supported. The typical usage of a **jboss.xml** level **container-configuration** is to override one or more aspects of a **container-configuration** coming from the **standardjboss.xml** descriptor. This is done by specifying **container-configuration** that references the name of an existing **standardjboss.xml** **container-configuration/container-name** as the value for the **container-configuration/extends** attribute. The following example shows an example of defining a new **Secured Stateless SessionBean** configuration that is an extension of the **Standard Stateless SessionBean** configuration.

```

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

```

```

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

```

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

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

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

Now that you know how to specify which container configuration an EJB is using and can define a deployment unit level override, we now will look at the **container-configuration** child elements in the following sections. A number of the elements specify interface class implementations whose configuration is affected by other elements, so before starting in on the configuration elements you need to understand the **org.jboss.metadata.XmlLoadable** interface.

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

```

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

```

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

20.3.1.3.1. 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.

20.3.1.3.2. The call-logging element

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

20.3.1.3.3. The invoker-proxy-binding-name element

The **invoker-proxy-binding-name** element specifies the name of the default invoker to use. In the absence of a bean level **invoker-bindings** specification, the **invoker-proxy-binding** whose name matches the **invoker-proxy-binding-name** element value will be used to create home and remote proxies.

20.3.1.3.4. The sync-on-commit-only element

This configures a performance optimization that will cause entity bean state to be synchronized with the database only at commit time. Normally the state of all the beans in a transaction would need to be synchronized when an finder method is called or when an remove method is called, for example.

20.3.1.3.5. insert-after-ejb-post-create

This is another entity bean optimization which cause the database insert command for a new entity bean to be delayed until the **ejbPostCreate** method is called. This allows normal CMP fields as well as CMR fields to be set in a single insert, instead of the default insert followed by an update, which allows removes the requirement for relation ship fields to allow null values.

20.3.1.3.6. call-ejb-store-on-clean

By the specification the container is required to call **ejbStore** method on an entity bean instance when transaction commits even if the instance was not modified in the transaction. Setting this to false will cause JBoss to only call **ejbStore** for dirty objects.

20.3.1.3.7. 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** structure through which EJB method invocations pass. The first interceptor in the chain is invoked when the **MBeanServer** passes a method invocation to the container. The last interceptor invokes the business method on the bean. We will discuss the **Interceptor** interface latter in this chapter when we talk about the container plugin framework. Generally, care must be taken when changing an existing standard EJB interceptor configuration as the EJB contract regarding security, transactions, persistence, and thread safety derive from the interceptors.

20.3.1.3.8. The instance-pool element

The **instance-pool** element specifies the fully qualified class name of an **org.jboss.ejb.InstancePool** interface implementation to use as the container **InstancePool**. We will discuss the InstancePool interface in detail latter in this chapter when we talk about the container plugin framework.

20.3.1.3.9. The container-pool-conf element

The **container-pool-conf** is passed to the **InstancePool** implementation class given by the

instance-pool element if it implements the **XmlLoadable** interface. All current JBoss **InstancePool** implementations derive from the **org.jboss.ejb.plugins.AbstractInstancePool** class which provides support for elements shown in Figure 20.5, “The container-pool-conf element DTD”.

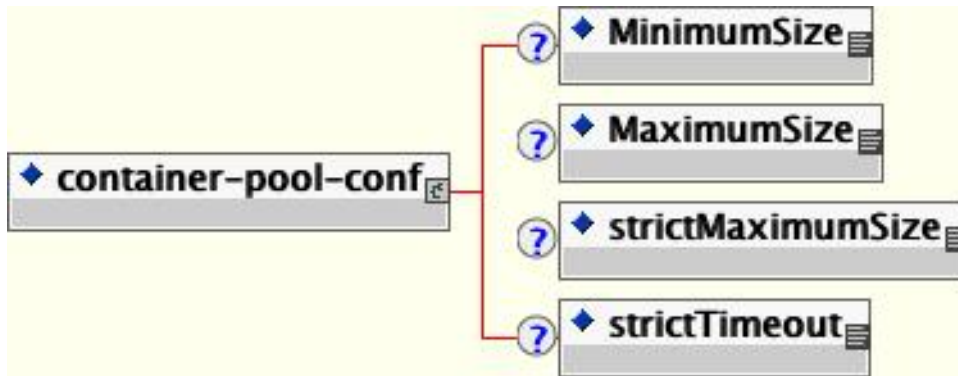


Figure 20.5. The container-pool-conf element DTD

- **MinimumSize**: The **MinimumSize** element gives the minimum number of instances to keep in the pool, although JBoss does not currently seed an **InstancePool** to the **MinimumSize** value.
- **MaximumSize**: 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.
- **strictMaximumSize**: 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.
- **strictTimeout**: How long a request blocks waiting for an instance pool object is controlled by the **strictTimeout** element. The **strictTimeout** 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 a **Long** so the maximum possible wait time is 9,223,372,036,854,775,807 or about 292,471,208 years, and this is the default value.

20.3.1.3.10. The instance-cache element

The **instance-cache** element specifies the fully qualified class name of the **org.jboss.ejb.InstanceCache** interface implementation. This element is only meaningful for entity and stateful session beans as these are the only EJB types that have an associated identity. We will discuss the **InstanceCache** interface in detail latter in this chapter when we talk about the container plugin framework.

20.3.1.3.11. The container-cache-conf element

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 which 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 is used 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 `standardjboss.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, shown in [Figure 20.6, “The container-cache-conf element DTD”](#).

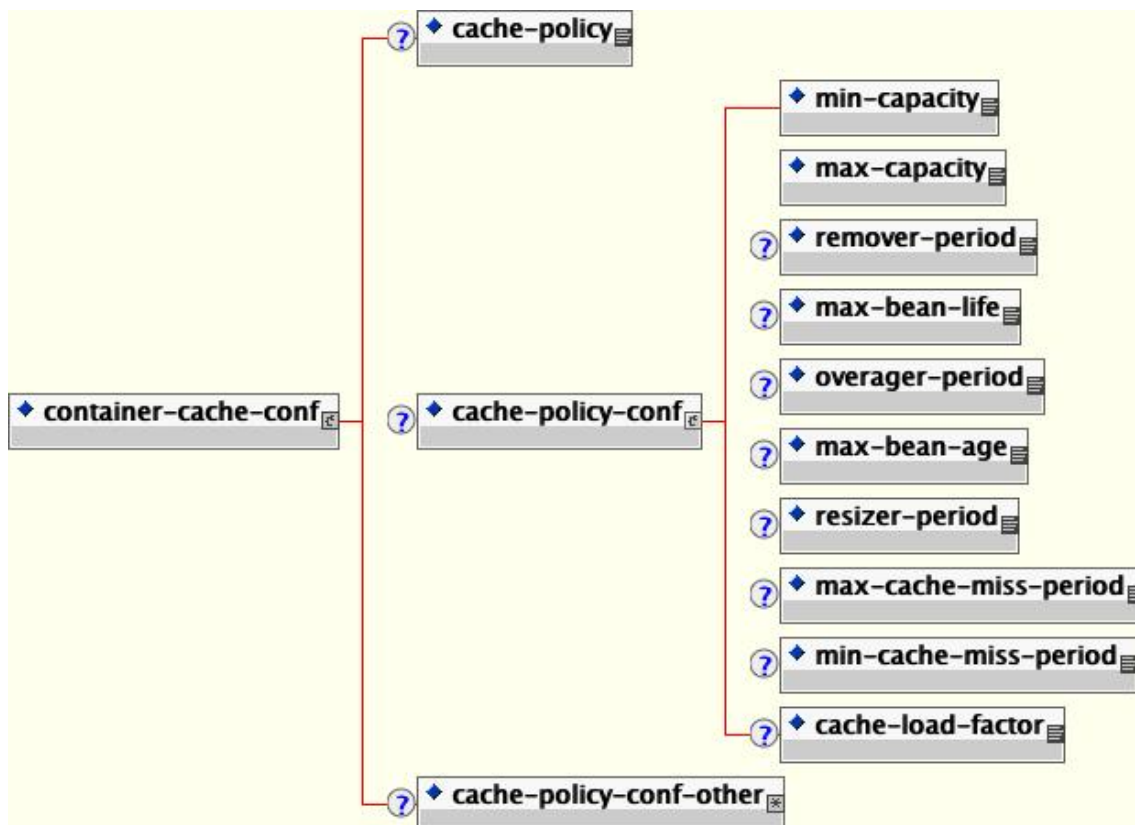


Figure 20.6. The container-cache-conf element DTD

- **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 **current-capacity * 1/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 in seconds that a bean can exist inactive. After this period, as a result, the bean will be 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.

20.3.1.3.12. 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.

20.3.1.3.13. The web-class-loader Element

The **web-class-loader** element specifies a subclass of **org.jboss.web.WebClassLoader** that is used in conjunction with the **WebService** MBean to allow dynamic loading of resources and classes from deployed ears, EJB JARs and WARs. A **WebClassLoader** is associated with a **Container** and must have an **org.jboss.mx.loading.UnifiedClassLoader** as its parent. It overrides the **getURLs()** method to return a different set of URLs for remote loading than what is used for local loading.

WebClassLoader has two methods meant to be overridden by subclasses: **getKey()** and **getBytes()**. The latter is a no-op in this implementation and should be overridden by subclasses with bytecode generation ability, such as the classloader used by the iiop module.

WebClassLoader subclasses must have a constructor with the same signature as the **WebClassLoader(ObjectName containerName, UnifiedClassLoader parent)** constructor.

20.3.1.3.14. 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.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. This class pops the next waiting transaction from the queue and notifies only those threads waiting associated with that transaction. The **QueuedPessimisticEJBLock** is the current default used by the standard configurations.
- **org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLockNoADE**: This behaves the same as the **QueuedPessimisticEJBLock** except that deadlock detection is disabled.
- **org.jboss.ejb.plugins.lock.SimpleReadWriteEJBLock**: This lock allows multiple read locks concurrently. Once a writer has requested the lock, future read-lock requests whose transactions do not already have the read lock will block until all writers are done; then all the waiting readers will concurrently go (depending on the `reentrant` setting / `methodLock`). A reader who promotes gets first crack at the write lock, ahead of other waiting writers. If there is already a reader that is promoting, we throw an inconsistent read exception. Of course, writers have to wait for all read-locks to release before taking the write lock.
- **org.jboss.ejb.plugins.lock.NoLock**: an anti-locking policy used with the instance per transaction container configurations.

Locking and deadlock detection will be discussed in more detail in [Section 20.4, “Entity Bean Locking and Deadlock Detection”](#).

20.3.1.3.15. The commit-option and optiond-refresh-rate elements

The **commit-option** value specifies the EJB entity bean persistent storage commit option. It must be one of **A**, **B**, **C** or **D**.

- **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 commit option 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.

20.3.1.3.16. The security-domain element

The **security-domain** element specifies the JNDI name of the object that implements the **org.jboss.security.AuthenticationManager** and **org.jboss.security.RealmMapping** interfaces. It is more typical to specify the **security-domain** under the **jboss** root element so that all EJBs in a given deployment are secured in the same manner. However, it is possible to configure the security domain for each bean configuration. The details of the security manager interfaces and configuring the security layer are discussed in [Chapter 10, Security on JBoss](#).

20.3.1.3.17. cluster-config

The **cluster-config** element allows to specify cluster specific settings for all EJBs that use the container configuration. Specification of the cluster configuration may be done at the container configuration level or at the individual EJB deployment level.

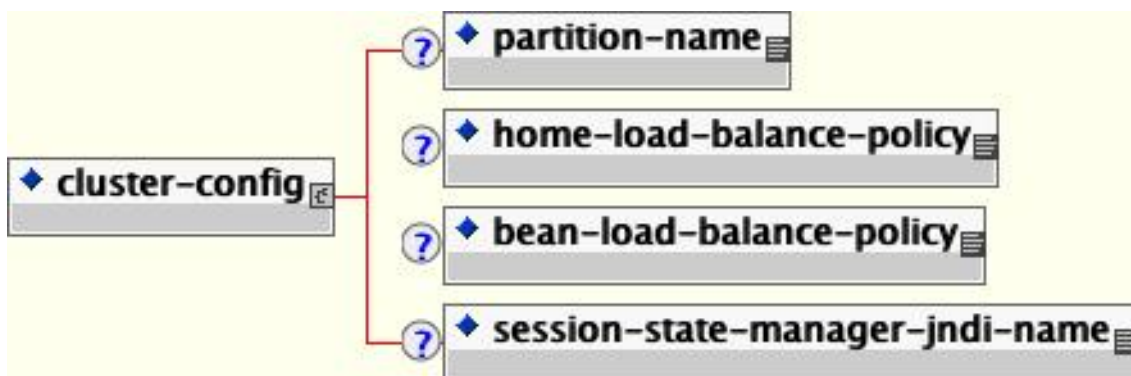


Figure 20.7. The cluster-config and related elements

- **partition-name**: The **partition-name** element indicates where to find the **org.jboss.ha.framework.interfaces.HAPartition** interface to be used by the container to exchange clustering information. This is not the full JNDI name under which **HAPartition** is bound. Rather, it should correspond to the **PartitionName** attribute of the **ClusterPartitionMBean** service that is managing the desired cluster. The actual JNDI name of the **HAPartition** binding will be formed by appending **/HASessionState/** to the partition-name value. The default value is **DefaultPartition**.
- **home-load-balance-policy**: The **home-load-balance-policy** element indicates the Java class name to be used to load balance calls made on the home proxy. The class must implement the **org.jboss.ha.framework.interface.LoadBalancePolicy** interface. The default policy is **org.jboss.ha.framework.interfaces.RoundRobin**.
- **bean-load-balance-policy**: The **bean-load-balance-policy** element indicates the java class name to be used to load balance calls in the bean proxy. The class must implement the **org.jboss.ha.framework.interface.LoadBalancePolicy** interface. For entity beans

and stateful session beans, the default is

org.jboss.ha.framework.interfaces.FirstAvailable. For stateless session beans, **org.jboss.ha.framework.interfaces.RoundRobin**.

- **session-state-manager-jndi-name**: The **session-state-manager-jndi-name** element indicates the name of the **org.jboss.ha.framework.interfaces.HASessionState** to be used by the container as a backend for state session management in the cluster. Unlike the **partition-name** element, this is a JNDI name under which the **HASessionState** implementation is bound. The default location used is **/HASessionState/Default**.

20.3.1.3.18. The depends element

The **depends** element gives a JMX **ObjectName** of a service on which the container or EJB depends. Specification of explicit dependencies on other services avoids having to rely on the deployment order being after the required services are started.

20.3.2. Container Plug-in Framework

The JBoss EJB container uses a framework pattern that allows one to change implementations of various aspects of the container behavior. The container itself does not perform any significant work other than connecting the various behavioral components together. Implementations of the behavioral components are referred to as plugins, because you can plug in a new implementation by changing a container configuration. Examples of plug-in behavior you may want to change include persistence management, object pooling, object caching, container invokers and interceptors. There are four subclasses of the **org.jboss.ejb.Container** class, each one implementing a particular bean type:

- **org.jboss.ejb.EntityContainer**: handles **javax.ejb.EntityBean** types
- **org.jboss.ejb.StatelessSessionContainer**: handles Stateless **javax.ejb.SessionBean** types
- **org.jboss.ejb.StatefulSessionContainer**: handles Stateful **javax.ejb.SessionBean** types
- **org.jboss.ejb.MessageDrivenContainer** handles **javax.ejb.MessageDrivenBean** types

The EJB containers delegate much of their behavior to components known as container plug-ins. The interfaces that make up the container plugin points include the following:

- **org.jboss.ejb.ContainerPlugin**
- **org.jboss.ejb.ContainerInvoker**
- **org.jboss.ejb.Interceptor**
- **org.jboss.ejb.InstancePool**
- **org.jboss.ejb.InstanceCache**
- **org.jboss.ejb.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.

20.3.2.1. 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 below.

Example 20.4. The org.jboss.ejb.ContainerPlugin interface

```
public interface ContainerPlugin
    extends Service, AllowedOperationsFlags
{
    /** co
     * 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);
}
```

20.3.2.2. 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 below.

Example 20.5. The org.jboss.ejb.Interceptor interface

```
import org.jboss.invocation.Invocation;

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

All interceptors defined in the container configuration are created and added to the container interceptor chain by the **EJBDeployer**. The last interceptor is not added by the deployer but rather by the container itself because this is the interceptor that interacts with the EJB bean implementation.

The order of the interceptor in the chain is important. The idea behind ordering is that interceptors that are not tied to a particular **EnterpriseContext** instance are positioned before interceptors that interact with caches and pools.

Implementers of the **Interceptor** interface form a linked-list like structure through which the **Invocation** object is passed. The first interceptor in the chain is invoked when an invoker passes a **Invocation** to the container via the JMX bus. The last interceptor invokes the business method on the bean. There are usually on the order of five interceptors in a chain depending on the bean type and

container configuration. **Interceptor** semantic complexity ranges from simple to complex. An example of a simple interceptor would be **LoggingInterceptor**, while a complex example is **EntitySynchronizationInterceptor**.

One of the main advantages of an interceptor pattern is flexibility in the arrangement of interceptors. Another advantage is the clear functional distinction between different interceptors. For example, logic for transaction and security is cleanly separated between the **TXInterceptor** and **SecurityInterceptor** respectively.

If any of the interceptors fail, the call is terminated at that point. This is a fail-quickly type of semantic. For example, if a secured EJB is accessed without proper permissions, the call will fail as the **SecurityInterceptor** before any transactions are started or instances caches are updated.

20.3.2.3. 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.

Example 20.6. 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).

20.3.2.4. 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.

Example 20.7. 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**.

20.3.2.5. 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

Example 20.8. The org.jboss.ejb.EntityPersistenceManager interface

```

public interface EntityPersistenceManager
    extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the
     * bean class.
     *
     * @return the new instance
     */
    Object createBeanClassInstance() throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbCreate
method
     * on the instance and to handle the results properly wrt the
persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void createEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbPostCreate
method
     * on the instance and to handle the results properly wrt the
persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void postCreateEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when single entities are to be found. The
     * persistence manager must find out whether the wanted instance is
     * available in the persistence store, and if so it shall use the
     * ContainerInvoker plugin to create an EJBObject to the instance,
which
     * is to be returned as result.

```



```

*
* @param finderMethod the find method in the home interface that
was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an EJBObject representing the found entity
*/
Object findEntity(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
throws Exception;

/**
* This method is called when collections of entities are to be
* found. The persistence manager must find out whether the wanted
* instances are available in the persistence store, and if so it
* shall use the ContainerInvoker plugin to create EJBObjects to
* the instances, which are to be returned as result.
*
* @param finderMethod the find method in the home interface that
was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an EJBObject collection representing the found
* entities
*/
Collection findEntities(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when an entity shall be activated. The
* persistence manager must call the ejbActivate method on the
* instance.
*
* @param instance the instance to use for the activation
*
* @throws RemoteException thrown if some system exception occurs
*/
void activateEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
* This method is called whenever an entity shall be load from the
* underlying storage. The persistence manager must load the state
* from the underlying storage and then call ejbLoad on the
* supplied instance.
*
* @param instance the instance to synchronize
*
* @throws RemoteException thrown if some system exception occurs
*/
void loadEntity(EntityEnterpriseContext instance)

```

```
throws RemoteException;

/**
 * This method is used to determine if an entity should be stored.
 *
 * @param instance the instance to check
 * @return true, if the entity has been modified
 * @throws Exception thrown if some system exception occurs
 */
boolean isModified(EntityEnterpriseContext instance) throws
Exception;

/**
 * This method is called whenever an entity shall be stored to the
 * underlying storage. The persistence manager must call ejbStore
 * on the supplied instance and then store the state to the
 * underlying storage.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void storeEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is called when an entity shall be passivate. The
 * persistence manager must call the ejbPassivate method on the
 * instance.
 *
 * @param instance the instance to passivate
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void passivateEntity(EntityEnterpriseContext instance)
throws RemoteException;

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

20.3.2.6. The org.jboss.ejb.EntityPersistenceStore interface

As per the EJB 2.1 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.cmp.jdbc.JDBCStoreManager** which is the entry point for the CMP2 persistence engine. The **EntityPersistenceStore** interface is shown below.

Example 20.9. The org.jboss.ejb.EntityPersistenceStore interface

```
public interface EntityPersistenceStore
    extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the
     * bean class.
     *
     * @return the new instance
     *
     * @throws Exception
     */
    Object createBeanClassInstance()
        throws Exception;

    /**
     * Initializes the instance context.
     *
     * <p>This method is called before createEntity, and should
     * reset the value of all cmpFields to 0 or null.
     *
     * @param ctx
     *
     * @throws RemoteException
     */
    void initEntity(EntityEnterpriseContext ctx);

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for handling the results
     * properly wrt the persistent store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     * @return The primary key computed by CMP PM or null for BMP
     *
     * @throws Exception
     */
    Object createEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when single entities are to be found. The
     * persistence manager must find out whether the wanted instance
     * is available in the persistence store, if so it returns the
```

```

    * primary key of the object.
    *
    * @param finderMethod the find method in the home interface that
was    * called
    * @param args any finder parameters
    * @param instance the instance to use for the finder call
    * @return a primary key representing the found entity
    *
    * @throws RemoteException thrown if some system exception occurs
    * @throws FinderException thrown if some heuristic problem occurs
    */
    Object findEntity(Method finderMethod,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
    * This method is called when collections of entities are to be
    * found. The persistence manager must find out whether the wanted
    * instances are available in the persistence store, and if so it
    * must return a collection of primaryKeys.
    *
    * @param finderMethod the find method in the home interface that
was    * called
    * @param args any finder parameters
    * @param instance the instance to use for the finder call
    * @return an primary key collection representing the found
    * entities
    *
    * @throws RemoteException thrown if some system exception occurs
    * @throws FinderException thrown if some heuristic problem occurs
    */
    Collection findEntities(Method finderMethod,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
    * This method is called when an entity shall be activated.
    *
    * <p>With the PersistenceManager factorization most EJB
    * calls should not exists However this calls permits us to
    * introduce optimizations in the persistence store. Particularly
    * the context has a "PersistenceContext" that a PersistenceStore
    * can use (JAWS does for smart updates) and this is as good a
    * callback as any other to set it up.
    * @param instance the instance to use for the activation
    *
    * @throws RemoteException thrown if some system exception occurs
    */
    void activateEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**

```

```

* This method is called whenever an entity shall be load from the
* underlying storage. The persistence manager must load the state
* from the underlying storage and then call ejbLoad on the
* supplied instance.
*
* @param instance the instance to synchronize
*
* @throws RemoteException thrown if some system exception occurs
*/
void loadEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
* This method is used to determine if an entity should be stored.
*
* @param instance the instance to check
* @return true, if the entity has been modified
* @throws Exception thrown if some system exception occurs
*/
boolean isModified(EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called whenever an entity shall be stored to the
* underlying storage. The persistence manager must call ejbStore
* on the supplied instance and then store the state to the
* underlying storage.
*
* @param instance the instance to synchronize
*
* @throws RemoteException thrown if some system exception occurs
*/
void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
* This method is called when an entity shall be passivate. The
* persistence manager must call the ejbPassivate method on the
* instance.
*
* <p>See the activate discussion for the reason for
* exposing EJB callback * calls to the store.
*
* @param instance the instance to passivate
*
* @throws RemoteException thrown if some system exception occurs
*/
void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
* This method is called when an entity shall be removed from the
* underlying storage. The persistence manager must call ejbRemove
* on the instance and then remove its state from the underlying
* storage.
*

```

```

        * @param instance the instance to remove
        *
        * @throws RemoteException thrown if some system exception occurs
        * @throws RemoveException thrown if the instance could not be
removed
        */
        void removeEntity(EntityEnterpriseContext instance)
            throws RemoteException, RemoveException;
    }

```

The default BMP implementation of the **EntityPersistenceManager** interface is **org.jboss.ejb.plugins.BMPPersistenceManager**. The BMP persistence manager is fairly simple since all persistence logic is in the entity bean itself. The only duty of the persistence manager is to perform container callbacks.

20.3.2.7. 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

The **StatefulSessionPersistenceManager** interface is shown below.

Example 20.10. 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 session beans. More specifically, the persistence manager serializes beans in a flat file whose name is composed of the bean name and session id with a `.ser` extension. The persistence manager restores a bean's state during activation and respectively stores its state during passivation from the bean's `.ser` file.

20.4. ENTITY BEAN LOCKING AND DEADLOCK DETECTION

This section provides information on what entity bean locking is and how entity beans are accessed and locked within JBoss. It also describes the problems you may encounter as you use entity beans within your system and how to combat these issues. Deadlocking is formally defined and examined. And, finally, we walk you through how to fine tune your system in terms of entity bean locking.

20.4.1. Why JBoss Needs Locking

Locking is about protecting the integrity of your data. Sometimes you need to be sure that only one user can update critical data at one time. Sometimes, access to sensitive objects in your system need to be serialized so that data is not corrupted by concurrent reads and writes. Databases traditionally provide this sort of functionality with transactional scopes and table and row locking facilities.

Entity beans are a great way to provide an object-oriented interface to relational data. Beyond that, they can improve performance by taking the load off of the database through caching and delaying updates until absolutely needed so that the database efficiency can be maximized. But, with caching, data integrity is a problem, so some form of application server level locking is needed for entity beans to provide the transaction isolation properties that you are used to with traditional databases.

20.4.2. Entity Bean Lifecycle

With the default configuration of JBoss there is only one active instance of a given entity bean in memory at one time. This applies for every cache configuration and every type of **commit-option**. The lifecycle for this instance is different for every commit-option though.

- For commit option *A*, this instance is cached and used between transactions.
- For commit option *B*, this instance is cached and used between transactions, but is marked as dirty at the end of a transaction. This means that at the start of a new transaction **ejbLoad** must be called.
- For commit option *C*, this instance is marked as dirty, released from the cache, and marked for passivation at the end of a transaction.
- For commit option *D*, a background refresh thread periodically calls **ejbLoad** on stale beans within the cache. Otherwise, this option works in the same way as *A*.

When a bean is marked for passivation, the bean is placed in a passivation queue. Each entity bean container has a passivation thread that periodically passivates beans that have been placed in the passivation queue. A bean is pulled out of the passivation queue and reused if the application requests access to a bean of the same primary key.

On an exception or transaction rollback, the entity bean instance is thrown out of cache entirely. It is not put into the passivation queue and is not reused by an instance pool. Except for the passivation queue, there is no entity bean instance pooling.

20.4.3. Default Locking Behavior

Entity bean locking is totally decoupled from the entity bean instance. The logic for locking is totally isolated and managed in a separate lock object. Because there is only one allowed instance of a given entity bean active at one time, JBoss employs two types of locks to ensure data integrity and to conform to the EJB spec.

- **Method Lock:** The method lock ensures that only one thread of execution at a time can invoke on a given Entity Bean. This is required by the EJB spec.
- **Transaction Lock:** A transaction lock ensures that only one transaction at a time has access to a give Entity Bean. This ensures the ACID properties of transactions at the application server level. Since, by default, there is only one active instance of any given Entity Bean at one time, JBoss must protect this instance from dirty reads and dirty writes. So, the default entity bean locking behavior will lock an entity bean within a transaction until it completes. This means that if any method at all is invoked on an entity bean within a transaction, no other transaction can have access to this bean until the holding transaction commits or is rolled back.

20.4.4. Pluggable Interceptors and Locking Policy

We saw that the basic entity bean lifecycle and behavior is defined by the container configuration defined in **standardjboss.xml** descriptor. Let's look at the **container-interceptors** definition for the *Standard CMP 2.x EntityBean* configuration.

```
<container-interceptors>

<interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</intercep
tor>
    <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>

<interceptor>org.jboss.ejb.plugins.CallValidationInterceptor</interceptor>
    <interceptor
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</intercepto
r>

<interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</intercepto
r>

<interceptor>org.jboss.resource.connectionmanager.CachedConnectionIntercep
tor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</inter
ceptor>

<interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</inter
ceptor>
</container-interceptors>
```


The interceptors shown above define most of the behavior of the entity bean. Below is an explanation of the interceptors that are relevant to this section.

- **EntityLockInterceptor:** This interceptor's role is to schedule any locks that must be acquired before the invocation is allowed to proceed. This interceptor is very lightweight and delegates all locking behavior to a pluggable locking policy.
- **EntityInstanceInterceptor:** The job of this interceptor is to find the entity bean within the cache or create a new one. This interceptor also ensures that there is only one active instance of a bean in memory at one time.
- **EntitySynchronizationInterceptor:** The role of this interceptor is to synchronize the state of the cache with the underlying storage. It does this with the **ejbLoad** and **ejbStore** semantics of the EJB specification. In the presence of a transaction this is triggered by transaction demarcation. It registers a callback with the underlying transaction monitor through the JTA interfaces. If there is no transaction the policy is to store state upon returning from invocation. The synchronization policies *A*, *B* and *C* of the specification are taken care of here as well as the JBoss specific commit-option *D*.

20.4.5. Deadlock

Finding deadlock problems and resolving them is the topic of this section. We will describe what deadlocking MBeans, how you can detect it within your application, and how you can resolve deadlocks. Deadlock can occur when two or more threads have locks on shared resources. [Figure 20.8, “Deadlock definition example”](#) illustrates a simple deadlock scenario. Here, **Thread 1** has the lock for **Bean A**, and **Thread 2** has the lock for **Bean B**. At a later time, **Thread 1** tries to lock **Bean B** and blocks because **Thread 2** has it. Likewise, as **Thread 2** tries to lock **A** it also blocks because **Thread 1** has the lock. At this point both threads are deadlocked waiting for access to the resource already locked by the other thread.

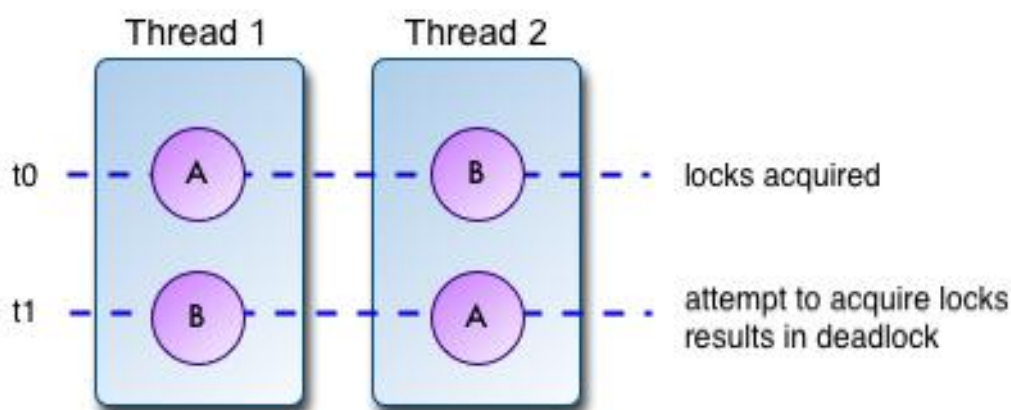


Figure 20.8. Deadlock definition example

The default locking policy of JBoss is to lock an Entity bean when an invocation occurs in the context of a transaction until the transaction completes. Because of this, it is very easy to encounter deadlock if you have long running transactions that access many entity beans, or if you are not careful about ordering the access to them. Various techniques and advanced configurations can be used to avoid deadlocking problems. They are discussed later in this section.

20.4.5.1. Deadlock Detection

Fortunately, JBoss is able to perform deadlock detection. JBoss holds a global internal graph of waiting transactions and what transactions they are blocking on. Whenever a thread determines that it cannot acquire an entity bean lock, it figures out what transaction currently holds the lock on the bean and add

itself to the blocked transaction graph. An example of what the graph may look like is given in [Table 20.1, “An example blocked transaction table”](#).

Table 20.1. An example blocked transaction table

Blocking TX	Tx that holds needed lock
Tx1	Tx2
Tx3	Tx4
Tx4	Tx1

Before the thread actually blocks it tries to detect whether there is deadlock problem. It does this by traversing the block transaction graph. As it traverses the graph, it keeps track of what transactions are blocked. If it sees a blocked node more than once in the graph, then it knows there is deadlock and will throw an **ApplicationDeadlockException**. This exception will cause a transaction rollback which will cause all locks that transaction holds to be released.

20.4.5.2. Catching ApplicationDeadlockException

Since JBoss can detect application deadlock, you should write your application so that it can retry a transaction if the invocation fails because of the **ApplicationDeadlockException**. Unfortunately, this exception can be deeply embedded within a **RemoteException**, so you have to search for it in your catch block. For example:

```
try {
    // ...
} catch (RemoteException ex) {
    Throwable cause = null;
    RemoteException rex = ex;
    while (rex.detail != null) {
        cause = rex.detail;
        if (cause instanceof ApplicationDeadlockException) {
            // ... We have deadlock, force a retry of the transaction.
            break;
        }
        if (cause instanceof RemoteException) {
            rex = (RemoteException)cause;
        }
    }
}
```

20.4.5.3. Viewing Lock Information

The **EntityLockMonitor** MBean service allows one to view basic locking statistics as well as printing out the state of the transaction locking table. To enable this monitor uncomment its configuration in the **conf/jboss-service.xml**:

```
<mbean code="org.jboss.monitor.EntityLockMonitor"
      name="jboss.monitor:name=EntityLockMonitor"/>
```

The **EntityLockMonitor** has no configurable attributes. It does have the following read-only attributes:

- **MedianWaitTime**: The median value of all times threads had to wait to acquire a lock.
- **AverageContentenders**: The ratio of the total number of contentions to the sum of all threads that had to wait for a lock.
- **TotalContentions**: The total number of threads that had to wait to acquire the transaction lock. This happens when a thread attempts to acquire a lock that is associated with another transaction
- **MaxContentenders**: The maximum number of threads that were waiting to acquire the transaction lock.

It also has the following operations:

- **clearMonitor**: This operation resets the lock monitor state by zeroing all counters.
- **printLockMonitor**: This operation prints out a table of all EJB locks that lists the **ejbName** of the bean, the total time spent waiting for the lock, the count of times the lock was waited on and the number of transactions that timed out waiting for the lock.

20.4.6. Advanced Configurations and Optimizations

The default locking behavior of entity beans can cause deadlock. Since access to an entity bean locks the bean into the transaction, this also can present a huge performance/throughput problem for your application. This section walks through various techniques and configurations that you can use to optimize performance and reduce the possibility of deadlock.

20.4.6.1. Short-lived Transactions

Make your transactions as short-lived and fine-grained as possible. The shorter the transaction you have, the less likelihood you will have concurrent access collisions and your application throughput will go up.

20.4.6.2. Ordered Access

Ordering the access to your entity beans can help lessen the likelihood of deadlock. This means making sure that the entity beans in your system are always accessed in the same exact order. In most cases, user applications are just too complicated to use this approach and more advanced configurations are needed.

20.4.6.3. Read-Only Beans

Entity beans can be marked as read-only. When a bean is marked as read-only, it never takes part in a transaction. This means that it is never transactionally locked. Using commit-option *D* with this option is sometimes very useful when your read-only bean's data is sometimes updated by an external source.

To mark a bean as read-only, use the **read-only** flag in the **jboss.xml** deployment descriptor.

Example 20.11. Marking an entity bean read-only using jboss.xml

```
<jboss>
  <enterprise-beans>
```

```

        <entity>
            <ejb-name>MyEntityBean</ejb-name>
            <jndi-name>MyEntityHomeRemote</jndi-name>
            <read-only>True</read-only>
        </entity>
    </enterprise-beans>
</jboss>

```

20.4.6.4. Explicitly Defining Read-Only Methods

After reading and understanding the default locking behavior of entity beans, you're probably wondering, "Why lock the bean if its not modifying the data?" JBoss allows you to define what methods on your entity bean are read only so that it will not lock the bean within the transaction if only these types of methods are called. You can define these read only methods within a **jboss.xml** deployment descriptor. Wildcards are allowed for method names. The following is an example of declaring all getter methods and the **anotherReadOnlyMethod** as read-only.

Example 20.12. Defining entity bean methods as read only

```

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>nextgen.EnterpriseEntity</ejb-name>
      <jndi-name>nextgen.EnterpriseEntity</jndi-name>
      <method-attributes>
        <method>
          <method-name>get*</method-name>
          <read-only>true</read-only>
        </method>
        <method>
          <method-name>anotherReadOnlyMethod</method-name>
          <read-only>true</read-only>
        </method>
      </method-attributes>
    </entity>
  </enterprise-beans>
</jboss>

```

20.4.6.5. Instance Per Transaction Policy

The Instance Per Transaction policy is an advanced configuration that can totally wipe away deadlock and throughput problems caused by JBoss's default locking policy. The default Entity Bean locking policy is to only allow one active instance of a bean. The Instance Per Transaction policy breaks this requirement by allocating a new instance of a bean per transaction and dropping this instance at the end of the transaction. Because each transaction has its own copy of the bean, there is no need for transaction based locking.

This option does sound great but does have some drawbacks right now. First, the transactional isolation behavior of this option is equivalent to **READ_COMMITTED**. This can create repeatable reads when they are not desired. In other words, a transaction could have a copy of a stale bean. Second, this configuration option currently requires commit-option *B* or *C* which can be a performance drain since an

ejbLoad must happen at the beginning of the transaction. But, if your application currently requires commit-option *B* or *C* anyways, then this is the way to go. The JBoss developers are currently exploring ways to allow commit-option *A* as well (which would allow the use of caching for this option).

JBoss has container configurations named **Instance Per Transaction CMP 2.x EntityBean** and **Instance Per Transaction BMP EntityBean** defined in the standardjboss.xml that implement this locking policy. To use this configuration, you just have to reference the name of the container configuration to use with your bean in the jboss.xml deployment descriptor as show below.

Example 20.13. An example of using the Instance Per Transaction policy.

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>MyCMP2Bean</ejb-name>
      <jndi-name>MyCMP2</jndi-name>
      <configuration-name>
        Instance Per Transaction CMP 2.x EntityBean
      </configuration-name>
    </entity>
    <entity>
      <ejb-name>MyBMPBean</ejb-name>
      <jndi-name>MyBMP</jndi-name>
      <configuration-name>
        Instance Per Transaction BMP EntityBean
      </configuration-name>
    </entity>
  </enterprise-beans>
</jboss>
```

20.4.7. Running Within a Cluster

Currently there is no distributed locking capability for entity beans within the cluster. This functionality has been delegated to the database and must be supported by the application developer. For clustered entity beans, it is suggested to use commit-option *B* or *C* in combination with a row locking mechanism. For CMP, there is a row-locking configuration option. This option will use a SQL **select for update** when the bean is loaded from the database. With commit-option *B* or *C*, this implements a transactional lock that can be used across the cluster. For BMP, you must explicitly implement the select for update invocation within the BMP's **ejbLoad** method.

20.4.8. Troubleshooting

This section will describe some common locking problems and their solution.

20.4.8.1. Locking Behavior Not Working

Many JBoss users observe that locking does not seem to be working and see concurrent access to their beans, and thus dirty reads. Here are some common reasons for this:

- If you have custom **container-configurations**, make sure you have updated these configurations.

- Make absolutely sure that you have implemented **equals** and **hashCode** correctly from custom/complex primary key classes.
- Make absolutely sure that your custom/complex primary key classes serialize correctly. One common mistake is assuming that member variable initializations will be executed when a primary key is unmarshalled.

20.4.8.2. IllegalStateException

An `IllegalStateException` with the message "removing bean lock and it has tx set!" usually means that you have not implemented **equals** and/or **hashCode** correctly for your custom/complex primary key class, or that your primary key class is not implemented correctly for serialization.

20.4.8.3. Hangs and Transaction Timeouts

One long outstanding bug of JBoss is that on a transaction timeout, that transaction is only marked for a rollback and not actually rolled back. This responsibility is delegated to the invocation thread. This can cause major problems if the invocation thread hangs indefinitely since things like entity bean locks will never be released. The solution to this problem is not a good one. You really just need to avoid doing stuff within a transaction that could hang indefinitely. One common mistake is making connections across the internet or running a web-crawler within a transaction.

20.5. EJB TIMER CONFIGURATION

The J2EE timer service allows for any EJB object to register for a timer callback either at a designated time in the future. Timer events can be used for auditing, reporting or other cleanup tasks that need to need to happen at some given time in the future. Timer events are intended to be persistent and should be executed even in the event of a server failure. Coding to EJB timers is a standard part of the J2EE specification, so we won't explore the programming model. We will, instead, look at the configuration of the timer service in JBoss so that you can understand how to make timers work best in your environment

The EJB timer service is configure by several related MBeans in the **ejb-deployer.xml** file. The primary MBean is the **EJBTimerService** MBean.

```
<mbean code="org.jboss.ejb.txtimer.EJBTimerServiceImpl"
name="jboss.ejb:service=EJBTimerService">
  <attribute
name="RetryPolicy">jboss.ejb:service=EJBTimerService,retryPolicy=fixedDelay</attribute>
  <attribute
name="PersistencePolicy">jboss.ejb:service=EJBTimerService,persistencePolicy=database</attribute>
  <attribute
name="TimerIdGeneratorClassName">org.jboss.ejb.txtimer.BigIntegerTimerIdGenerator</attribute>
  <attribute
name="TimedObjectInvokerClassName">org.jboss.ejb.txtimer.TimedObjectInvokerImpl</attribute>
</mbean>
```

The **EJBTimerService** has the following configurable attributes:

- **RetryPolicy**: This is name of the MBean that implements the retry policy. The MBean must support the **org.jboss.ejb.txtimer.RetryPolicy interface**. JBoss provides one implementation, **FixedDelayRetryPolicy**, which will be described later.
- **PersistencePolicy**: This is the name of the MBean that implements the the persistence strategy for saving timer events. The MBean must support the **org.jboss.ejb.txtimer.PersistencePolicy interface**. JBoss provides two implementations, **NoopPersistencePolicy** and **DatabasePersistencePolicy**, which will be described later.
- **TimerIdGeneratorClassName**: This is the name of a class that provides the timer ID generator strategy. This class must implement the **org.jboss.ejb.txtimer.TimerIdGenerator interface**. JBoss provides the **org.jboss.ejb.txtimer.BigIntegerTimerIdGenerator** implementation.
- **TimedObjectInvokerClassname**: This is the name of a class that provides the timer method invocation strategy. This class must implement the **org.jboss.ejb.txtimer.TimedObjectInvoker interface**. JBoss provides the **org.jboss.ejb.txtimer.TimedObjectInvokerImpl** implementation.

The retry policy MBean definition used is shown here:

```
<mbean code="org.jboss.ejb.txtimer.FixedDelayRetryPolicy"
      name="jboss.ejb:service=EJBTimerService, retryPolicy=fixedDelay">
  <attribute name="Delay">100</attribute>
</mbean>
```

The retry policy takes one configuration value:

- **Delay**: This is the delay (ms) before retrying a failed timer execution. The default delay is 100ms.

If EJB timers do not need to be persisted, the **NoopPersistence** policy can be used. This MBean is commented out by default, but when enabled will look like this:

```
<mbean code="org.jboss.ejb.txtimer.NoopPersistencePolicy"
      name="jboss.ejb:service=EJBTimerService,persistencePolicy=noop"/>
```

Most applications that use timers will want timers to be persisted. For that the **DatabasePersistencePolicy** MBean should be used.

```
<mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
      name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
  <!-- DataSource JNDI name -->
  <depends optional-attribute-
name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</depe
nds>
  <!-- The plugin that handles database persistence -->
  <attribute
name="DatabasePersistencePlugin">org.jboss.ejb.txtimer.GeneralPurposeDatab
asePersistencePlugin</attribute>
</mbean>
```

- **DataSource**: This is the MBean for the DataSource that timer data will be written to.

- **DatabasePersistencePlugin:** This is the name of the class the implements the persistence strategy. This should be **`org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin`**.

CHAPTER 21. THE CMP ENGINE

This chapter will explore the use of container managed persistence (CMP) in JBoss. We will assume a basic familiarity the EJB CMP model and focus on the operation of the JBoss CMP engine. Specifically, we will look at how to configure and optimize CMP applications on JBoss. For more introductory coverage of basic CMP concepts, we recommend *Enterprise Java Beans, Fourth Edition* (O'Reilly 2004).

21.1. EXAMPLE CODE

This chapter is example-driven. We will work with the crime portal application which stores information about imaginary criminal organizations. The data model we will be working with is shown in [Figure 21.1](#), “The crime portal example classes”.

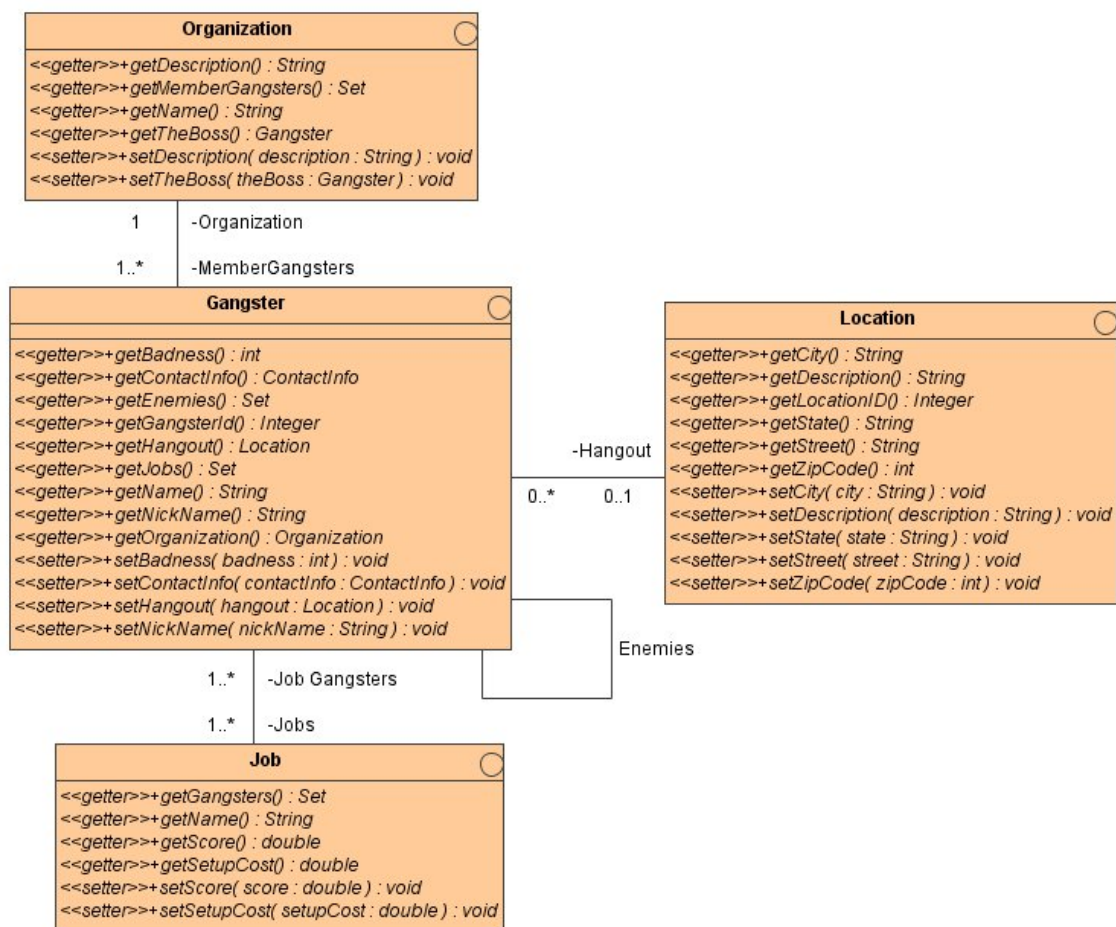


Figure 21.1. The crime portal example classes

The source code for the crime portal is available in the `src/main/org/jboss/cmp2` directory of the example code. To build the example code, run Ant as shown below

```
[examples]$ ant -Dchap=cmp2 config
```

This command builds and deploys the application to the JBoss server. When you start your JBoss server, or if it is already running, you should see the following deployment messages:

```

15:46:36,704 INFO [OrganizationBean$Proxy] Creating organization Yakuza,
Japanese Gangsters
15:46:36,790 INFO [OrganizationBean$Proxy] Creating organization Mafia,
Italian Bad Guys
  
```

```

15:46:36,797 INFO  [OrganizationBean$Proxy] Creating organization Triads,
Kung Fu Movie Extras
15:46:36,877 INFO  [GangsterBean$Proxy] Creating Gangster 0 'Bodyguard'
Yojimbo
15:46:37,003 INFO  [GangsterBean$Proxy] Creating Gangster 1 'Master'
Takeshi
15:46:37,021 INFO  [GangsterBean$Proxy] Creating Gangster 2 'Four finger'
Yuriko
15:46:37,040 INFO  [GangsterBean$Proxy] Creating Gangster 3 'Killer' Chow
15:46:37,106 INFO  [GangsterBean$Proxy] Creating Gangster 4 'Lightning'
Shogi
15:46:37,118 INFO  [GangsterBean$Proxy] Creating Gangster 5 'Pizza-Face'
Valentino
15:46:37,133 INFO  [GangsterBean$Proxy] Creating Gangster 6 'Toothless'
Toni
15:46:37,208 INFO  [GangsterBean$Proxy] Creating Gangster 7 'Godfather'
Corleone
15:46:37,238 INFO  [JobBean$Proxy] Creating Job 10th Street Jeweler Heist
15:46:37,247 INFO  [JobBean$Proxy] Creating Job The Greate Train Robbery
15:46:37,257 INFO  [JobBean$Proxy] Creating Job Cheap Liquor Snatch and
Grab

```

Since the beans in the examples are configured to have their tables removed on undeployment, anytime you restart the JBoss server you need to rerun the config target to reload the example data and re-deploy the application.

21.1.1. Enabling CMP Debug Logging

In order to get meaningful feedback from the chapter tests, you will want to increase the log level of the CMP subsystem before running the test. To enable debug logging add the following category to your **log4j.xml** file:

```

<category name="org.jboss.ejb.plugins.cmp">
  <priority value="DEBUG"/>
</category>

```

In addition to this, it is necessary to decrease the threshold on the **CONSOLE** appender to allow debug level messages to be logged to the console. The following changes also need to be applied to the **log4j.xml** file.

```

<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="Target" value="System.out"/>
  <param name="Threshold" value="DEBUG" />

  <layout class="org.apache.log4j.PatternLayout">
    <!-- The default pattern: Date Priority [Category] Message\n -->
    <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
  </layout>
</appender>

```

To see the full workings of the CMP engine you would need to enable the custom **TRACE** level priority on the **org.jboss.ejb.plugins.cmp** category as shown here:

-

```
<category name="org.jboss.ejb.plugins.cmp">
  <priority value="TRACE" class="org.jboss.logging.XLevel"/>
</category>
```

21.1.2. Running the examples

The first test target illustrates a number of the customization features that will be discussed throughout this chapter. To run these tests execute the following ant target:

```
[examples]$ ant -Dchap=cmp2 -Dex=test run-example
```

```
22:30:09,862 DEBUG [OrganizationEJB#findByPrimaryKey] Executing SQL:
SELECT t0_OrganizationEJ
B.name FROM ORGANIZATION t0_OrganizationEJB WHERE
t0_OrganizationEJB.name=?
22:30:09,927 DEBUG [OrganizationEJB] Executing SQL: SELECT desc, the_boss
FROM ORGANIZATION W
HERE (name=?)
22:30:09,931 DEBUG [OrganizationEJB] load relation SQL: SELECT id FROM
GANGSTER WHERE (organi
zation=?)
22:30:09,947 DEBUG [StatelessSessionContainer] Useless invocation of
remove() for stateless s
ession bean
22:30:10,086 DEBUG [GangsterEJB#findBadDudes_ejbql] Executing SQL: SELECT
t0_g.id FROM GANGST
ER t0_g WHERE (t0_g.badness > ?)
22:30:10,097 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT
t0_GangsterEJB.id FRO
M GANGSTER t0_GangsterEJB WHERE t0_GangsterEJB.id=?
22:30:10,102 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT
t0_GangsterEJB.id FRO
M GANGSTER t0_GangsterEJB WHERE t0_GangsterEJB.id=?
```

These tests exercise various finders, selectors and object to table mapping issues. We will refer to the tests throughout the chapter.

The other main target runs a set of tests to demonstrate the optimized loading configurations presented in [Section 21.7, “Optimized Loading”](#). Now that the logging is setup correctly, the read-ahead tests will display useful information about the queries performed. Note that you do not have to restart the JBoss server for it to recognize the changes to the log4j.xml file, but it may take a minute or so. The following shows the actual execution of the readahead client:

```
[examples]$ ant -Dchap=cmp2 -Dex=readahead run-example
```

When the readahead client is executed, all of the SQL queries executed during the test are displayed in the JBoss server console. The important items of note when analyzing the output are the number of queries executed, the columns selected, and the number of rows loaded. The following shows the read-ahead none portion of the JBoss server console output from readahead:

```
22:44:31,570 INFO [ReadAheadTest]
#####
### read-ahead none
###
```

```

22:44:31,582 DEBUG [GangsterEJB#findAll_none] Executing SQL: SELECT
t0_g.id FROM GANGSTER t0_
g ORDER BY t0_g.id ASC
22:44:31,604 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,615 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,622 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,628 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,635 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,644 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,649 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,658 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,670 INFO [ReadAheadTest]
###
#####
...

```

We will revisit this example and explore the output when we discuss the settings for optimized loading.

21.2. THE JBOSSCMP-JDBC STRUCTURE

The `jbosscmp-jdbc.xml` descriptor is used to control the behavior of the JBoss engine. This can be done globally through the `conf/standardjbosscmp-jdbc.xml` descriptor found in the server configuration file set, or per EJB JAR deployment via a `META-INF/jbosscmp-jdbc.xml` descriptor.

The DTD for the `jbosscmp-jdbc.xml` descriptor can be found in `JBOSS_DIST/docs/dtd/jbosscmp-jdbc_4_0.dtd`. The public doctype for this DTD is:

```

<!DOCTYPE jbosscmp-jdbc PUBLIC
"-//JBoss//DTD JBOSSCMP-JDBC 4.0//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_4_0.dtd">

```

The top level elements are shown in [Figure 21.2, “The jbosscmp-jdbc content model.”](#).

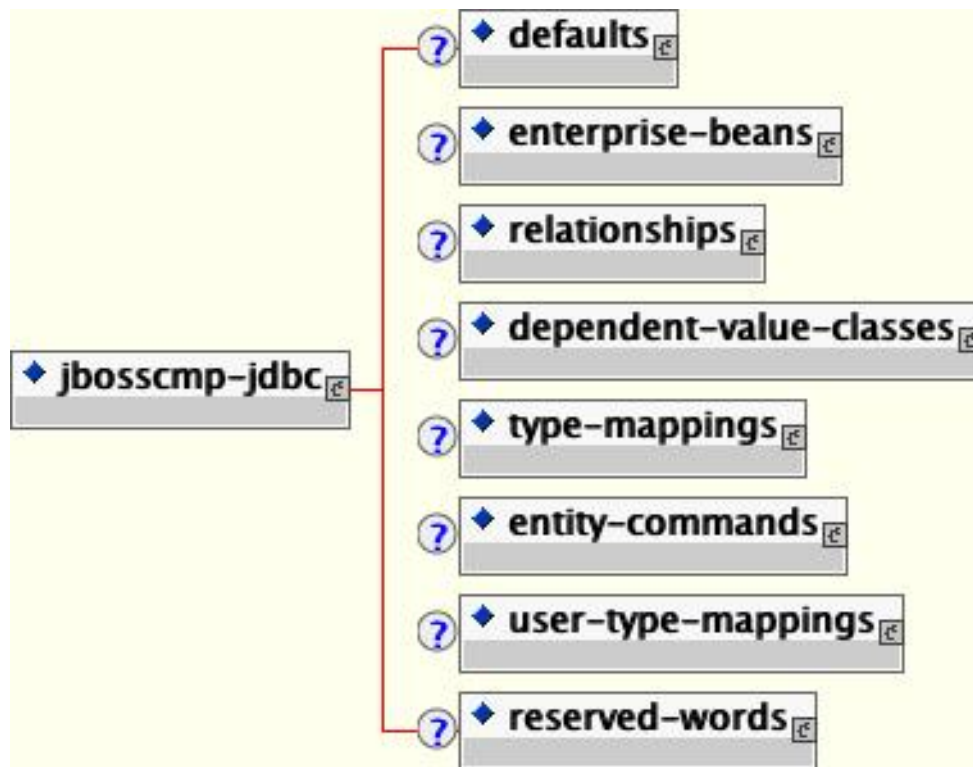


Figure 21.2. The jbossCMP-jdbc content model.

- **defaults:** The **defaults** section allows for the specification of default behavior/settings for behavior that controls entity beans. Use of this section simplifies the amount of information needed for the common behaviors found in the entity beans section. See [Section 21.12, “Defaults”](#) for the details of the defaults content.
- **enterprise-beans:** The **enterprise-beans** element allows for customization of entity beans defined in the `ejb-jar.xml` **enterprise-beans** descriptor. This is described in detail in [Section 21.3, “Entity Beans”](#).
- **relationships:** The **relationships** element allows for the customization of tables and the loading behavior of entity relationships. This is described in detail in [Section 21.5, “Container Managed Relationships”](#).
- **dependent-value-classes:** The **dependent-value-classes** element allows for the customization of the mapping of dependent value classes to tables. Dependent value classes are described in detail in [Section 21.4.5, “Dependent Value Classes \(DVCs\)”](#) (DVCs).
- **type-mappings:** The **type-mappings** element defines the Java to SQL type mappings for a database, along with SQL templates, and function mappings. This is described in detail in [Section 21.13, “Datasource Customization”](#).
- **entity-commands:** The **entity-commands** element allows for the definition of the entity creation command instances that know how to create an entity instance in a persistent store. This is described in detail in [Section 21.11, “Entity Commands and Primary Key Generation”](#).
- **user-type-mappings:** The **user-type-mappings** elements defines a mapping of a user types to a column using a mapper class. A mapper is like a mediator. When storing, it takes an instance of the user type and translates it to a column value. When loading, it takes a column value and translates it to an instance of the user type. Details of the user type mappings are described in [Section 21.13.4, “User Type Mappings”](#).

- **reserved-words:** The **reserved-words** element defines one or more reserved words that should be escaped when generating tables. Each reserved word is specified as the content of a **word** element.

21.3. ENTITY BEANS

We'll start our look at entity beans in JBoss by examining one of the CMP entity beans in the crime portal. We'll look at the gangster bean, which is implemented as local CMP entity bean. Although JBoss can provide remote entity beans with pass-by-reference semantics for calls in the same VM to get the performance benefit as from local entity beans, the use of local entity beans is strongly encouraged.

We'll start with the required home interface. Since we're only concerned with the CMP fields at this point, we'll show only the methods dealing with the CMP fields.

```
// Gangster Local Home Interface
public interface GangsterHome
    extends EJBLocalHome
{
    Gangster create(Integer id, String name, String nickName)
        throws CreateException;
    Gangster findByPrimaryKey(Integer id)
        throws FinderException;
}
```

The local interface is what clients will use to talk. Again, it contains only the CMP field accessors.

```
// Gangster Local Interface
public interface Gangster
    extends EJBLocalObject
{
    Integer getGangsterId();

    String getName();

    String getNickName();
    void setNickName(String nickName);

    int getBadness();
    void setBadness(int badness);
}
```

Finally, we have the actual gangster bean. Despite its size, very little code is actually required. The bulk of the class is the create method.

```
// Gangster Implementation Class
public abstract class GangsterBean
    implements EntityBean
{
    private EntityContext ctx;
    private Category log = Category.getInstance(getClass());
    public Integer ejbCreate(Integer id, String name, String nickName)
        throws CreateException
    {
        log.info("Creating Gangster " + id + " '" + nickName + "' " +
name);
    }
}
```

```

        setGangsterId(id);
        setName(name);
        setNickName(nickName);
        return null;
    }

    public void ejbPostCreate(Integer id, String name, String nickName) {
    }

    // CMP field accessors -----
--
    public abstract Integer getGangsterId();
    public abstract void setGangsterId(Integer gangsterId);
    public abstract String getName();
    public abstract void setName(String name);
    public abstract String getNickName();
    public abstract void setNickName(String nickName);
    public abstract int getBadness();
    public abstract void setBadness(int badness);
    public abstract ContactInfo getContactInfo();
    public abstract void setContactInfo(ContactInfo contactInfo);
    //...

    // EJB callbacks -----
--
    public void setEntityContext(EntityContext context) { ctx = context;
}
    public void unsetEntityContext() { ctx = null; }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void ejbRemove() { log.info("Removing " + getName()); }
    public void ejbStore() { }
    public void ejbLoad() { }
}

```

The only thing missing now is the **ejb-jar.xml** deployment descriptor. Although the actual bean class is named **GangsterBean**, we've called the entity **GangsterEJB**.

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-
J2EE_Certification_and_Standards_Compliance" version="2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/ebb-
jar_2_1.xsd">
    <display-name>Crime Portal</display-name>

    <enterprise-beans>
        <entity>
            <display-name>Gangster Entity Bean</display-name>
            <ejb-name>GangsterEJB</ejb-name>
            <local-home>org.jboss.cmp2.crimeportal.GangsterHome</local-
home>
            <local>org.jboss.cmp2.crimeportal.Gangster</local>

            <ejb-class>org.jboss.cmp2.crimeportal.GangsterBean</ejb-class>

```

```

    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>gangster</abstract-schema-name>

    <cmp-field>
      <field-name>gangsterId</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>name</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>nickName</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>badness</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>contactInfo</field-name>
    </cmp-field>
    <primkey-field>gangsterId</primkey-field>

    <!-- ... -->
  </entity>
</enterprise-beans>
</ejb-jar>

```

Note that we've specified a CMP version of **2.x** to indicate that this is EJB 2.x CMP entity bean. The abstract schema name was set to **gangster**. That will be important when we look at EJB-QL queries in [Section 21.6, "Queries"](#).

21.3.1. Entity Mapping

The JBoss configuration for the entity is declared with an **entity** element in the **jbosscomp-jdbc.xml** file. This file is located in the **META-INF** directory of the EJB JAR and contains all of the optional configuration information for configuring the CMP mapping. The **entity** elements for each entity bean are grouped together in the **enterprise-beans** element under the top level **jbosscomp-jdbc** element. A stubbed out entity configuration is shown below.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jbosscomp-jdbc PUBLIC
    "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jbosscomp-jdbc_3_2.dtd">
<jbosscomp-jdbc>
  <defaults>
    <!-- application-wide CMP defaults -->
  </defaults>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- overrides to defaults section -->
      <table-name>gangster</table-name>
      <!-- CMP Fields (see CMP-Fields) -->
      <!-- Load Groups (see Load Groups)-->
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>

```



```
        <!-- Queries (see Queries) -->
    </entity>
</enterprise-beans>
</jbosscmp-jdbc>
```

The **ejb-name** element is required to match the entity specification here with the one in the **ejb-jar.xml** file. The remainder of the elements specify either overrides the global or application-wide CMP defaults and CMP mapping details specific to the bean. The application defaults come from the **defaults** section of the **jbosscmp-jdbc.xml** file and the global defaults come from the **defaults** section of the **standardjbosscmp-jdbc.xml** file in the **conf** directory for the current server configuration file set. The **defaults** section is discussed in [Section 21.12, “Defaults”](#). [Figure 21.3, “The entity element content model”](#) shows the full **entity** content model.

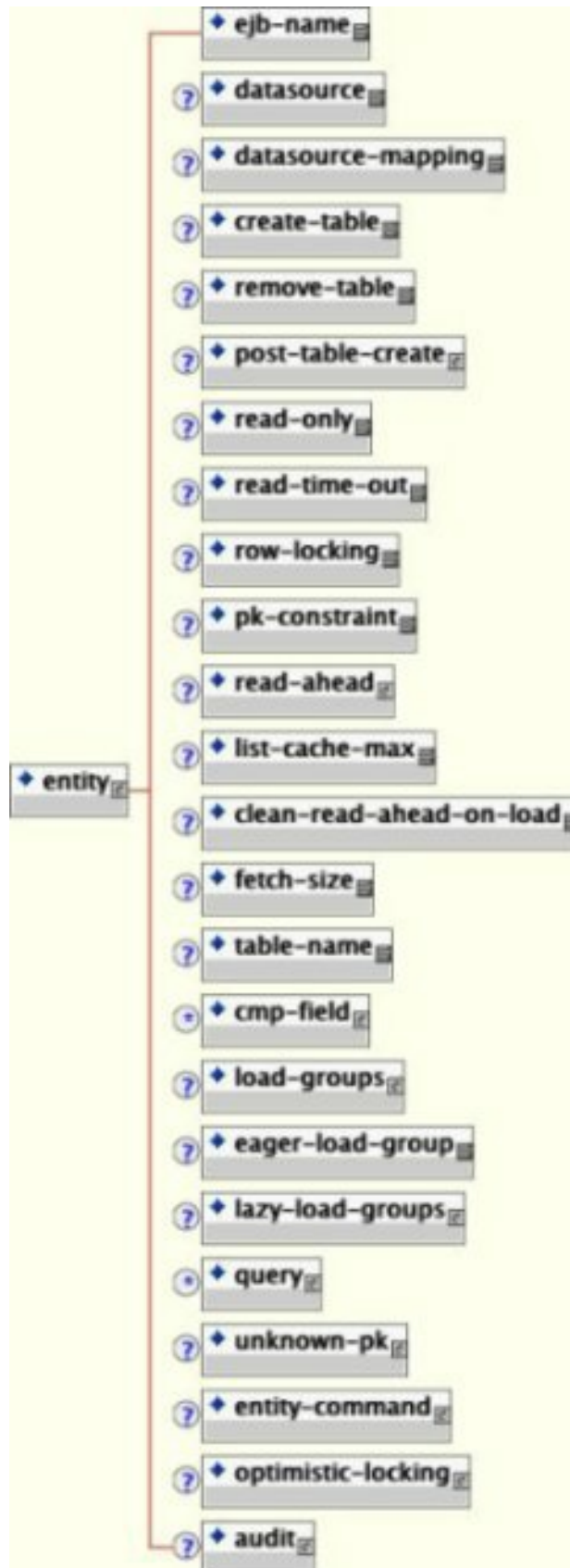


Figure 21.3. The entity element content model

A detailed description of each entity element follows:

- **ejb-name**: This required element is the name of the EJB to which this configuration applies. This element must match an **ejb-name** of an entity in the **ejb-jar.xml** file.
- **datasource**: This optional element is the **jndi-name** used to look up the datasource. All

database connections used by an entity or relation-table are obtained from the datasource. Having different datasources for entities is not recommended, as it vastly constrains the domain over which finders and ejbSelects can query. The default is `java:/DefaultDS` unless overridden in the defaults section.

- **datasource-mapping**: This optional element specifies the name of the **type-mapping**, which determines how Java types are mapped to SQL types, and how EJB-QL functions are mapped to database specific functions. Type mappings are discussed in [Section 21.13.3, “Mapping”](#). The default is **Hypersonic SQL** unless overridden in the defaults section.
- **create-table**: This optional element when true, specifies that JBoss should attempt to create a table for the entity. When the application is deployed, JBoss checks if a table already exists before creating the table. If a table is found, it is logged, and the table is not created. This option is very useful during the early stages of development when the table structure changes often. The default is false unless overridden in the defaults section.
- **alter-table**: If **create-table** is used to automatically create the schema, **alter-table** can be used to keep the schema current with changes to the entity bean. Alter table will perform the following specific tasks:
 - new fields will be created
 - fields which are no longer used will be removed
 - string fields which are shorter than the declared length will have their length increased to the declared length. (not supported by all databases)
- **remove-table**: This optional element when true, JBoss will attempt to drop the table for each entity and each relation table mapped relationship. When the application is undeployed, JBoss will attempt to drop the table. This option is very useful during the early stages of development when the table structure changes often. The default is false unless overridden in the defaults section.
- **post-table-create**: This optional element specifies an arbitrary SQL statement that should be executed immediately after the database table is created. This command is only executed if **create-table** is true and the table did not previously exist.
- **read-only**: This optional element when true specifies that the bean provider will not be allowed to change the value of any fields. A field that is read-only will not be stored in, or inserted into, the database. If a primary key field is read-only, the create method will throw a **CreateException**. If a set accessor is called on a read-only field, it throws an **EJBException**. Read-only fields are useful for fields that are filled in by database triggers, such as last update. The **read-only** option can be overridden on a per **cmp-field** basis, and is discussed in [Section 21.4.3, “Read-only Fields”](#). The default is false unless overridden in the **defaults** section.
- **read-time-out**: This optional element is the amount of time in milliseconds that a read on a read-only field is valid. A value of 0 means that the value is always reloaded at the start of a transaction, and a value of -1 means that the value never times out. This option can also be overridden on a per **cmp-field** basis. If **read-only** is false, this value is ignored. The default is -1 unless overridden in the **defaults** section.
- **row-locking**: This optional element if true specifies that JBoss will lock all rows loaded in a transaction. Most databases implement this by using the **SELECT FOR UPDATE** syntax when loading the entity, but the actual syntax is determined by the **row-locking-template** in the

datasource-mapping used by this entity. The default is false unless overridden in the **defaults** section.

- **pk-constraint**: This optional element if true specifies that JBoss will add a primary key constraint when creating tables. The default is true unless overridden in the defaults section.
- **read-ahead**: This optional element controls caching of query results and **cmr-fields** for the entity. This option is discussed in [Section 21.7.3, “Read-ahead”](#).
- **fetch-size**: This optional element specifies the number of entities to read in one round-trip to the underlying datastore. The default is 0 unless overridden in the defaults section.
- **list-cache-max**: This optional element specifies the number of read-lists that can be tracked by this entity. This option is discussed in **on-load**. The default is 1000 unless overridden in the defaults section.
- **clean-read-ahead-on-load**: When an entity is loaded from the read ahead cache, JBoss can remove the data used from the read ahead cache. The default is **false**.
- **table-name**: This optional element is the name of the table that will hold data for this entity. Each entity instance will be stored in one row of this table. The default is the **ejb-name**.
- **cmp-field**: The optional element allows one to define how the **ejb-jar.xml** **cmp-field** is mapped onto the persistence store. This is discussed in [Section 21.4, “CMP Fields”](#).
- **load-groups**: This optional element specifies one or more groupings of CMP fields to declare load groupings of fields. This is discussed in [Section 21.7.2, “Load Groups”](#).
- **eager-load-groups**: This optional element defines one or more load grouping as eager load groups. This is discussed in [Section 21.8.2, “Eager-loading Process”](#).
- **lazy-load-groups**: This optional element defines one or more load grouping as lazy load groups. This is discussed in [Section 21.8.3, “Lazy loading Process”](#).
- **query**: This optional element specifies the definition of finders and selectors. This is discussed in [Section 21.6, “Queries”](#).
- **unknown-pk**: This optional element allows one to define how an unknown primary key type of `java.lang.Object` maps to the persistent store.
- **entity-command**: This optional element allows one to define the entity creation command instance. Typically this is used to define a custom command instance to allow for primary key generation. This is described in detail in [Section 21.11, “Entity Commands and Primary Key Generation”](#).
- **optimistic-locking**: This optional element defines the strategy to use for optimistic locking. This is described in detail in [Section 21.10, “Optimistic Locking”](#).
- **audit**: This optional element defines the CMP fields that will be audited. This is described in detail in [Section 21.4.4, “Auditing Entity Access”](#).

21.4. CMP FIELDS

CMP fields are declared on the bean class as abstract getter and setter methods that follow the JavaBean property accessor conventions. Our gangster bean, for example, has a **getName()** and a **setName()** method for accessing the **name** CMP field. In this section we will look at how the configure

these declared CMP fields and control the persistence and behavior.

21.4.1. CMP Field Declaration

The declaration of a CMP field starts in the **ejb-jar.xml** file. On the gangster bean, for example, the **gangsterId**, **name**, **nickName** and **badness** would be declared in the **ejb-jar.xml** file as follows:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field><field-name>gangsterId</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>nickName</field-name></cmp-field>
      <cmp-field><field-name>badness</field-name></cmp-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

Note that the J2EE deployment descriptor doesn't declare any object-relational mapping details or other configuration. It is nothing more than a simple declaration of the CMP fields.

21.4.2. CMP Field Column Mapping

The relational mapping configuration of a CMP field is done in the **jbosscmp-jdbc.xml** file. The structure is similar to the **ejb-jar.xml** with an entity **element** that has **cmp-field** elements under it with the additional configuration details.

The following is shows the basic column name and data type mappings for the gangster bean.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <table-name>gangster</table-name>

      <cmp-field>
        <field-name>gangsterId</field-name>
        <column-name>id</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>name</field-name>
        <column-name>name</column-name>
        <not-null/>
      </cmp-field>
      <cmp-field>
        <field-name>nickName</field-name>
        <column-name>nick_name</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(64)</sql-type>
      </cmp-field>
      <cmp-field>
        <field-name>badness</field-name>
        <column-name>badness</column-name>
      </cmp-field>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

```

    </entity>
  </enterprise-beans>
    </jbosscmp-jdbc>

```

The full content model of the **cmp-field** element of the **jbosscmp-jdbc.xml** is shown below.

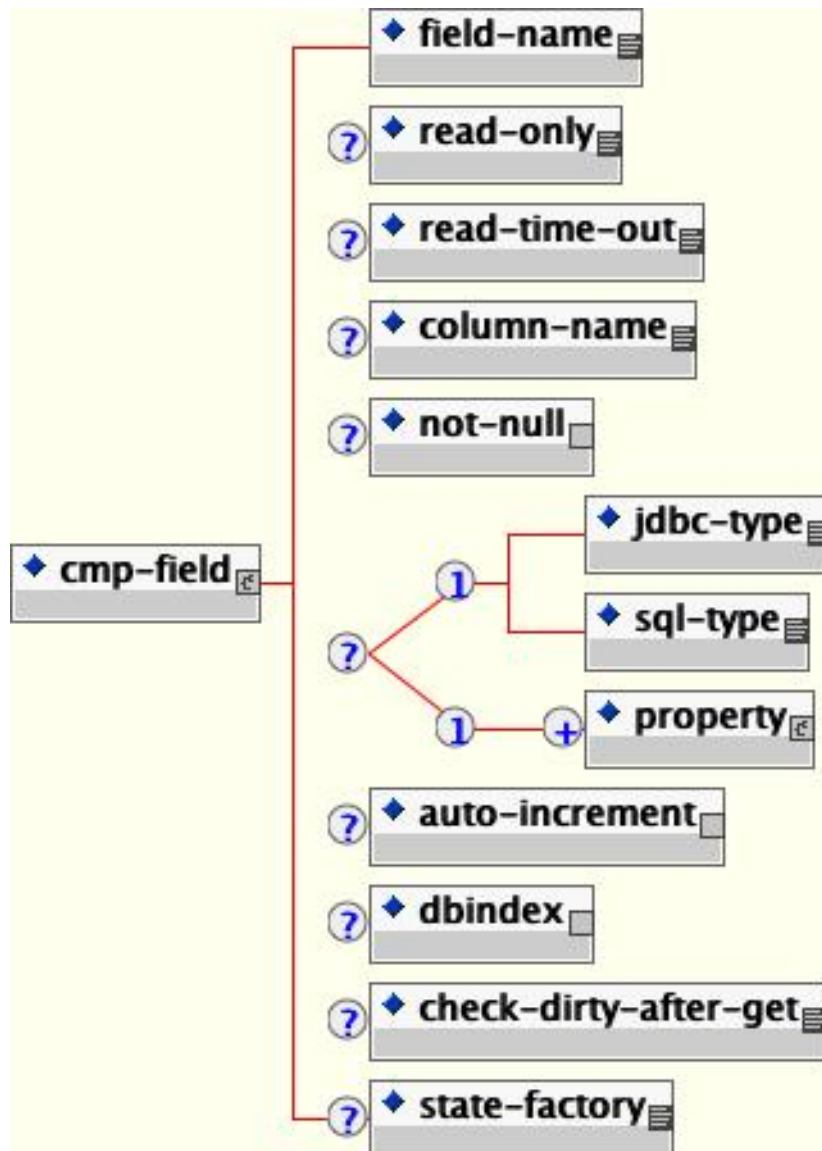


Figure 21.4. The JBoss entity element content model

A detailed description of each element follows:

- **field-name**: This required element is the name of the **cmp-field** that is being configured. It must match the **field-name** element of a **cmp-field** declared for this entity in the **ejb-jar.xml** file.
- **read-only**: This declares that field in question is read-only. This field will not be written to the database by JBoss. Read-only fields are discussed in [Section 21.4.3, “Read-only Fields”](#).
- **read-only-timeout**: This is the time in milliseconds that a read-only field value will be considered valid.
- **column-name**: This optional element is the name of the column to which the **cmp-field** is mapped. The default is to use the **field-name** value.

- **not-null**: This optional element indicates that JBoss should add a NOT NULL to the end of the column declaration when automatically creating the table for this entity. The default for primary key fields and primitives is not null.
- **jdbc-type**: This is the JDBC type that is used when setting parameters in a JDBC prepared statement or loading data from a JDBC result set. The valid types are defined in `java.sql.Types`. This is only required if **sql-type** is specified. The default JDBC type will be based on the database type in the **datasourcemappping**.
- **sql-type**: This is the SQL type that is used in create table statements for this field. Valid SQL types are only limited by your database vendor. This is only required if **jdbc-type** is specified. The default SQL type will be base on the database type in the **datasourcemappping**
- **property**: This optional element allows one to define how the properties of a dependent value class CMP field should be mapped to the persistent store. This is discussed further in [Section 21.4.5, “Dependent Value Classes \(DVCs\)”](#).
- **auto-increment**: The presence of this optional field indicates that it is automatically incremented by the database layer. This is used to map a field to a generated column as well as to an externally manipulated column.
- **dbindex**: The presence of this optional field indicates that the server should create an index on the corresponding column in the database. The index name will be `fieldname_index`.
- **check-dirty-after-get**: This value defaults to false for primitive types and the basic `java.lang` immutable wrappers (**Integer**, **String**, etc...). For potentially mutable objects, JBoss will mark they field as potentially dirty after a get operation. If the dirty check on an object is too expensive, you can optimize it away by setting **check-dirty-after-get** to false.
- **state-factory**: This specifies class name of a state factory object which can perform dirty checking for this field. State factory classes must implement the **CMPFieldStateFactory** interface.

21.4.3. Read-only Fields

JBoss allows for read-only CMP fields by setting the **read-only** and **read-time-out** elements in the **cmp-field** declaration. These elements work the same way as they do at the entity level. If a field is read-only, it will never be used in an **INSERT** or **UPDATE** statement. If a primary key field is **read-only**, the create method will throw a **CreateException**. If a set accessor is called for a read-only field, it throws an **EJBException**. Read-only fields are useful for fields that are filled in by database triggers, such as last update. A read-only CMP field declaration example follows:

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field>
        <field-name>lastUpdated</field-name>
        <read-only>true</read-only>
        <read-time-out>1000</read-time-out>
      </cmp-field>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```


21.4.4. Auditing Entity Access

The **audit** element of the entity section allows one to specify how access to an entity bean is audited. This is only allowed when an entity bean is accessed under a security domain so that this is a caller identity established. The content model of the audit element is given [Figure 21.5, “The jbosscmp-jdbc.xml audit element content model”](#).

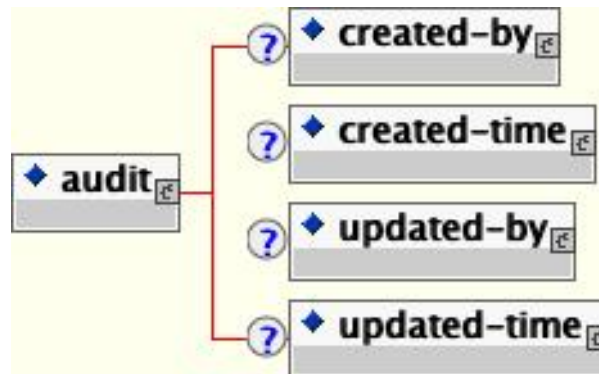


Figure 21.5. The jbosscmp-jdbc.xml audit element content model

- **created-by**: This optional element indicates that the caller who created the entity should be saved to either the indicated **column-name** or cmp **field-name**.
- **created-time**: This optional element indicates that the time of entity creation should be saved to either the indicated **column-name** or cmp **field-name**.
- **updated-by**: This optional element indicates that the caller who last modified the entity should be saved to either the indicated **column-name** or CMP **field-name**.
- **updated-time**: This optional element indicates that the last time of entity modification should be saved to either the indicated **column-name** or CMP **field-name**.

For each element, if a **field-name** is given, the corresponding audit information should be stored in the specified CMP field of the entity bean being accessed. Note that there does not have to be an corresponding CMP field declared on the entity. In case there are matching field names, you will be able to access audit fields in the application using the corresponding CMP field abstract getters and setters. Otherwise, the audit fields will be created and added to the entity internally. You will be able to access audit information in EJB-QL queries using the audit field names, but not directly through the entity accessors.

If, on the other hand, a **column-name** is specified, the corresponding audit information should be stored in the indicated column of the entity table. If JBoss is creating the table the **jdbc-type** and **sql-type** element can then be used to define the storage type.

The declaration of audit information with given column names is shown below.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>AuditChangedNamesEJB</ejb-name>
      <table-name>cmp2_audit_changednames</table-name>
      <audit>
        <created-by>
          <column-name>createdby</column-name>
        </created-by>
        <created-time>
```



```

        <column-name>createdtime</column-name>
      </created-time>
      <updated-by>
        <column-name>updatedby</column-name></updated-by>
      <updated-time>
        <column-name>updatedtime</column-name>
      </updated-time>
    </audit>
  </entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

21.4.5. Dependent Value Classes (DVCs)

A dependent value class (DVC) is a fancy term used to identify any Java class that is the type of a **cmp-field** other than the automatically recognized types core types such as strings and number values. By default, a DVC is serialized, and the serialized form is stored in a single database column. Although not discussed here, there are several known issues with the long-term storage of classes in serialized form.

JBoss also supports the storage of the internal data of a DVC into one or more columns. This is useful for supporting legacy JavaBeans and database structures. It is not uncommon to find a database with a highly flattened structure (e.g., a **PURCHASE_ORDER** table with the fields **SHIP_LINE1**, **SHIP_LINE2**, **SHIP_CITY**, etc. and an additional set of fields for the billing address). Other common database structures include telephone numbers with separate fields for area code, exchange, and extension, or a person's name spread across several fields. With a DVC, multiple columns can be mapped to one logical field.

JBoss requires that a DVC to be mapped must follow the JavaBeans naming specification for simple properties, and that each property to be stored in the database must have both a getter and a setter method. Furthermore, the bean must be serializable and must have a no argument constructor. A property can be any simple type, an unmapped DVC or a mapped DVC, but cannot be an EJB. A DVC mapping is specified in a **dependent-value-class** element within the **dependent-value-classes** element.

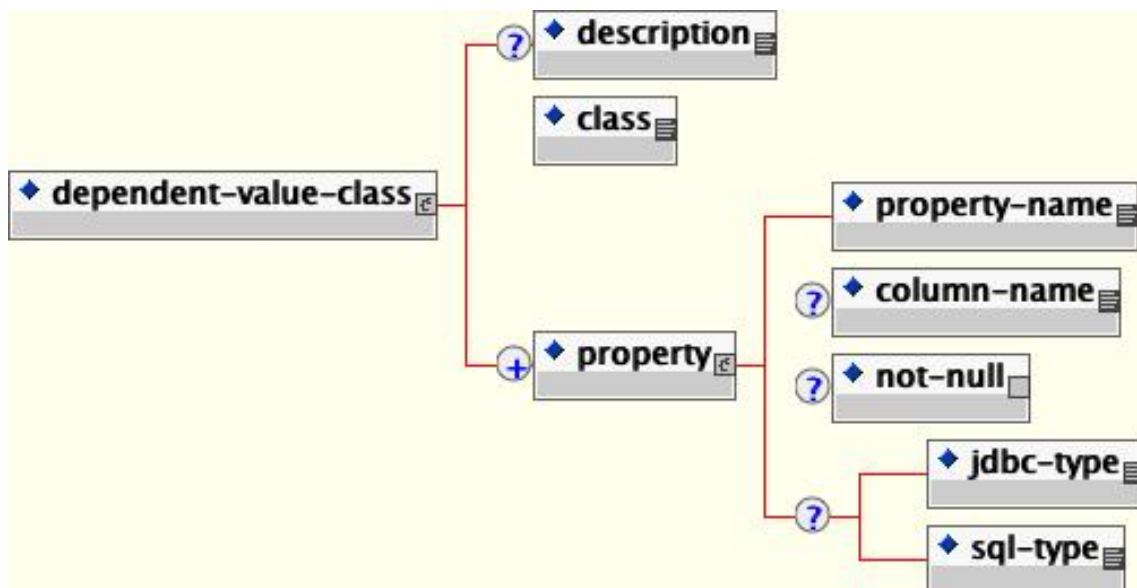


Figure 21.6. The jbosscomp-jdbc dependent-value-class element model.

Here is an example of a simple **ContactInfo** DVC class.

```
public class ContactInfo
    implements Serializable
{
    /** The cell phone number. */
    private PhoneNumber cell;

    /** The pager number. */
    private PhoneNumber pager;

    /** The email address */
    private String email;

    /**
     * Creates empty contact info.
     */
    public ContactInfo() {
    }

    public PhoneNumber getCell() {
        return cell;
    }

    public void setCell(PhoneNumber cell) {
        this.cell = cell;
    }

    public PhoneNumber getPager() {
        return pager;
    }

    public void setPager(PhoneNumber pager) {
        this.pager = pager;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email.toLowerCase();
    }

    // ... equals, hashCode, toString
}
```

The contact info includes a phone number, which is represented by another DVC class.

```
public class PhoneNumber
    implements Serializable
{
    /** The first three digits of the phone number. */
    private short areaCode;

    /** The middle three digits of the phone number. */
    private short exchange;
```

```

    /** The last four digits of the phone number. */
    private short extension;

    // ... getters and setters

    // ... equals, hashCode, toString
}

```

The DVC mappings for these two classes are relatively straight forward.

```

<dependent-value-classes>
  <dependent-value-class>
    <description>A phone number</description>
    <class>org.jboss.cmp2.crimeportal.PhoneNumber</class>
    <property>
      <property-name>areaCode</property-name>
      <column-name>area_code</column-name>
    </property>
    <property>
      <property-name>exchange</property-name>
      <column-name>exchange</column-name>
    </property>
    <property>
      <property-name>extension</property-name>
      <column-name>extension</column-name>
    </property>
  </dependent-value-class>

  <dependent-value-class>
    <description>General contact info</description>
    <class>org.jboss.cmp2.crimeportal.ContactInfo</class>
    <property>
      <property-name>cell</property-name>
      <column-name>cell</column-name>
    </property>
    <property>
      <property-name>pager</property-name>
      <column-name>pager</column-name>
    </property>
    <property>
      <property-name>email</property-name>
      <column-name>email</column-name>
      <jdbc-type>VARCHAR</jdbc-type>
      <sql-type>VARCHAR(128)</sql-type>
    </property>
  </dependent-value-class>
</dependent-value-classes>

```

Each DVC is declared with a **dependent-value-class** element. A DVC is identified by the Java class type declared in the class element. Each property to be persisted is declared with a property element. This specification is based on the **cmp-field** element, so it should be self-explanatory. This restriction will also be removed in a future release. The current proposal involves storing the primary key fields in the case of a local entity and the entity handle in the case of a remote entity.

The **dependent-value-classes** section defines the internal structure and default mapping of the classes. When JBoss encounters a field that has an unknown type, it searches the list of registered DVCs, and if a DVC is found, it persists this field into a set of columns, otherwise the field is stored in serialized form in a single column. JBoss does not support inheritance of DVCs; therefore, this search is only based on the declared type of the field. A DVC can be constructed from other DVCs, so when JBoss runs into a DVC, it flattens the DVC tree structure into a set of columns. If JBoss finds a DVC circuit during startup, it will throw an **EJBException**. The default column name of a property is the column name of the base **cmp-field** followed by an underscore and then the column name of the property. If the property is a DVC, the process is repeated. For example, a **cmp-field** named **info** that uses the **ContactInfo** DVC would have the following columns:

```
info_cell_area_code
info_cell_exchange
info_cell_extension
info_pager_area_code
info_pager_exchange
info_pager_extension
info_email
```

The automatically generated column names can quickly become excessively long and awkward. The default mappings of columns can be overridden in the entity element as follows:

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field>
        <field-name>contactInfo</field-name>
        <property>
          <property-name>cell.areaCode</property-name>
          <column-name>cell_area</column-name>
        </property>
        <property>
          <property-name>cell.exchange</property-name>
          <column-name>cell_exch</column-name>
        </property>
        <property>
          <property-name>cell.extension</property-name>
          <column-name>cell_ext</column-name>
        </property>

        <property>
          <property-name>pager.areaCode</property-name>
          <column-name>page_area</column-name>
        </property>
        <property>
          <property-name>pager.exchange</property-name>
          <column-name>page_exch</column-name>
        </property>
        <property>
          <property-name>pager.extension</property-name>
          <column-name>page_ext</column-name>
        </property>

        <property>
```

```

        <property-name>email</property-name>
        <column-name>email</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(128)</sql-type>
    </property>
</cmp-field>
</entity>
</enterprise-beans>
</jboss-cmp-jdbc>

```

When overriding property info for the entity, you need to refer to the property from a flat perspective as in `cell.areaCode`.

21.5. CONTAINER MANAGED RELATIONSHIPS

Container Managed Relationships (CMRs) are a powerful new feature of CMP 2.0. Programmers have been creating relationships between entity objects since EJB 1.0 was introduced (not to mention since the introduction of databases), but before CMP 2.0 the programmer had to write a lot of code for each relationship in order to extract the primary key of the related entity and store it in a pseudo foreign key field. The simplest relationships were tedious to code, and complex relationships with referential integrity required many hours to code. With CMP 2.0 there is no need to code relationships by hand. The container can manage one-to-one, one-to-many and many-to-many relationships, with referential integrity. One restriction with CMRs is that they are only defined between local interfaces. This means that a relationship cannot be created between two entities in separate applications, even in the same application server.

There are two basic steps to create a container managed relationship: create the **cmr-field** abstract accessors and declare the relationship in the `ejb-jar.xml` file. The following two sections describe these steps.

21.5.1. CMR-Field Abstract Accessors

CMR-Field abstract accessors have the same signatures as **cmp-fields**, except that single-valued relationships must return the local interface of the related entity, and multi-valued relationships can only return a `java.util.Collection` (or `java.util.Set`) object. For example, to declare a one-to-many relationship between organization and gangster, we declare the relationship from organization to gangster in the `OrganizationBean` class:

```

public abstract class OrganizationBean
    implements EntityBean
{
    public abstract Set getMemberGangsters();
    public abstract void setMemberGangsters(Set gangsters);
}

```

We also can declare the relationship from gangster to organization in the `GangsterBean` class:

```

public abstract class GangsterBean
    implements EntityBean
{
    public abstract Organization getOrganization();
    public abstract void setOrganization(Organization org);
}

```

Although each bean declared a CMR field, only one of the two beans in a relationship must have a set of accessors. As with CMP fields, a CMR field is required to have both a getter and a setter method.

21.5.2. Relationship Declaration

The declaration of relationships in the `ejb-jar.xml` file is complicated and error prone. Although we recommend using a tool like XDoclet to manage the deployment descriptors for CMR fields, it's still important to understand how the descriptor works. The following illustrates the declaration of the organization/gangster relationship:

```
<ejb-jar>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters </ejb-
relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>OrganizationEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>memberGangsters</cmr-field-name>
          <cmr-field-type>java.util.Set</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          gangster-belongs-to-org
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <cascade-delete/>
        <relationship-role-source>
          <ejb-name>GangsterEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>organization</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</ejb-jar>
```

As you can see, each relationship is declared with an **ejb-relation** element within the top level **relationships** element. The relation is given a name in the **ejb-relation-name** element. This is important because we will need to refer to the role by name in the `jbosscmp-jdbc.xml` file. Each **ejb-relation** contains two **ejb-relationship-role** elements (one for each side of the relationship). The **ejb-relationship-role** tags are as follows:

- **ejb-relationshiprole-name**: This optional element is used to identify the role and match the database mapping the `jbosscmp-jdbc.xml` file. The relationship role names for each side of a relationship must be different.
- **multiplicity**: This indicates the multiplicity of this side of the relationship. The valid values are **One** or **Many**. In this example, the multiplicity of the organization is **One** and the multiplicity of the

gangster is **Many** because the relationship is from one organization to many gangsters. Note, as with all XML elements, this element is case sensitive.

- **cascade-delete**: When this optional element is present, JBoss will delete the child entity when the parent entity is deleted. Cascade deletion is only allowed for a role where the other side of the relationship has a multiplicity of one. The default is to not cascade delete.
- **relationship-role-source**
 - **ejb-name**: This required element gives the name of the entity that has the role.
- **cmr-field**
 - **cmr-field-name**: This is the name of the CMR field of the entity has one, if it has one.
 - **cmr-field-type**: This is the type of the CMR field, if the field is a collection type. It must be `java.util.Collection` or `java.util.Set`.

After adding the CMR field abstract accessors and declaring the relationship, the relationship should be functional. The next section discusses the database mapping of the relationship.

21.5.3. Relationship Mapping

Relationships can be mapped using either a foreign key or a separate relation table. One-to-one and one-to-many relationships use the foreign key mapping style by default, and many-to-many relationships use only the relation table mapping style. The mapping of a relationship is declared in the **relationships** section of the `jbosscmp-jdbc.xml` descriptor via **ejb-relation** elements. Relationships are identified by the **ejb-relation-name** from the `ejb-jar.xml` file. The `jbosscmp-jdbc.xml` **ejb-relation** element content model is shown in Figure 21.7, “The `jbosscmp-jdbc.xml` **ejb-relation** element content model”.

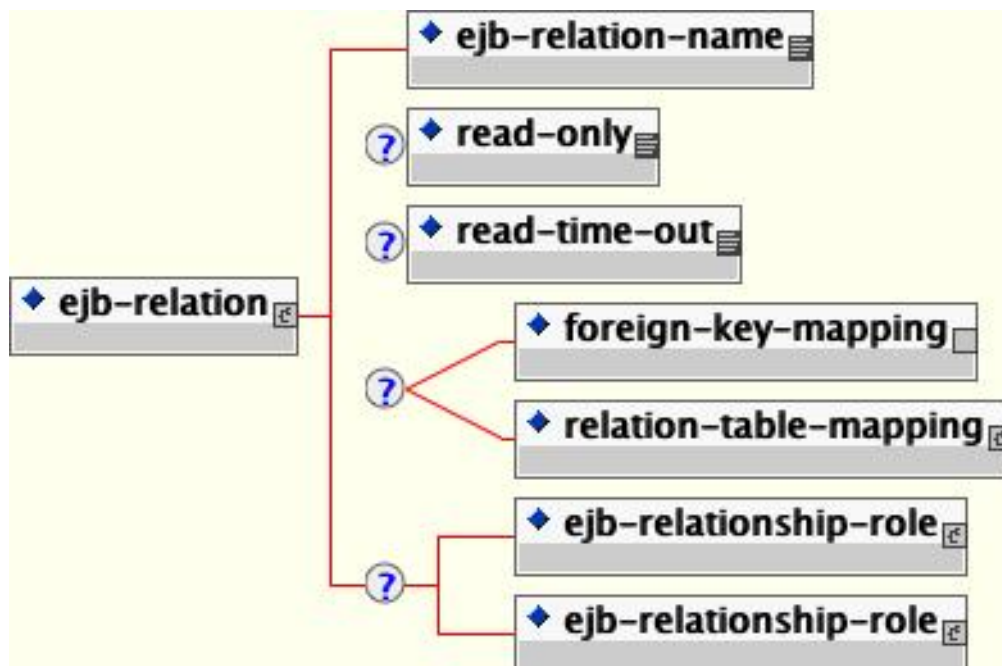


Figure 21.7. The `jbosscmp-jdbc.xml` **ejb-relation** element content model

The basic template of the relationship mapping declaration for **Organization-Gangster** relationship follows:

```

<jbosscmp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <foreign-key-mapping/>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters</ejb-
relationship-role-name>
        <key-fields>
          <key-field>
            <field-name>name</field-name>
            <column-name>organization</column-name>
          </key-field>
        </key-fields>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-belongs-to-org</ejb-
relationship-role-name>
        <key-fields/>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</jbosscmp-jdbc>

```

After the **ejb-relation-name** of the relationship being mapped is declared, the relationship can be declared as read only using the **read-only** and **read-time-out** elements. They have the same semantics as their counterparts in the entity element.

The **ejb-relation** element must contain either a **foreign-key-mapping** element or a **relation-table-mapping** element, which are described in [Section 21.5.3.2, “Foreign Key Mapping”](#) and [Section 21.5.3.3, “Relation table Mapping”](#). This element may also contain a pair of **ejb-relationship-role** elements as described in the following section.

21.5.3.1. Relationship Role Mapping

Each of the two **ejb-relationship-role** elements contains mapping information specific to an entity in the relationship. The content model of the **ejb-relationship-role** element is shown in [Figure 21.8, “The jbosscmp-jdbc ejb-relationship-role element content model”](#).

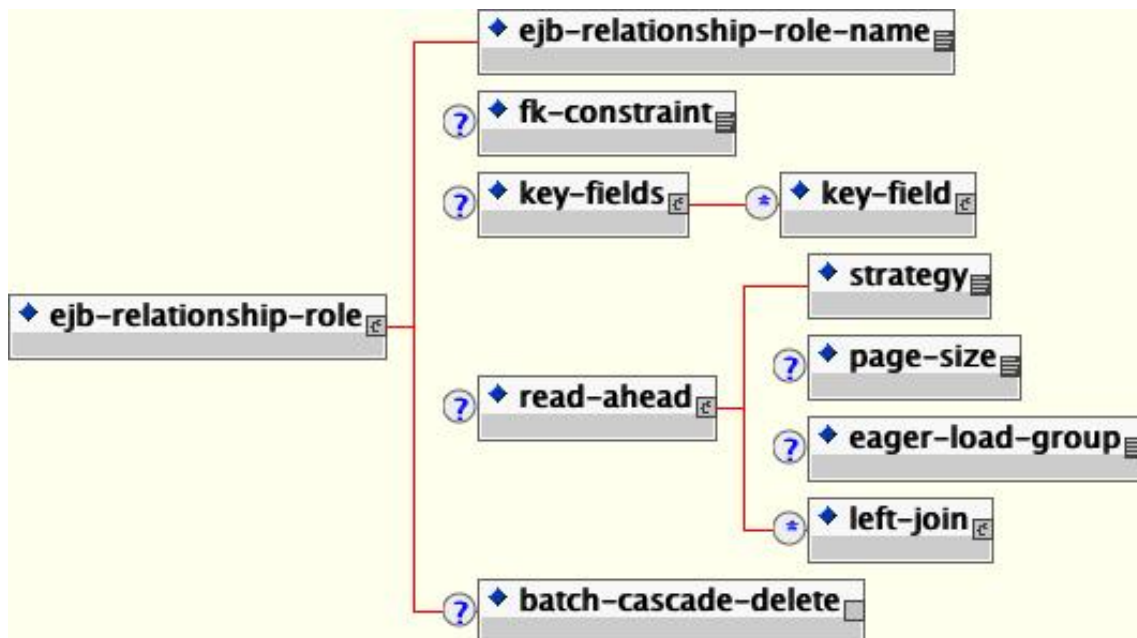


Figure 21.8. The jbossCMP-jdbc ejb-relationship-role element content model

A detailed description of the main elements follows:

- **ejb-relationship-role-name**: This required element gives the name of the role to which this configuration applies. It must match the name of one of the roles declared for this relationship in the `ejb-jar.xml` file.
- **fk-constraint**: This optional element is a true/false value that indicates whether JBoss should add a foreign key constraint to the tables for this side of the relationship. JBoss will only add generate the constraint if both the primary table and the related table were created by JBoss during deployment.
- **key-fields**: This optional element specifies the mapping of the primary key fields of the current entity, whether it is mapped in the relation table or in the related object. The **key-fields** element must contain a **key-field** element for each primary key field of the current entity. The **key-fields** element can be empty if no foreign key mapping is needed for this side of the relation. An example of this would be the many side of a one-to-many relationship. The details of this element are described below.
- **read-ahead**: This optional element controls the caching of this relationship. This option is discussed in [Section 21.8.3.1, “Relationships”](#).
- **batch-cascade-delete**: This indicates that a cascade delete on this relationship should be performed with a single SQL statement. This requires that the relationship be marked as **batch-delete** in the `ejb-jar.xml`.

As noted above, the **key-fields** element contains a **key-field** for each primary key field of the current entity. The **key-field** element uses the same syntax as the **cmp-field** element of the entity, except that **key-field** does not support the **not-null** option. Key fields of a **relation-table** are automatically not null, because they are the primary key of the table. On the other hand, foreign key fields must be nullable by default. This is because the CMP specification requires an insert into the database after the `ejbCreate` method and an update to it after to pick up CMR changes made in `ejbPostCreate`. Since the EJB specification does not allow a relationship to be modified until `ejbPostCreate`, a foreign key will be initially set to null. There is a similar problem with removal. You

can change this insert behavior using the `jboss.xmlinsert-after-ejb-post-create` container configuration flag. The following example illustrates the creation of a new bean configuration that uses `insert-after-ejb-post-create` by default.

```
<jboss>
  <!-- ... -->
  <container-configurations>
    <container-configuration extends="Standard CMP 2.x EntityBean">
      <container-name>INSERT after ejbPostCreate
Container</container-name>
      <insert-after-ejb-post-create>true</insert-after-ejb-post-
create>
    </container-configuration>
  </container-configurations>
</jboss>
```

An alternate means of working around the non-null foreign key issue is to map the foreign key elements onto non-null CMP fields. In this case you simply populate the foreign key fields in `ejbCreate` using the associated CMP field setters.

The content model of the key-fields element is [Figure 21.9, “The jbosscmp-jdbc key-fields element content model”](#).

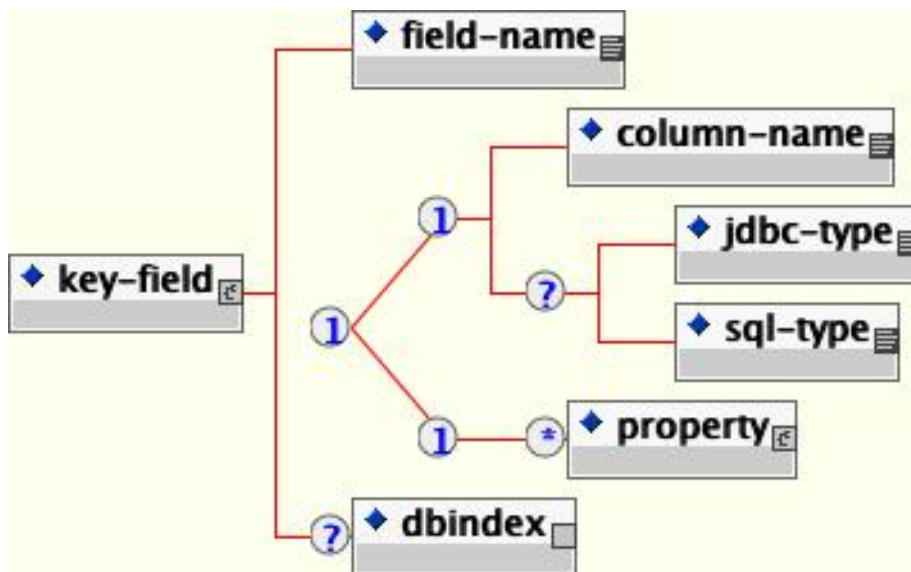


Figure 21.9. The `jbosscmp-jdbc` key-fields element content model

A detailed description of the elements contained in the `key-field` element follows:

- **field-name**: This required element identifies the field to which this mapping applies. This name must match a primary key field of the current entity.
- **column-name**: Use this element to specify the column name in which this primary key field will be stored. If this is relationship uses **foreign-key-mapping**, this column will be added to the table for the related entity. If this relationship uses **relation-table-mapping**, this column is added to the **relation-table**. This element is not allowed for mapped dependent value class; instead use the **property** element.
- **jdbc-type**: This is the JDBC type that is used when setting parameters in a JDBC **PreparedStatement** or loading data from a JDBC **ResultSet**. The valid types are defined in `java.sql.Types`.

- **sql-type**: This is the SQL type that is used in create table statements for this field. Valid types are only limited by your database vendor.
- **property**: Use this element for to specify the mapping of a primary key field which is a dependent value class.
- **dbindex**: The presence of this optional field indicates that the server should create an index on the corresponding column in the database, and the index name will be **fieldname_index**.

21.5.3.2. Foreign Key Mapping

Foreign key mapping is the most common mapping style for one-to-one and one-to-many relationships, but is not allowed for many-to many relationships. The foreign key mapping element is simply declared by adding an empty foreign **key-mapping** element to the **ejb-relation** element.

As noted in the previous section, with a foreign key mapping the **key-fields** declared in the **ejb-relationship-role** are added to the table of the related entity. If the **key-fields** element is empty, a foreign key will not be created for the entity. In a one-to-many relationship, the many side (**Gangster** in the example) must have an empty **key-fields** element, and the one side (**Organization** in the example) must have a **key-fields** mapping. In one-to-one relationships, one or both roles can have foreign keys.

The foreign key mapping is not dependent on the direction of the relationship. This means that in a one-to-one unidirectional relationship (only one side has an accessor) one or both roles can still have foreign keys. The complete foreign key mapping for the **Organization-Gangster** relationship is shown below with the foreign key elements highlighted in bold:

```
<jbosscmp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <foreign-key-mapping/>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters</ejb-
relationship-role-name>
        <key-fields> <key-field> <field-name>name</field-name>
<column-name>organization</column-name> </key-field> </key-fields>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-belongs-to-org</ejb-
relationship-role-name>
        <key-fields/>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</jbosscmp-jdbc>
```

21.5.3.3. Relation table Mapping

Relation table mapping is less common for one-to-one and one-to-many relationships, but is the only mapping style allowed for many-to-many relationships. Relation table mapping is defined using the **relation-table-mapping** element, the content model of which is shown below.

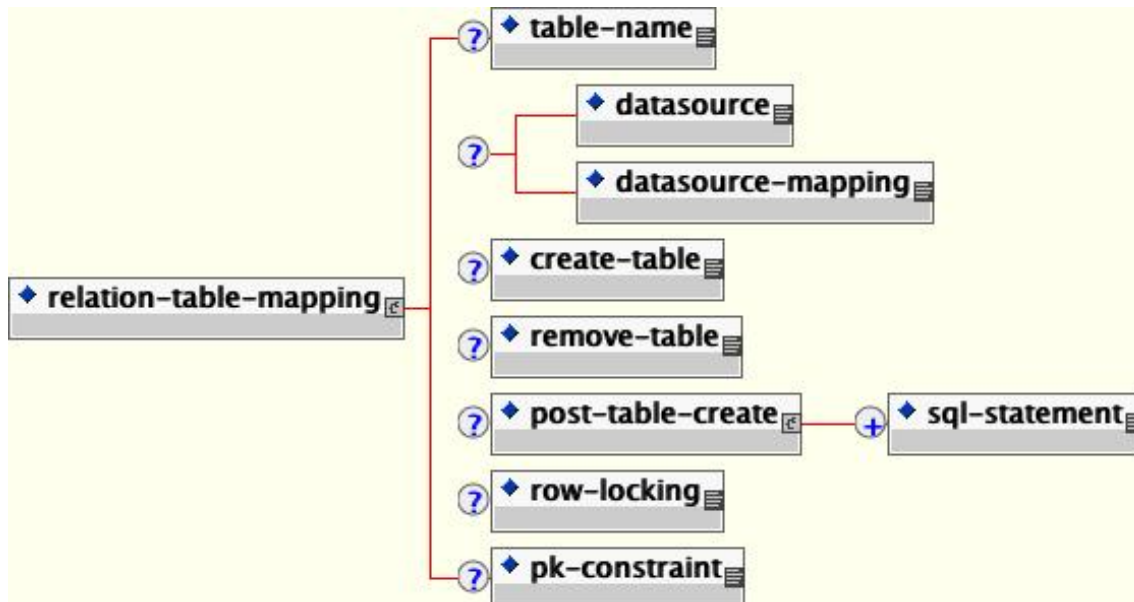


Figure 21.10. The jbosscmp-jdbc relation-table-mapping element content model

The relation-table-mapping for the **Gangster - Job** relationship is shown in with table mapping elements highlighted in bold:

Example 21.1. The jbosscmp-jdbc.xml Relation-table Mapping

```
<jbosscmp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Gangster-Jobs</ejb-relation-name>
      <relation-table-mapping>
        <table-name>gangster_job</table-name>
      </relation-table-mapping>
      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-has-jobs</ejb-
relationship-role-name>
        <key-fields>
          <key-field>
            <field-name>gangsterId</field-name>
            <column-name>gangster</column-name>
          </key-field>
        </key-fields>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>job-has-gangsters</ejb-
relationship-role-name>
        <key-fields>
          <key-field>
            <field-name>name</field-name>
            <column-name>job</column-name>
          </key-field>
        </key-fields>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</jbosscmp-jdbc>
```

The **relation-table-mapping** element contains a subset of the options available in the **entity** element. A detailed description of these elements is reproduced here for convenience:

- **table-name**: This optional element gives the name of the table that will hold data for this relationship. The default table name is based on the entity and **cmr-field** names.
- **datasource**: This optional element gives the **jndi-name** used to look up the datasource. All database connections are obtained from the datasource. Having different datasources for entities is not recommended, as it vastly constrains the domain over which finders and **ejbSelects** can query.
- **datasourcemapping**: This optional element allows one to specify the name of the **type-mapping** to use.
- **create-table**: This optional element if true indicates JBoss should attempt to create a table for the relationship. When the application is deployed, JBoss checks if a table already exists before creating the table. If a table is found, it is logged, and the table is not created. This option is very useful during the early stages of development when the table structure changes often.
- **post-table-create**: This optional element specifies an arbitrary SQL statement that should be executed immediately after the database table is created. This command is only executed if **create-table** is true and the table did not previously exist.
- **remove-table**: This optional element if true indicates JBoss should attempt to drop the **relation-table** when the application is undeployed. This option is very useful during the early stages of development when the table structure changes often.
- **row-locking**: This optional element if true indicates JBoss should lock all rows loaded in a transaction. Most databases implement this by using the **SELECT FOR UPDATE** syntax when loading the entity, but the actual syntax is determined by the **row-locking-template** in the **datasource-mapping** used by this entity.
- **pk-constraint**: This optional element if true indicates JBoss should add a primary key constraint when creating tables.

21.6. QUERIES

Entity beans allow for two types of queries: finders and selects. A finder provides queries on an entity bean to clients of the bean. The select method is designed to provide private query statements to an entity implementation. Unlike finders, which are restricted to only return entities of the same type as the home interface on which they are defined, select methods can return any entity type or just one field of the entity. EJB-QL is the query language used to specify finders and select methods in a platform independent way.

21.6.1. Finder and select Declaration

The declaration of finders has not changed in CMP 2.0. Finders are still declared in the home interface (local or remote) of the entity. Finders defined on the local home interface do not throw a `RemoteException`. The following code declares the **findBadDudes_ejbql** finder on the **GangsterHome** interface. The **ejbql** suffix here is not required. It is simply a naming convention used here to differentiate the different types of query specifications we will be looking at.

```
public interface GangsterHome
    extends EJBLocalHome
```

```
{
    Collection findBadDudes_ejbql(int badness) throws FinderException;
}
```

Select methods are declared in the entity implementation class, and must be public and abstract just like CMP and CMR field abstract accessors and must throw a **FinderException**. The following code declares an select method:

```
public abstract class GangsterBean
    implements EntityBean
{
    public abstract Set ejbSelectBoss_ejbql(String name)
        throws FinderException;
}
```

21.6.2. EJB-QL Declaration

Every select or finder method (except **findByPrimaryKey**) must have an EJB-QL query defined in the **ejb-jar.xml** file. The EJB-QL query is declared in a query element, which is contained in the entity element. The following are the declarations for **findBadDudes_ejbql** and **ejbSelectBoss_ejbql** queries:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <query>
        <query-method>
          <method-name>findBadDudes_ejbql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <ejb-ql><![CDATA[
          SELECT OBJECT(g) FROM gangster g WHERE g.badness > ?1
        ]]></ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>ejbSelectBoss_ejbql</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
        <ejb-ql><![CDATA[
          SELECT DISTINCT underling.organization.theBoss FROM
gangster underling WHERE underling.name = ?1 OR underling.nickName = ?1
        ]]></ejb-ql>
      </query>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

EJB-QL is similar to SQL but has some surprising differences. The following are some important things to note about EJB-QL:

- EJB-QL is a typed language, meaning that it only allows comparison of like types (i.e., strings can only be compared with strings).
- In an equals comparison a variable (single valued path) must be on the left hand side. Some examples follow:

```
g.hangout.state = 'CA' Legal
'CA' = g.shippingAddress.state NOT Legal
'CA' = 'CA' NOT Legal
(r.amountPaid * .01) > 300 NOT Legal
r.amountPaid > (300 / .01) Legal
```

- Parameters use a base 1 index like `java.sql.PreparedStatement`.
- Parameters are only allowed on the right hand side of a comparison. For example:

```
gangster.hangout.state = ?1 Legal
?1 = gangster.hangout.state NOT Legal
```

21.6.3. Overriding the EJB-QL to SQL Mapping

The EJB-QL query can be overridden in the `jbosscmp-jdbc.xml` file. The finder or select is still required to have an EJB-QL declaration, but the `ejb-ql` element can be left empty. Currently the SQL can be overridden with JBossQL, DynamicQL, DeclaredSQL or a BMP style custom `ejbFind` method. All EJB-QL overrides are non-standard extensions to the EJB specification, so use of these extensions will limit portability of your application. All of the EJB-QL overrides, except for BMP custom finders, are declared using a **query** element in the `jbosscmp-jdbc.xml` file. The content model is shown in [Figure 21.11](#), “The `jbosscmp-jdbc` query element content model”.

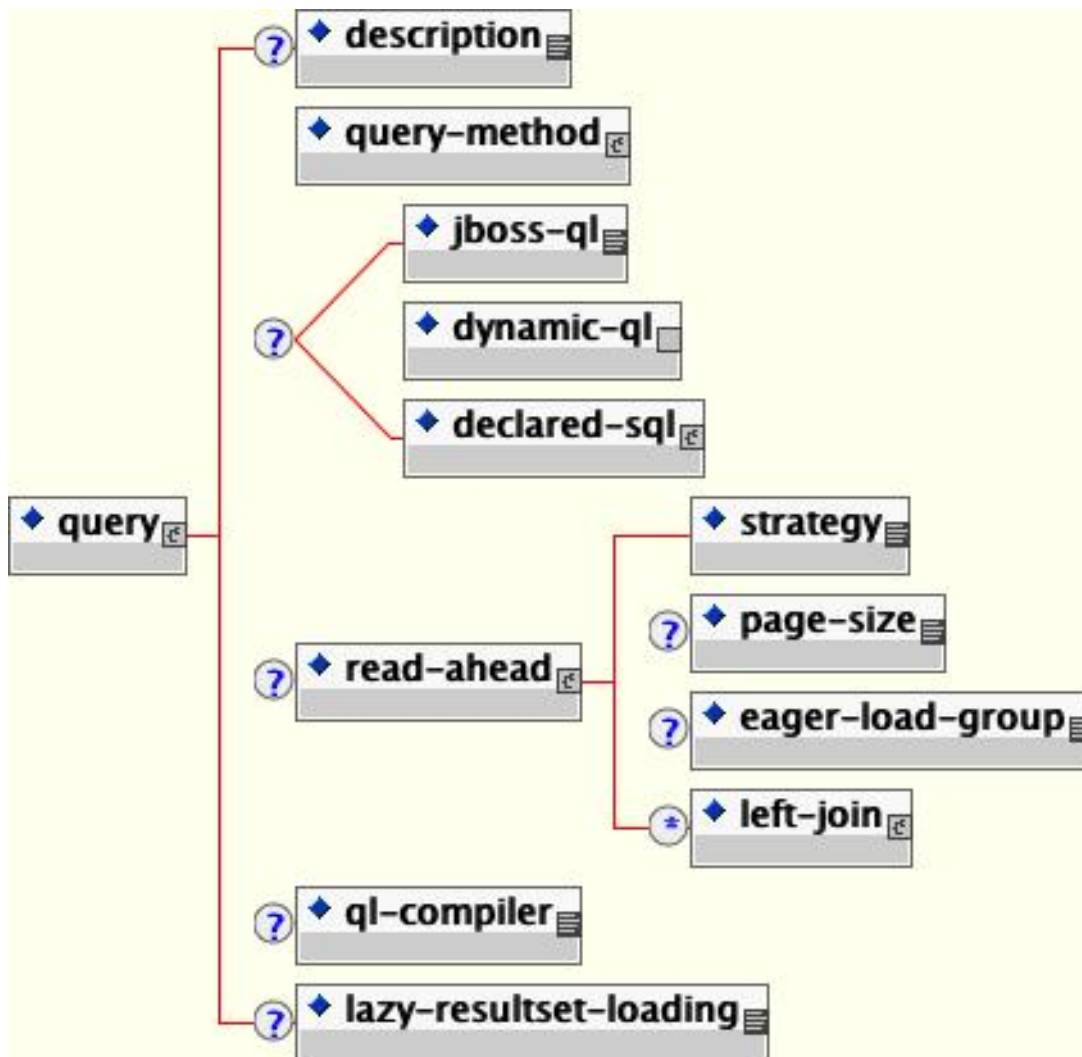


Figure 21.11. The jbosscomp-jdbc query element content model

- **description**: An optional description for the query.
- **query-method**: This required element specifies the query method that being configured. This must match a **query-method** declared for this entity in the **ejb-jar.xml** file.
- **jboss-ql**: This is a JBossQL query to use in place of the EJB-QL query. JBossQL is discussed in [Section 21.6.4, “JBossQL”](#).
- **dynamic-ql**: This indicated that the method is a dynamic query method and not an EJB-QL query. Dynamic queries are discussed in [Section 21.6.5, “DynamicQL”](#).
- **declared-sql**: This query uses declared SQL in place of the EJB-QL query. Declared SQL is discussed in [Section 21.6.6, “DeclaredSQL”](#).
- **read-ahead**: This optional element allows one to optimize the loading of additional fields for use with the entities referenced by the query. This is discussed in detail in [Section 21.7, “Optimized Loading”](#).

21.6.4. JBossQL

JBossQL is a superset of EJB-QL that is designed to address some of the inadequacies of EJB-QL. In addition to a more flexible syntax, new functions, key words, and clauses have been added to JBossQL. At the time of this writing, JBossQL includes support for an **ORDER BY**, **OFFSET** and **LIMIT** clauses,

parameters in the **IN** and **LIKE** operators, the **COUNT**, **MAX**, **MIN**, **AVG**, **SUM**, **UCASE** and **LCASE** functions. Queries can also include functions in the **SELECT** clause for select methods.

JBossQL is declared in the **jbossCMP-jdbc.xml** file with a **jboss-q1** element containing the JBossQL query. The following example provides an example JBossQL declaration.

```
<jbossCMP-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findBadDudes_jbossql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <jboss-q1><![CDATA[ SELECT OBJECT(g) FROM gangster g
WHERE g.badness > ?1 ORDER BY g.badness DESC ]]></jboss-q1>
      </query>
    </entity>
  </enterprise-beans>
</jbossCMP-jdbc>
```

The corresponding generated SQL is straightforward.

```
SELECT t0_g.id
FROM gangster t0_g
WHERE t0_g.badness > ?
ORDER BY t0_g.badness DESC
```

Another capability of JBossQL is the ability to retrieve finder results in blocks using the **LIMIT** and **OFFSET** functions. For example, to iterate through the large number of jobs performed, the following **findManyJobs_jbossql** finder may be defined.

```
<jbossCMP-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findManyJobs_jbossql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <jboss-q1><![CDATA[ SELECT OBJECT(j) FROM jobs j OFFSET ?
1 LIMIT ?2 ]]></jboss-q1>
      </query>
    </entity>
  </enterprise-beans>
</jbossCMP-jdbc>
```

21.6.5. DynamicQL

DynamicQL allows the runtime generation and execution of JBossQL queries. A DynamicQL query method is an abstract method that takes a JBossQL query and the query arguments as parameters. JBoss compiles the JBossQL and executes the generated SQL. The following generates a JBossQL query that selects all the gangsters that have a hangout in any state in the states set:

```
public abstract class GangsterBean
    implements EntityBean
{
    public Set ejbHomeSelectInStates(Set states)
        throws FinderException
    {
        // generate JBossQL query
        StringBuffer jbossQl = new StringBuffer();
        jbossQl.append("SELECT OBJECT(g) ");
        jbossQl.append("FROM gangster g ");
        jbossQl.append("WHERE g.hangout.state IN (");

        for (int i = 0; i < states.size(); i++) {
            if (i > 0) {
                jbossQl.append(", ");
            }

            jbossQl.append("?").append(i+1);
        }

        jbossQl.append(") ORDER BY g.name");

        // pack arguments into an Object[]
        Object[] args = states.toArray(new Object[states.size()]);

        // call dynamic-ql query
        return ejbSelectGeneric(jbossQl.toString(), args);
    }
}
```

The DynamicQL select method may have any valid select method name, but the method must always take a string and an object array as parameters. DynamicQL is declared in the **jbosscomp-jdbc.xml** file with an empty **dynamic-ql** element. The following is the declaration for **ejbSelectGeneric**.

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>ejbSelectGeneric</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.Object[]</method-param>
          </method-params>
        </query-method>
        <dynamic-ql/>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

```

        </query>
    </entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

21.6.6. DeclaredSQL

DeclaredSQL is based on the legacy JAWS CMP 1.1 engine finder declaration, but has been updated for CMP 2.0. Commonly this declaration is used to limit a query with a **WHERE** clause that cannot be represented in q EJB-QL or JBossQL. The content model for the declared-sql element is given in Figure 21.12, “The jbosscomp-jdbc declared-sql element content model.>”.

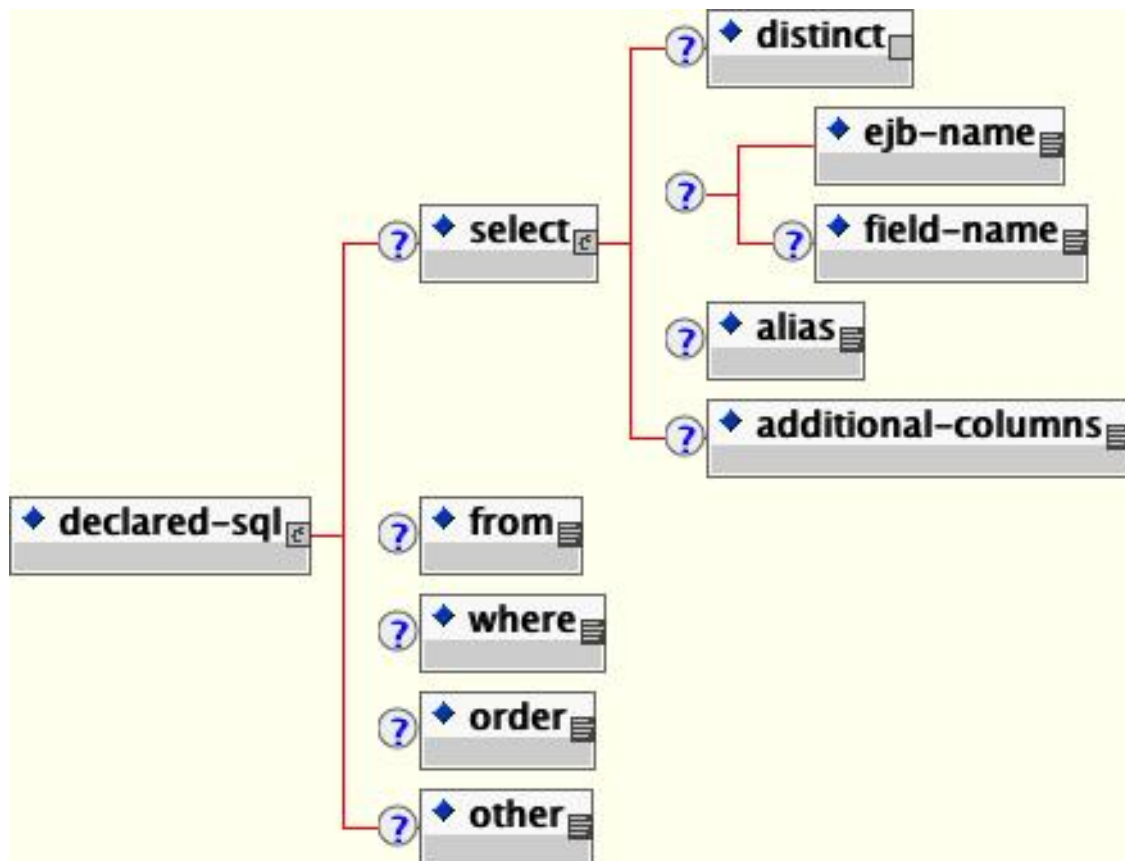


Figure 21.12. The jbosscomp-jdbc declared-sql element content model.>

- **select**: The **select** element specifies what is to be selected and consists of the following elements:
 - **distinct**: If this empty element is present, JBoss will add the **DISTINCT** keyword to the generated **SELECT** clause. The default is to use **DISTINCT** if method returns a `java.util.Set`
 - **ejb-name**: This is the **ejb-name** of the entity that will be selected. This is only required if the query is for a select method.
 - **field-name**: This is the name of the CMP field that will be selected from the specified entity. The default is to select entire entity.
 - **alias**: This specifies the alias that will be used for the main select table. The default is to use the **ejb-name**.

- **additional-columns**: Declares other columns to be selected to satisfy ordering by arbitrary columns with finders or to facilitate aggregate functions in selects.
- **from**: The **from** element declares additional SQL to append to the generated **FROM** clause.
- **where**: The **where** element declares the **WHERE** clause for the query.
- **order**: The **order** element declares the **ORDER** clause for the query.
- **other**: The **other** element declares additional SQL that is appended to the end of the query.

The following is an example DeclaredSQL declaration.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findBadDudes_declaredsql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <declared-sql>
          <where><![CDATA[ badness > {0} ]]></where>
          <order><![CDATA[ badness DESC ]]></order>
        </declared-sql>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

The generated SQL would be:

```
SELECT id
FROM gangster
WHERE badness > ?
ORDER BY badness DESC
```

As you can see, JBoss generates the **SELECT** and **FROM** clauses necessary to select the primary key for this entity. If desired an additional **FROM** clause can be specified that is appended to the end of the automatically generated **FROM** clause. The following is example DeclaredSQL declaration with an additional **FROM** clause.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>ejbSelectBoss_declaredsql</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

```

        </query-method>
        <declared-sql>
            <select>
                <distinct/>
                <ejb-name>GangsterEJB</ejb-name>
                <alias>boss</alias>
            </select>
            <from><![CDATA[, gangster g, organization o]]></from>
            <where><![CDATA[
                (LCASE(g.name) = {0} OR LCASE(g.nick_name) = {0})
AND
                g.organization = o.name AND o.the_boss = boss.id
            ]]></where>
        </declared-sql>
    </query>
</entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

The generated SQL would be:

```

SELECT DISTINCT boss.id
FROM gangster boss, gangster g, organization o
WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?) AND
      g.organization = o.name AND o.the_boss = boss.id

```

Notice that the **FROM** clause starts with a comma. This is because the container appends the declared **FROM** clause to the end of the generated **FROM** clause. It is also possible for the **FROM** clause to start with a SQL **JOIN** statement. Since this is a select method, it must have a **select** element to declare the entity that will be selected. Note that an alias is also declared for the query. If an alias is not declared, the **table-name** is used as the alias, resulting in a **SELECT** clause with the **table_name.field_name** style column declarations. Not all database vendors support the that syntax, so the declaration of an alias is preferred. The optional empty **distinct** element causes the **SELECT** clause to use the **SELECT DISTINCT** declaration. The DeclaredSQL declaration can also be used in select methods to select a CMP field.

Now we will see an example which overrides a select to return all of the zip codes an **Organization** operates in.

```

<jbosscomp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>OrganizationEJB</ejb-name>
            <query>
                <query-method>
                    <method-
name>ejbSelectOperatingZipCodes_declaredsql</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <declared-sql> <select> <distinct/> <ejb-
name>LocationEJB</ejb-name> <field-name>zipCode</field-name>
<alias>hangout</alias> </select> <from><![CDATA[ , organization o,
gangster g ]]></from> <where><![CDATA[ LCASE(o.name) = {0} AND o.name =

```

```

g.organization AND g.hangout = hangout.id ]]></where> <order><![CDATA[
hangout.zip ]]></order> </declared-sql>
    </query>
  </entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

The corresponding SQL would be:

```

SELECT DISTINCT hangout.zip
  FROM location hangout, organization o, gangster g
 WHERE LCASE(o.name) = ? AND o.name = g.organization AND g.hangout =
 hangout.id
        ORDER BY hangout.zip

```

21.6.6.1. Parameters

DeclaredSQL uses a completely new parameter handling system, which supports entity and DVC parameters. Parameters are enclosed in curly brackets and use a zero-based index, which is different from the one-based EJB-QL parameters. There are three categories of parameters: simple, DVC, and entity.

- simple:** A simple parameter can be of any type except for a known (mapped) DVC or an entity. A simple parameter only contains the argument number, such as `{0}`. When a simple parameter is set, the JDBC type used to set the parameter is determined by the **datasourcemapping** for the entity. An unknown DVC is serialized and then set as a parameter. Note that most databases do not support the use of a BLOB value in a WHERE clause.
- DVC:** A DVC parameter can be any known (mapped) DVC. A DVC parameter must be dereferenced down to a simple property (one that is not another DVC). For example, if we had a CVS property of type **ContactInfo**, valid parameter declarations would be `{0.email}` and `{0.cell.areaCode}` but not `{0.cell}`. The JDBC type used to set a parameter is based on the class type of the property and the **datasourcemapping** of the entity. The JDBC type used to set the parameter is the JDBC type that is declared for that property in the **dependent-value-class** element.
- entity:** An entity parameter can be any entity in the application. An entity parameter must be dereferenced down to a simple primary key field or simple property of a DVC primary key field. For example, if we had a parameter of type **Gangster**, a valid parameter declaration would be `{0.gangsterId}`. If we had some entity with a primary key field named `info` of type **ContactInfo**, a **valid parameter** declaration would be `{0.info.cell.areaCode}`. Only fields that are members of the primary key of the entity can be dereferenced (this restriction may be removed in later versions). The JDBC type used to set the parameter is the JDBC type that is declared for that field in the entity declaration.

21.6.7. EJBQL 2.1 and SQL92 queries

The default query compiler doesn't fully support EJB-QL 2.1 or the SQL92 standard. If you need either of these functions, you can replace the query compiler. The default compiler is specified in `standardjbosscomp-jdbc.xml`.

```

<defaults>
  ...
  <ql-compiler>org.jboss.ejb.plugins.comp.jdbc.JDBCEJBQLCompiler</ql-

```

```

compiler>
...
</defaults>

```

To use the SQL92 compiler, simply specify the SQL92 compiler in **ql-compiler** element.

```

<defaults>
...
  <ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.EJBQLToSQL92Compiler</ql-
compiler>
...
</defaults>

```

This changes the query compiler for all beans in the entire system. You can also specify the ql-compiler for each element in **jbosscomp-jdbc.xml**. Here is an example using one of our earlier queries.

```

<query>
  <query-method>
    <method-name>findBadDudes_ejbql</method-name>
    <method-params>
      <method-param>int</method-param>
    </method-params>
  </query-method>
  <ejb-ql><![CDATA[
    SELECT OBJECT(g)
    FROM gangster g
    WHERE g.badness > ?1]]>
  </ejb-ql>
  <ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.EJBQLToSQL92Compiler</ql-
compiler>
</query>

```

One important limitation of SQL92 query compiler is that it always selects all the fields of an entity regardless the **read-ahead** strategy in use. For example, if a query is configured with the **on-loadread-ahead** strategy, the first query will include all the fields, not just primary key fields but only the primary key fields will be read from the **ResultSet**. Then, on load, other fields will be actually loaded into the read-ahead cache. The **on-findread-ahead** with the default load group * works as expected.

21.6.8. BMP Custom Finders

JBoss also supports bean managed persistence custom finders. If a custom finder method matches a finder declared in the home or local home interface, JBoss will always call the custom finder over any other implementation declared in the **ejb-jar.xml** or **jbosscomp-jdbc.xml** files. The following simple example finds the entities by a collection of primary keys:

```

public abstract class GangsterBean
  implements EntityBean
{
  public Collection ejbFindByPrimaryKeys(Collection keys)
  {
    return keys;
  }
}

```

This is a very useful finder because it quickly converts primary keys into real Entity objects without contacting the database. One drawback is that it can create an Entity object with a primary key that does not exist in the database. If any method is invoked on the bad Entity, a `NoSuchEntityException` will be thrown. Another drawback is that the resulting entity bean violates the EJB specification in that it implements a finder, and the JBoss EJB verifier will fail the deployment of such an entity unless the `StrictVerifier` attribute is set to false.

21.7. OPTIMIZED LOADING

The goal of optimized loading is to load the smallest amount of data required to complete a transaction in the fewest number of queries. The tuning of JBoss depends on a detailed knowledge of the loading process. This section describes the internals of the JBoss loading process and its configuration. Tuning of the loading process really requires a holistic understanding of the loading system, so this chapter may have to be read more than once.

21.7.1. Loading Scenario

The easiest way to investigate the loading process is to look at a usage scenario. The most common scenario is to locate a collection of entities and iterate over the results performing some operation. The following example generates an html table containing all of the gangsters:

```
public String createGangsterHtmlTable_none()
    throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");

    Collection gangsters = gangsterHome.findAll_none();
    for (Iterator iter = gangsters.iterator(); iter.hasNext();) {
        Gangster gangster = (Gangster) iter.next();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
        table.append("</tr>");
    }

    return table.toString();
}
```

Assume this code is called within a single transaction and all optimized loading has been disabled. At the **`findAll_none`** call, JBoss will execute the following query:

```
SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

Then as each of the eight gangster in the sample database is accessed, JBoss will execute the following eight queries:

```
SELECT name, nick_name, badness, hangout, organization
```



```

FROM gangster WHERE (id=0)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=1)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=2)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=3)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=4)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=5)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=6)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=7)

```

There are two problems with this scenario. First, an excessive number of queries are executed because JBoss executes one query for the **findAll** and one query to access each element found. The reason for this behavior has to do with the handling of query results inside the JBoss container. Although it appears that the actual entity beans selected are returned when a query is executed, JBoss really only returns the primary keys of the matching entities, and does not load the entity until a method is invoked on it. This is known as the *n+1* problem and is addressed with the read-ahead strategies described in the following sections.

Second, the values of unused fields are loaded needlessly. JBoss loads the **hangout** and **organization** fields, which are never accessed. (we have disabled the complex **contactInfo** field for the sake of clarity)

The following table shows the execution of the queries:

Table 21.1. Unoptimized Query Execution

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

21.7.2. Load Groups

The configuration and optimization of the loading system begins with the declaration of named load groups in the entity. A load group contains the names of CMP fields and CMR Fields that have a foreign key (e.g., **Gangster** in the Organization-Gangster example) that will be loaded in a single operation. An example configuration is shown below:

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <load-groups>
        <load-group>
          <load-group-name>basic</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
        </load-group>
        <load-group>
          <load-group-name>contact info</load-group-name>
          <field-name>nickName</field-name>
          <field-name>contactInfo</field-name>
          <field-name>hangout</field-name>
        </load-group>
      </load-groups>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

In this example, two load groups are declared: **basic** and **contact info**. Note that the load groups do not need to be mutually exclusive. For example, both of the load groups contain the **nickName** field. In addition to the declared load groups, JBoss automatically adds a group named ***** (the star group) that contains every CMP field and CMR field with a foreign key in the entity.

21.7.3. Read-ahead

Optimized loading in JBoss is called read-ahead. This refers to the technique of reading the row for an entity being loaded, as well as the next several rows; hence the term read-ahead. JBoss implements two main strategies (**on-find** and **on-load**) to optimize the loading problem identified in the previous section. The extra data loaded during read-ahead is not immediately associated with an entity object in memory, as entities are not materialized in JBoss until actually accessed. Instead, it is stored in the preload cache where it remains until it is loaded into an entity or the end of the transaction occurs. The following sections describe the read-ahead strategies.

21.7.3.1. on-find

The **on-find** strategy reads additional columns when the query is invoked. If the query is **on-find** optimized, JBoss will execute the following query when the query is executed.

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

All of the required data would be in the preload cache, so no additional queries would need to be executed while iterating through the query results. This strategy is effective for queries that return a small

amount of data, but it becomes very inefficient when trying to load a large result set into memory. The following table shows the execution of this query:

Table 21.2. on-find Optimized Query Execution

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

The **read-ahead** strategy and **load-group** for a query is defined in the **query** element. If a **read-ahead** strategy is not declared in the **query** element, the strategy declared in the **entity** element or **defaults** element is used. The **on-find** configuration follows:

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!--...-->
      <query>
        <query-method>
          <method-name>findAll_onfind</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
        <read-ahead>
          <strategy>on-find</strategy>
          <page-size>4</page-size>
          <eager-load-group>basic</eager-load-group>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

One problem with the **on-find** strategy is that it must load additional data for every entity selected. Commonly in web applications only a fixed number of results are rendered on a page. Since the preloaded data is only valid for the length of the transaction, and a transaction is limited to a single web HTTP hit, most of the preloaded data is not used. The **on-load** strategy discussed in the next section does not suffer from this problem.

21.7.3.1.1. Left join read ahead

Left join read ahead is an enhanced **on-findread-ahead** strategy. It allows you to preload in one SQL query not only fields from the base instance but also related instances which can be reached from the base instance by CMR navigation. There are no limitation for the depth of CMR navigations. There are also no limitations for cardinality of CMR fields used in navigation and relationship type mapping, i.e. both foreign key and relation-table mapping styles are supported. Let's look at some examples. Entity and relationship declarations can be found below.

21.7.3.1.2. D#findByPrimaryKey

Suppose we have an entity **D**. A typical SQL query generated for the **findByPrimaryKey** would look like this:

```
SELECT t0_D.id, t0_D.name FROM D t0_D WHERE t0_D.id=?
```

Suppose that while executing **findByPrimaryKey** we also want to preload two collection-valued CMR fields **bs** and **cs**.

```
<query>
  <query-method>
    <method-name>findByPrimaryKey</method-name>
    <method-params>
      <method-param>java.lang.Long</method-param>
    </method-params>
  </query-method>
  <jboss-ql><![CDATA[SELECT OBJECT(o) FROM D AS o WHERE o.id = ?1]]>
</jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>basic</eager-load-group>
    <left-join cmr-field="bs" eager-load-group="basic"/>
    <left-join cmr-field="cs" eager-load-group="basic"/>
  </read-ahead>
</query>
```

The **left-join** declares the relations to be eager loaded. The generated SQL would look like this:

```
SELECT t0_D.id, t0_D.name,
       t1_D_bs.id, t1_D_bs.name,
       t2_D_cs.id, t2_D_cs.name
FROM D t0_D
     LEFT OUTER JOIN B t1_D_bs ON t0_D.id=t1_D_bs.D_FK
     LEFT OUTER JOIN C t2_D_cs ON t0_D.id=t2_D_cs.D_FK
WHERE t0_D.id=?
```

For the **D** with the specific id we preload all its related **B**'s and **C**'s and can access those instance loading them from the read ahead cache, not from the database.

21.7.3.1.3. D#findAll

In the same way, we could optimize the **findAll** method on **D** selects all the **D**'s. A normal **findAll** query would look like this:

```
SELECT DISTINCT t0_o.id, t0_o.name FROM D t0_o ORDER BY t0_o.id DESC
```

To preload the relations, we simply need to add the **left-join** elements to the query.

```
<query>
  <query-method>
    <method-name>findAll</method-name>
  </query-method>
  <jboss-ql><![CDATA[SELECT DISTINCT OBJECT(o) FROM D AS o ORDER BY o.id
DESC]]></jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>basic</eager-load-group>
    <left-join cmr-field="bs" eager-load-group="basic"/>
    <left-join cmr-field="cs" eager-load-group="basic"/>
  </read-ahead>
</query>
```

And here is the generated SQL:

```
SELECT DISTINCT t0_o.id, t0_o.name,
                t1_o_bs.id, t1_o_bs.name,
                t2_o_cs.id, t2_o_cs.name
FROM D t0_o
  LEFT OUTER JOIN B t1_o_bs ON t0_o.id=t1_o_bs.D_FK
  LEFT OUTER JOIN C t2_o_cs ON t0_o.id=t2_o_cs.D_FK
ORDER BY t0_o.id DESC
```

Now the simple **findAll** query now preloads the related **B** and **C** objects for each **D** object.

21.7.3.1.4. A#findAll

Now let's look at a more complex configuration. Here we want to preload instance **A** along with several relations.

- its parent (self-relation) reached from **A** with CMR field **parent**
- the **B** reached from **A** with CMR field **b**, and the related **C** reached from **B** with CMR field **c**
- **B** reached from **A** but this time with CMR field **b2** and related to it **C** reached from **B** with CMR field **c**.

For reference, the standard query would be:

```
SELECT t0_o.id, t0_o.name FROM A t0_o ORDER BY t0_o.id DESC FOR UPDATE
```

The following metadata describes our preloading plan.

```
<query>
  <query-method>
    <method-name>findAll</method-name>
  </query-method>
  <jboss-ql><![CDATA[SELECT OBJECT(o) FROM A AS o ORDER BY o.id DESC]]>
</jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>basic</eager-load-group>
    <left-join cmr-field="parent" eager-load-group="basic"/>
    <left-join cmr-field="b" eager-load-group="basic">
      <left-join cmr-field="c" eager-load-group="basic"/>
    </left-join>
    <left-join cmr-field="b2" eager-load-group="basic">
      <left-join cmr-field="c" eager-load-group="basic"/>
    </left-join>
  </read-ahead>
</query>
```

The SQL query generated would be:

```
SELECT t0_o.id, t0_o.name,
       t1_o_parent.id, t1_o_parent.name,
       t2_o_b.id, t2_o_b.name,
       t3_o_b_c.id, t3_o_b_c.name,
       t4_o_b2.id, t4_o_b2.name,
       t5_o_b2_c.id, t5_o_b2_c.name
FROM A t0_o
LEFT OUTER JOIN A t1_o_parent ON t0_o.PARENT=t1_o_parent.id
LEFT OUTER JOIN B t2_o_b ON t0_o.B_FK=t2_o_b.id
LEFT OUTER JOIN C t3_o_b_c ON t2_o_b.C_FK=t3_o_b_c.id
LEFT OUTER JOIN B t4_o_b2 ON t0_o.B2_FK=t4_o_b2.id
LEFT OUTER JOIN C t5_o_b2_c ON t4_o_b2.C_FK=t5_o_b2_c.id
ORDER BY t0_o.id DESC FOR UPDATE
```

With this configuration, you can navigate CMRs from any found instance of **A** without an additional database load.

21.7.3.1.5. A#findMeParentGrandParent

Here is another example of self-relation. Suppose, we want to write a method that would preload an instance, its parent, grand-parent and its grand-grand-parent in one query. To do this, we would used nested **left-join** declaration.

```
<query>
  <query-method>
    <method-name>findMeParentGrandParent</method-name>
    <method-params>
      <method-param>java.lang.Long</method-param>
    </method-params>
  </query-method>
  <jboss-ql><![CDATA[SELECT OBJECT(o) FROM A AS o WHERE o.id = ?1]]>
```

```

</jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>*</eager-load-group>
    <left-join cmr-field="parent" eager-load-group="basic">
      <left-join cmr-field="parent" eager-load-group="basic">
        <left-join cmr-field="parent" eager-load-group="basic"/>
      </left-join>
    </left-join>
  </read-ahead>
</query>

```

The generated SQL would be:

```

SELECT t0_o.id, t0_o.name, t0_o.secondName, t0_o.B_FK, t0_o.B2_FK,
t0_o.PARENT,
       t1_o_parent.id, t1_o_parent.name,
       t2_o_parent_parent.id, t2_o_parent_parent.name,
       t3_o_parent_parent_parent.id, t3_o_parent_parent_parent.name
FROM A t0_o
      LEFT OUTER JOIN A t1_o_parent ON t0_o.PARENT=t1_o_parent.id
      LEFT OUTER JOIN A t2_o_parent_parent ON
t1_o_parent.PARENT=t2_o_parent_parent.id
      LEFT OUTER JOIN A t3_o_parent_parent_parent
      ON t2_o_parent_parent.PARENT=t3_o_parent_parent_parent.id
WHERE (t0_o.id = ?) FOR UPDATE

```

Note, if we remove **left-join** metadata we will have only

```

SELECT t0_o.id, t0_o.name, t0_o.secondName, t0_o.B2_FK, t0_o.PARENT FOR
UPDATE

```

21.7.3.2. on-load

The **on-load** strategy block-loads additional data for several entities when an entity is loaded, starting with the requested entity and the next several entities in the order they were selected. This strategy is based on the theory that the results of a find or select will be accessed in forward order. When a query is executed, JBoss stores the order of the entities found in the list cache. Later, when one of the entities is loaded, JBoss uses this list to determine the block of entities to load. The number of lists stored in the cache is specified with the **list-cachemax** element of the entity. This strategy is also used when faulting in data not loaded in the **on-find** strategy.

As with the **on-find** strategy, **on-load** is declared in the **read-ahead** element. The **on-load** configuration for this example is shown below.

```

<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
    <query>
      <query-method>
        <method-name>findAll_onload</method-name>
      </query-method>
    </query>
  </enterprise-beans>
</jbosscmp-jdbc>

```

```

        <method-params/>
      </query-method>
    <jboss-ql><![CDATA[
      SELECT OBJECT(g)
      FROM gangster g
      ORDER BY g.gangsterId
    ]]></jboss-ql>
    <read-ahead>
      <strategy>on-load</strategy>
      <page-size>4</page-size>
      <eager-load-group>basic</eager-load-group>
    </read-ahead>
  </query>
</entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

With this strategy, the query for the finder method in remains unchanged.

```

SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

However, the data will be loaded differently as we iterate through the result set. For a page size of four, JBoss will only need to execute the following two queries to load the **name**, **nickName** and **badness** fields for the entities:

```

SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=4) OR (id=5) OR (id=6) OR (id=7)

```

The following table shows the execution of these queries:

Table 21.3. on-load Optimized Query Execution

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia

id	name	nick_name	badness	hangout	organization
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

21.7.3.3. none

The **none** strategy is really an anti-strategy. This strategy causes the system to fall back to the default lazy-load code, and specifically does not read-ahead any data or remember the order of the found entities. This results in the queries and performance shown at the beginning of this chapter. The none strategy is declared with a read-ahead element. If the **read-ahead** element contains a **page-size** element or **eager-load-group**, it is ignored. The none strategy is declared the following example.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <query>
        <query-method>
          <method-name>findAll_none</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
        <read-ahead>
          <strategy>none</strategy>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

21.8. LOADING PROCESS

In the previous section several steps use the phrase "when the entity is loaded." This was intentionally left vague because the commit option specified for the entity and the current state of the transaction determine when an entity is loaded. The following section describes the commit options and the loading processes.

21.8.1. Commit Options

Central to the loading process are the commit options, which control when the data for an entity expires. JBoss supports four commit options **A**, **B**, **C** and **D**. The first three are described in the Enterprise JavaBeans Specification, but the last one is specific to JBoss. A detailed description of each commit option follows:

- **A**: JBoss assumes it is the sole user of the database; therefore, JBoss can cache the current

value of an entity between transactions, which can result in substantial performance gains. As a result of this assumption, no data managed by JBoss can be changed outside of JBoss. For example, changing data in another program or with the use of direct JDBC (even within JBoss) will result in an inconsistent database state.

- **B:** JBoss assumes that there is more than one user of the database but keeps the context information about entities between transactions. This context information is used for optimizing loading of the entity. This is the default commit option.
- **C:** JBoss discards all entity context information at the end of the transaction.
- **D:** This is a JBoss specific commit option. This option is similar to commit option **A**, except that the data only remains valid for a specified amount of time.

The commit option is declared in the `jboss.xml` file. For a detailed description of this file see [Chapter 20, EJBs on JBoss](#). The following example changes the commit option to **A** for all entity beans in the application:

Example 21.2. The jboss.xml Commit Option Declaration

```
<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Standard CMP 2.x EntityBean</container-
name>
      <commit-option>A</commit-option>
    </container-configuration>
  </container-configurations>
</jboss>
```

21.8.2. Eager-loading Process

When an entity is loaded, JBoss must determine the fields that need to be loaded. By default, JBoss will use the **eager-load-group** of the last query that selected this entity. If the entity has not been selected in a query, or the last query used the **none** read-ahead strategy, JBoss will use the default **eager-load-group** declared for the entity. In the following example configuration, the **basic** load group is set as the default **eager-load-group** for the gangster entity bean:

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <load-groups>
        <load-group>
          <load-group-name>most</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
          <field-name>hangout</field-name>
          <field-name>organization</field-name>
        </load-group>
      </load-groups>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

```

        <eager-load-group>most</eager-load-group>
      </entity>
    </enterprise-beans>
  </jbosscmp-jdbc>

```

The eager loading process is initiated the first time a method is called on an entity in a transaction. A detailed description of the load process follows:

1. If the entity context is still valid, no loading is necessary, and therefore the loading process is done. The entity context will be valid when using commit option **A**, or when using commit option **D**, and the data has not timed out.
2. Any residual data in the entity context is flushed. This assures that old data does not bleed into the new load.
3. The primary key value is injected back into the primary key fields. The primary key object is actually independent of the fields and needs to be reloaded after the flush in step 2.
4. All data in the preload cache for this entity is loaded into the fields.
5. JBoss determines the additional fields that still need to be loaded. Normally the fields to load are determined by the eager-load group of the entity, but can be overridden if the entity was located using a query or CMR field with an **on-find** or **on-load** read ahead strategy. If all of the fields have already been loaded, the load process skips to step 7.
6. A query is executed to select the necessary column. If this entity is using the **on-load** strategy, a page of data is loaded as described in [Section 21.7.3.2, “on-load”](#). The data for the current entity is stored in the context and the data for the other entities is stored in the preload cache.
7. The **ejbLoad** method of the entity is called.

21.8.3. Lazy loading Process

Lazy loading is the other half of eager loading. If a field is not eager loaded, it must be lazy loaded. When an access to an unloaded field of a bean is made, JBoss loads the field and all the fields of any **lazy-load-group** the field belong to. JBoss performs a set join and then removes any field that is already loaded. An example configuration is shown below.

```

<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <load-groups>
        <load-group>
          <load-group-name>basic</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
        </load-group>
        <load-group>
          <load-group-name>contact info</load-group-name>
          <field-name>nickName</field-name>
          <field-name>contactInfo</field-name>
          <field-name>hangout</field-name>
        </load-group>
      </load-groups>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>

```

```

        </load-groups>
        <!-- ... -->
        <lazy-load-groups>
            <load-group-name>basic</load-group-name>
            <load-group-name>contact info</load-group-name>
        </lazy-load-groups>
    </entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

When the bean provider calls `getName()` with this configuration, JBoss loads **name**, **nickName** and **badness**, assuming they are not already loaded. When the bean provider calls `getNickName()`, the **name**, **nickName**, **badness**, **contactInfo**, and **hangout** are loaded. A detailed description of the lazy loading process follows:

1. All data in the preload cache for this entity is loaded into the fields.
2. If the field value was loaded by the preload cache the lazy load process is finished.
3. JBoss finds all of the lazy load groups that contain this field, performs a set join on the groups, and removes any field that has already been loaded.
4. A query is executed to select the necessary columns. As in the basic load process, JBoss may load a block of entities. The data for the current entity is stored in the context and the data for the other entities is stored in the preload cache.

21.8.3.1. Relationships

Relationships are a special case in lazy loading because a CMR field is both a field and query. As a field it can be **on-load** block loaded, meaning the value of the currently sought entity and the values of the CMR field for the next several entities are loaded. As a query, the field values of the related entity can be preloaded using **on-find**.

Again, the easiest way to investigate the loading is to look at a usage scenario. In this example, an HTML table is generated containing each gangster and their hangout. The example code follows:

Example 21.3. Relationship Lazy Loading Example Code

```

public String createGangsterHangoutHtmlTable()
    throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");
    Collection gangsters = gangsterHome.findAll_onfind();
    for (Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
        Gangster gangster = (Gangster)iter.next();

        Location hangout = gangster.getHangout();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
    }
}

```

```

        table.append("<td>").append(hangout.getCity());
        table.append("</td>");
        table.append("<td>").append(hangout.getState());
        table.append("</td>");
        table.append("<td>").append(hangout.getZipCode());
        table.append("</td>");
        table.append("</tr>");
    }

    table.append("</table>");return table.toString();
}

```

For this example, the configuration of the gangster's **findAll_onfind** query is unchanged from the **on-find** section. The configuration of the **Location** entity and **Gangster-Hangout** relationship follows:

Example 21.4. The jbosscmp-jdbc.xml Relationship Lazy Loading Configuration

```

<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <load-groups>
        <load-group>
          <load-group-name>quick info</load-group-name>
          <field-name>city</field-name>
          <field-name>state</field-name>
          <field-name>zipCode</field-name>
        </load-group>
      </load-groups>
      <eager-load-group/>
    </entity>
  </enterprise-beans>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Gangster-Hangout</ejb-relation-name>
      <foreign-key-mapping/>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          gangster-has-a-hangout
        </ejb-relationship-role-name>
        <key-fields/>
        <read-ahead>
          <strategy>on-find</strategy>
          <page-size>4</page-size>
          <eager-load-group>quick info</eager-load-group>
        </read-ahead>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          hangout-for-a-gangster
        </ejb-relationship-role-name>
        <key-fields>
          <key-field>

```

```

        <field-name>locationID</field-name>
        <column-name>hangout</column-name>
    </key-field>
</key-fields>
</ejb-relationship-role>
</ejb-relation>
</relationships>
</jboss-cmp-jdbc>

```

JBoss will execute the following query for the finder:

```

SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

Then when the hangout is accessed, JBoss executes the following two queries to load the **city**, **state**, and **zip** fields of the hangout:

```

SELECT gangster.id, gangster.hangout,
       location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
      ((gangster.id=0) OR (gangster.id=1) OR
       (gangster.id=2) OR (gangster.id=3))
SELECT gangster.id, gangster.hangout,
       location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
      ((gangster.id=4) OR (gangster.id=5) OR
       (gangster.id=6) OR (gangster.id=7))

```

The following table shows the execution of the queries:

Table 21.4. on-find Optimized Relationship Query Execution

id	name	nick_name	badness	hangout	id	city	st	zip
0	Yojimbo	Bodyguard	7	0	0	San Fran	CA	94108
1	Takeshi	Master	10	1	1	San Fran	CA	94133
2	Yuriko	Four finger	4	2	2	San Fran	CA	94133
3	Chow	Killer	9	3	3	San Fran	CA	94133

id	name	nick_name	badness	hangout	id	city	st	zip
4	Shogi	Lightning	8	4	4	San Fran	CA	94133
5	Valentino	Pizza-Face	4	5	5	New York	NY	10017
6	Toni	Toothless	2	6	6	Chicago	IL	60661
7	Corleone	Godfather	6	7	7	Las Vegas	NV	89109

21.8.4. Lazy loading result sets

By default, when a multi-object finder or select method is executed the JDBC result set is read to the end immediately. The client receives a collection of **EJBLocalObject** or CMP field values which it can then iterate through. For big result sets this approach is not efficient. In some cases it is better to delay reading the next row in the result set until the client tries to read the corresponding value from the collection. You can get this behavior for a query using the **lazy-resultset-loading** element.

```
<query>
  <query-method>
    <method-name>findAll</method-name>
  </query-method>
  <jboss-ql><![CDATA[select object(o) from A o]]></jboss-ql>
  <lazy-resultset-loading>true</lazy-resultset-loading>
</query>
```

There are some issues you should be aware of when using lazy result set loading. Special care should be taken when working with a **Collection** associated with a lazily loaded result set. The first call to **iterator()** returns a special **Iterator** that reads from the **ResultSet**. Until this **Iterator** has been exhausted, subsequent calls to **iterator()** or calls to the **add()** method will result in an exception. The **remove()** and **size()** methods work as would be expected.

21.9. TRANSACTIONS

All of the examples presented in this chapter have been defined to run in a transaction. Transaction granularity is a dominating factor in optimized loading because transactions define the lifetime of preloaded data. If the transaction completes, commits, or rolls back, the data in the preload cache is lost. This can result in a severe negative performance impact.

The performance impact of running without a transaction will be demonstrated with an example that uses an **on-find** optimized query that selects the first four gangsters (to keep the result set small), and it is executed without a wrapper transaction. The example code follows:

```
public String createGangsterHtmlTable_no_tx() throws FinderException
{
    StringBuffer table = new StringBuffer();
```

```

        table.append("<table>");

        Collection gangsters = gangsterHome.findFour();
        for(Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
            Gangster gangster = (Gangster)iter.next();
            table.append("<tr>");
            table.append("<td>").append(gangster.getName());
            table.append("</td>");
            table.append("<td>").append(gangster.getNickName());
            table.append("</td>");
            table.append("<td>").append(gangster.getBadness());
            table.append("</td>");
            table.append("</tr>");
        }

        table.append("</table>");
        return table.toString();
    }

```

The finder results in the following query being executed:

```

SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
WHERE t0_g.id < 4
ORDER BY t0_g.id ASC

```

Normally this would be the only query executed, but since this code is not running in a transaction, all of the preloaded data is thrown away as soon as finder returns. Then when the CMP field is accessed JBoss executes the following four queries (one for each loop):

```

SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=2) OR (id=3)
SELECT name, nick_name, badness
FROM gangster
WHERE (id=3)

```

It's actually worse than this. JBoss executes each of these queries three times; once for each CMP field that is accessed. This is because the preloaded values are discarded between the CMP field accessor calls.

The following figure shows the execution of the queries:

id§	name§	nick name§	badness§
0§	Yojimbo	Bodyguard	7
1§	Takeshi	Master	10
2§	Yuriko	Four finger	4
3§	Chow	Killer	9

Figure 21.13. No Transaction on-find optimized query execution

This performance is much worse than read ahead none because of the amount of data loaded from the database. The number of rows loaded is determined by the following equation:

$$n + n - 1 + n - 2 + \dots + 1 + = \frac{n * (n + 1)}{2} = O(n^2)$$

This all happens because the transaction in the example is bounded by a single call on the entity. This brings up the important question "How do I run my code in a transaction?" The answer depends on where the code runs. If it runs in an EJB (session, entity, or message driven), the method must be marked with the **Required** or **RequiresNewtrans-attribute** in the **assembly-descriptor**. If the code is not running in an EJB, a user transaction is necessary. The following code wraps a call to the declared method with a user transaction:

```
public String createGangsterHtmlTable_with_tx()
    throws FinderException
{
    UserTransaction tx = null;
    try {
        InitialContext ctx = new InitialContext();
        tx = (UserTransaction) ctx.lookup("UserTransaction");
        tx.begin();

        String table = createGangsterHtmlTable_no_tx();

        if (tx.getStatus() == Status.STATUS_ACTIVE) {
            tx.commit();
        }
        return table;
    } catch (Exception e) {
        try {
            if (tx != null) tx.rollback();
        } catch (SystemException unused) {
            // eat the exception we are exceptioning out anyway
        }
        if (e instanceof FinderException) {
            throw (FinderException) e;
        }
        if (e instanceof RuntimeException) {
            throw (RuntimeException) e;
        }
        throw new EJBException(e);
    }
}
```

```

    }
}

```

21.10. OPTIMISTIC LOCKING

JBoss has supports for optimistic locking of entity beans. Optimistic locking allows multiple instances of the same entity bean to be active simultaneously. Consistency is enforced based on the optimistic locking policy choice. The optimistic locking policy choice defines the set of fields that are used in the commit time write of modified data to the database. The optimistic consistency check asserts that the values of the chosen set of fields has the same values in the database as existed when the current transaction was started. This is done using a **select for UPDATE WHERE ...** statement that contains the value assertions.

You specify the optimistic locking policy choice using an **optimistic-locking** element in the **jbosscmp-jdbc.xml** descriptor. The content model of the **optimistic-locking** element is shown below and the description of the elements follows.

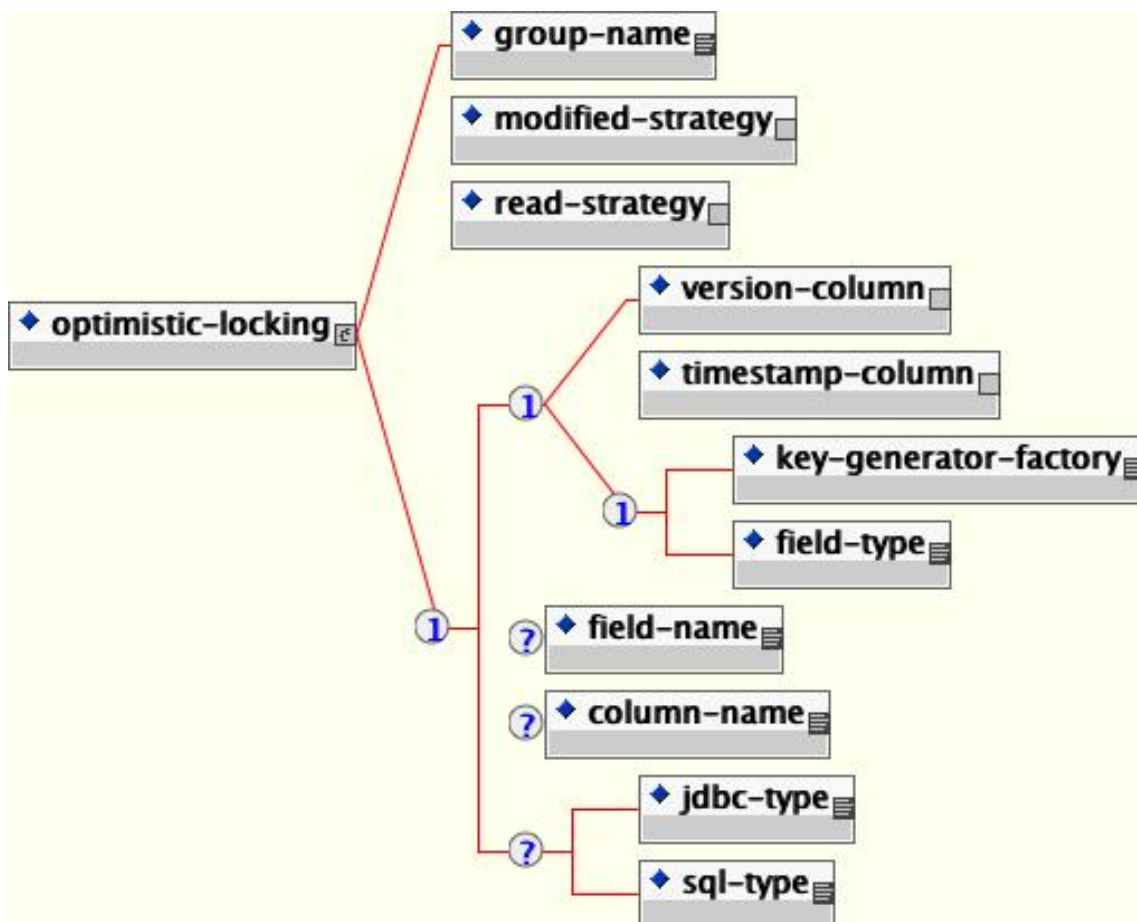


Figure 21.14. The jbosscmp-jdbc optimistic-locking element content model

- **group-name**: This element specifies that optimistic locking is based on the fields of a **load-group**. This value of this element must match one of the entity's **load-group-name**. The fields in this group will be used for optimistic locking.
- **modified-strategy**: This element specifies that optimistic locking is based on the modified fields. This strategy implies that the fields that were modified during transaction will be used for optimistic locking.

- **read-strategy**: This element specifies that optimistic locking is based on the fields read. This strategy implies that the fields that were read/changed in the transaction will be used for optimistic locking.
- **version-column**: This element specifies that optimistic locking is based on a version column strategy. Specifying this element will add an additional version field of type `java.lang.Long` to the entity bean for optimistic locking. Each update of the entity will increase the value of this field. The **field-name** element allows for the specification of the name of the CMP field while the **column-name** element allows for the specification of the corresponding table column.
- **timestamp-column**: This element specifies that optimistic locking is based on a timestamp column strategy. Specifying this element will add an additional version field of type `java.util.Date` to the entity bean for optimistic locking. Each update of the entity will set the value of this field to the current time. The **field-name** element allows for the specification of the name of the CMP field while the **column-name** element allows for the specification of the corresponding table column.
- **key-generator-factory**: This element specifies that optimistic locking is based on key generation. The value of the element is the JNDI name of a `org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory` implementation. Specifying this element will add an additional version field to the entity bean for optimistic locking. The type of the field must be specified via the **field-type** element. Each update of the entity will update the key field by obtaining a new value from the key generator. The **field-name** element allows for the specification of the name of the CMP field while the **column-name** element allows for the specification of the corresponding table column.

A sample `jbosscmp-jdbc.xml` descriptor illustrating all of the optimistic locking strategies is given below.

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
"-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
<jbosscmp-jdbc>
  <defaults>
    <datasource>java:/DefaultDS</datasource>
    <datasource-mapping>Hypersonic SQL</datasource-mapping>
  </defaults>
  <enterprise-beans>
    <entity>
      <ejb-name>EntityGroupLocking</ejb-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
      <table-name>entitygrouplocking</table-name>
      <cmp-field>
        <field-name>dateField</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>integerField</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>stringField</field-name>
      </cmp-field>
      <load-groups>
        <load-group>
          <load-group-name>string</load-group-name>
          <field-name>stringField</field-name>
```

```

        </load-group>
        <load-group>
            <load-group-name>all</load-group-name>
            <field-name>stringField</field-name>
            <field-name>dateField</field-name>
        </load-group>
    </load-groups>
    <optimistic-locking>
        <group-name>string</group-name>
    </optimistic-locking>
</entity>
<entity>
    <ejb-name>EntityModifiedLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entitymodifiedlocking</table-name>
    <cmp-field>
        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <modified-strategy/>
    </optimistic-locking>
</entity>
<entity>
    <ejb-name>EntityReadLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entityreadlocking</table-name>
    <cmp-field>
        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <read-strategy/>
    </optimistic-locking>
</entity>
<entity>
    <ejb-name>EntityVersionLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entityversionlocking</table-name>
    <cmp-field>
        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>

```

```

        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <version-column/>
        <field-name>versionField</field-name>
        <column-name>ol_version</column-name>
        <jdbc-type>INTEGER</jdbc-type>
        <sql-type>INTEGER(5)</sql-type>
    </optimistic-locking>
</entity>
<entity>
    <ejb-name>EntityTimestampLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entitytimestamplocking</table-name>
    <cmp-field>
        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <timestamp-column/>
        <field-name>versionField</field-name>
        <column-name>ol_timestamp</column-name>
        <jdbc-type>TIMESTAMP</jdbc-type>
        <sql-type>DATETIME</sql-type>
    </optimistic-locking>
</entity>
<entity>
    <ejb-name>EntityKeyGeneratorLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entitykeygenlocking</table-name>
    <cmp-field>
        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <key-generator-factory>UUIDKeyGeneratorFactory</key-
generator-factory>
        <field-type>java.lang.String</field-type>
        <field-name>uuidField</field-name>
        <column-name>ol_uuid</column-name>
        <jdbc-type>VARCHAR</jdbc-type>

```

```

        <sql-type>VARCHAR(32)</sql-type>
      </optimistic-locking>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>

```

21.11. ENTITY COMMANDS AND PRIMARY KEY GENERATION

Support for primary key generation outside of the entity bean is available through custom implementations of the entity creation command objects used to insert entities into a persistent store. The list of available commands is specified in `entity-commands` element of the `jbosscmp-jdbc.xml` descriptor. The default `entity-command` may be specified in the `jbosscmp-jdbc.xml` in `defaults` element. Each entity element can override the `entity-command` in `defaults` by specifying its own `entity-command`. The content model of the `entity-commands` and child elements is given below.



Figure 21.15. The `jbosscmp-jdbc.xml` `entity-commands` element model

Each `entity-command` element specifies an entity generation implementation. The `name` attribute specifies a name that allows the command defined in an `entity-commands` section to be referenced in the `defaults` and entity elements. The `class` attribute specifies the implementation of the `org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand` that supports the key generation. Database vendor specific commands typically subclass the `org.jboss.ejb.plugins.cmp.jdbc.JDBCIdentityColumnCreateCommand` if the database generates the primary key as a side effect of doing an insert, or the `org.jboss.ejb.plugins.cmp.jdbc.JDBCInsertPKCreateCommand` if the command must insert the generated key.

The optional `attribute` element(s) allows for the specification of arbitrary name/value property pairs that will be available to the entity command implementation class. The `attribute` element has a required `name` attribute that specifies the name property, and the `attribute` element content is the value of the property. The attribute values are accessible through the `org.jboss.ejb.plugins.cmp.jdbc.metadata.JDBCEntityCommandMetaData.getAttribute(String)` method.

21.11.1. Existing Entity Commands

The following are the current `entity-command` definitions found in the `standardjbosscmp-jdbc.xml` descriptor:

- **default:** (`org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand`) The `JDBCCreateEntityCommand` is the default entity creation as it is the `entity-command` referenced in the `standardjbosscmp-jdbc.xml` `defaults` element. This entity-command executes an `INSERT INTO` query using the assigned primary key value.
- **no-select-before-insert:** (`org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand`) This is a variation on `default` that skips select before insert by specifying an attribute `name="SQLExceptionProcessor"` that points to the

`jboss.jdbc:service=SQLExceptionProcessor` service. The `SQLExceptionProcessor` service provides a `boolean isDuplicateKey(SQLException e)` operation that allows a for determination of any unique constraint violation.

- **pk-sql** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPkSqlCreateCommand`) The `JDBCPkSqlCreateCommand` executes an `INSERT INTO` query statement provided by the `pk-sql` attribute to obtain the next primary key value. Its primary target usage are databases with sequence support.
- **mysql-get-generated-keys:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCMySQLCreateCommand`) The `JDBCMySQLCreateCommand` executes an `INSERT INTO` query using the `getGeneratedKeys` method from MySQL native `java.sql.Statement` interface implementation to fetch the generated key.
- **oracle-sequence:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCOracleCreateCommand`) The `JDBCOracleCreateCommand` is a create command for use with Oracle that uses a sequence in conjunction with a `RETURNING` clause to generate keys in a single statement. It has a required `sequence` element that specifies the name of the sequence column.
- **hsqldb-fetch-key:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCCHsqldbCreateCommand`) The `JDBCCHsqldbCreateCommand` executes an `INSERT INTO` query after executing a `CALL IDENTITY()` statement to fetch the generated key.
- **sybase-fetch-key:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JBCSybaseCreateCommand`) The `JBCSybaseCreateCommand` executes an `INSERT INTO` query after executing a `SELECT @@IDENTITY` statement to fetch the generated key.
- **mssql-fetch-key:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCSQLServerCreateCommand`) The `JDBCSQLServerCreateCommand` for Microsoft SQL Server that uses the value from an `IDENTITY` columns. By default uses `SELECT SCOPE_IDENTITY()` to reduce the impact of triggers; can be overridden with `pk-sql` attribute e.g. for V7.
- **informix-serial:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCInformixCreateCommand`) The `JDBCInformixCreateCommand` executes an `INSERT INTO` query after using the `getSerial` method from Informix native `java.sql.Statement` interface implementation to fetch the generated key.
- **postgresql-fetch-seq:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPostgreSQLCreateCommand`) The `JDBCPostgreSQLCreateCommand` for PostgreSQL that fetches the current value of the sequence. The optional `sequence` attribute can be used to change the name of the sequence, with the default being `table_pkColumn_seq`.
- **key-generator:** (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCKeyGeneratorCreateCommand`) The `JDBCKeyGeneratorCreateCommand` executes an `INSERT INTO` query after obtaining a

value for the primary key from the key generator referenced by the **key-generator-factory**. The **key-generator-factory** attribute must provide the name of a JNDI binding of the **org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory** implementation.

- get-generated-keys:**
 (org.jboss.ejb.plugins.cmp.jdbc.jdbc3.JDBCGetGeneratedKeysCreateCommand) The **JDBCGetGeneratedKeysCreateCommand** executes an **INSERT INTO** query using a statement built using the JDBC3 **prepareStatement(String, Statement.RETURN_GENERATED_KEYS)** that has the capability to retrieve the auto-generated key. The generated key is obtained by calling the **PreparedStatement.getGeneratedKeys** method. Since this requires JDBC3 support it is only available in JDK1.4.1+ with a supporting JDBC driver.

An example configuration using the **hsqldb-fetch-keyentity-command** with the generated key mapped to a known primary key **cmp-field** is shown below.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <pk-constraint>false</pk-constraint>
      <table-name>location</table-name>

      <cmp-field>
        <field-name>locationID</field-name>
        <column-name>id</column-name>
        <auto-increment/>
      </cmp-field>
      <!-- ... -->
      <entity-command name="hsqldb-fetch-key"/>

    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

An alternate example using an unknown primary key without an explicit **cmp-field** is shown below.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <pk-constraint>false</pk-constraint>
      <table-name>location</table-name>
      <unknown-pk>
        <unknown-pk-class>java.lang.Integer</unknown-pk-class>
        <field-name>locationID</field-name>
        <column-name>id</column-name>
        <jdbc-type>INTEGER</jdbc-type>
        <sql-type>INTEGER</sql-type>
        <auto-increment/>
      </unknown-pk>
      <!-- ... -->
      <entity-command name="hsqldb-fetch-key"/>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```



```
        </entity>
    </enterprise-beans>
</jbossCMP-jdbc>
```

21.12. DEFAULTS

JBoss global defaults are defined in the **standardjbossCMP-jdbc.xml** file of the **server/<server-name>/conf/** directory. Each application can override the global defaults in the **jbossCMP-jdbc.xml** file. The default options are contained in a **defaults** element of the configuration file, and the content model is shown below.

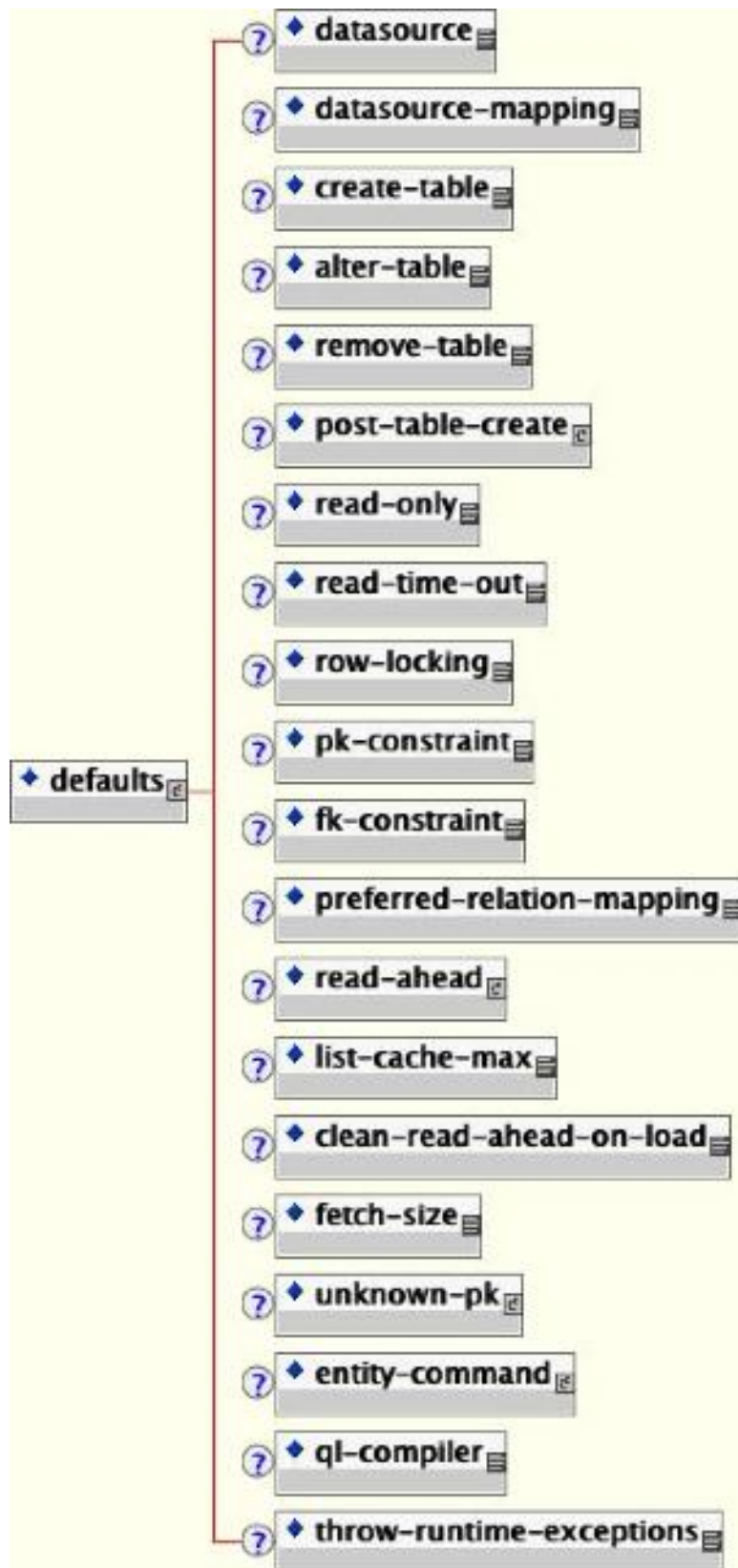


Figure 21.16. The jbossCMP-jdbc.xml defaults content model

An example of the defaults section follows:

```

<jbossCMP-jdbc>
  <defaults>
    <datasource>java:/DefaultDS</datasource>
    <datasource-mapping>Hypersonic SQL</datasource-mapping>
    <create-table>true</create-table>
  
```

```

        <remove-table>false</remove-table>
        <read-only>false</read-only>
        <read-time-out>300000</read-time-out>
        <pk-constraint>true</pk-constraint>
        <fk-constraint>false</fk-constraint>
        <row-locking>false</row-locking>
        <preferred-relation-mapping>foreign-key</preferred-relation-
mapping>
        <read-ahead>
            <strategy>on-load</strategy>
            <page-size>1000</page-size>
            <eager-load-group>*</eager-load-group>
        </read-ahead>
        <list-cache-max>1000</list-cache-max>
    </defaults>
</jbossCMP-jdbc>

```

21.12.1. A sample jbossCMP-jdbc.xml defaults declaration

Each option can apply to entities, relationships, or both, and can be overridden in the specific entity or relationship. A detailed description of each option follows:

- **datasource**: This optional element is the **jndi-name** used to look up the datasource. All database connections used by an entity or **relation-table** are obtained from the datasource. Having different datasources for entities is not recommended, as it vastly constrains the domain over which finders and ejbSelects can query.
- **datasource-mapping**: This optional element specifies the name of the **type-mapping**, which determines how Java types are mapped to SQL types, and how EJB-QL functions are mapped to database specific functions. Type mappings are discussed in [Section 21.13.3, "Mapping"](#).
- **create-table**: This optional element when true, specifies that JBoss should attempt to create a table for the entity. When the application is deployed, JBoss checks if a table already exists before creating the table. If a table is found, it is logged, and the table is not created. This option is very useful during the early stages of development when the table structure changes often. The default is false.
- **alter-table**: If **create-table** is used to automatically create the schema, **alter-table** can be used to keep the schema current with changes to the entity bean. Alter table will perform the following specific tasks:
 - new fields will be created
 - fields which are no longer used will be removed
 - string fields which are shorter than the declared length will have their length increased to the declared length. (not supported by all databases)
- **remove-table**: This optional element when true, JBoss will attempt to drop the table for each entity and each relation table mapped relationship. When the application is undeployed, JBoss will attempt to drop the table. This option is very useful during the early stages of development when the table structure changes often. The default is false.
- **read-only**: This optional element when true specifies that the bean provider will not be allowed to change the value of any fields. A field that is read-only will not be stored in, or inserted into, the database. If a primary key field is read-only, the create method will throw a

CreateException. If a set accessor is called on a **read-only** field, it throws an **EJBException**. Read only fields are useful for fields that are filled in by database triggers, such as last update. The **read-only** option can be overridden on a per field basis. The default is false.

- **read-time-out:** This optional element is the amount of time in milliseconds that a read on a read only field is valid. A value of 0 means that the value is always reloaded at the start of a transaction, and a value of -1 means that the value never times out. This option can also be overridden on a per CMP field basis. If **read-only** is false, this value is ignored. The default is -1.
- **row-locking:** This optional element if true specifies that JBoss will lock all rows loaded in a transaction. Most databases implement this by using the **SELECT FOR UPDATE** syntax when loading the entity, but the actual syntax is determined by the **row-locking-template** in the **datasource-mapping** used by this entity. The default is false.
- **pk-constraint:** This optional element if true specifies that JBoss will add a primary key constraint when creating tables. The default is true.
- **preferred-relation-mapping:** This optional element specifies the preferred mapping style for relationships. The **preferred-relation-mapping** element must be either **foreign-key** or **relation-table**.
- **read-ahead:** This optional element controls caching of query results and CMR fields for the entity. This option is discussed in [Section 21.7.3, “Read-ahead”](#).
- **list-cache-max:** This optional element specifies the number of **read-lists** that can be tracked by this entity. This option is discussed in [Section 21.7.3.2, “on-load”](#). The default is 1000.
- **clean-read-ahead-on-load:** When an entity is loaded from the read ahead cache, JBoss can remove the data used from the read ahead cache. The default is **false**.
- **fetch-size:** This optional element specifies the number of entities to read in one round-trip to the underlying datastore. The default is 0.
- **unknown-pk:** This optional element allows one to define the default mapping of an unknown primary key type of **java.lang.Object** maps to the persistent store.
- **entity-command:** This optional element allows one to define the default command for entity creation. This is described in detail in [Section 21.11, “Entity Commands and Primary Key Generation”](#).
- **ql-compiler:** This optional elements allows a replacement query compiler to be specified. Alternate query compilers were discussed in [Section 21.6.7, “EJBQL 2.1 and SQL92 queries”](#).
- **throw-runtime-exceptions:** This attribute, if set to true, indicates that an error in connecting to the database should be seen in the application as runtime **EJBException** rather than as a checked exception.

21.13. DATASOURCE CUSTOMIZATION

JBoss includes predefined type-mappings for many databases including: Cloudscape, DB2, DB2/400, Hypersonic SQL, InformixDB, InterBase, MS SQLSERVER, MS SQLSERVER2000, MySQL, Oracle7, Oracle8, Oracle9i, PointBase, PostgreSQL, PostgreSQL 7.2, SapDB, SOLID, and Sybase. If you do not

like the supplied mapping, or a mapping is not supplied for your database, you will have to define a new mapping. If you find an error in one of the supplied mappings, or if you create a new mapping for a new database, please consider posting a patch at the JBoss project page on SourceForge.

21.13.1. Type Mapping

Customization of a database is done through the **type-mapping** section of the **jbosscmp-jdbc.xml** descriptor. The content model for the type-mapping element is given in [Figure 21.17](#), “The jbosscmp-jdbc type-mapping element content model.”.

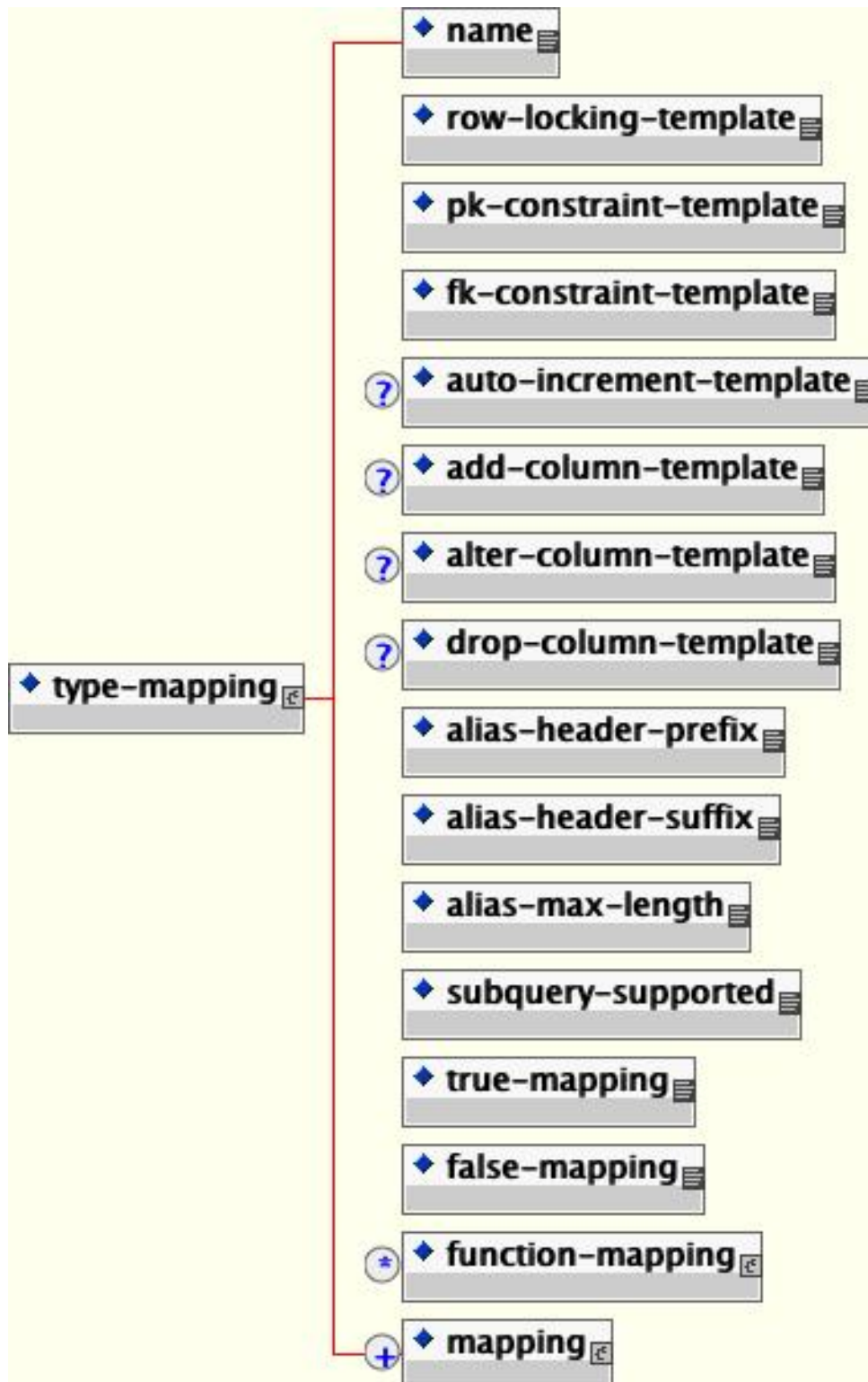


Figure 21.17. The jbosscmp-jdbc type-mapping element content model.

The elements are:

- **name**: This required element provides the name identifying the database customization. It is used to refer to the mapping by the **datasource-mapping** elements found in defaults and entity.
- **row-locking-template**: This required element gives the **PreparedStatement** template used to create a row lock on the selected rows. The template must support three arguments:
 1. the select clause
 2. the from clause. The order of the tables is currently not guaranteed
 3. the where clause

If row locking is not supported in select statement this element should be empty. The most common form of row locking is select for update as in: **SELECT ?1 FROM ?2 WHERE ?3 FOR UPDATE.**

- **pk-constraint-template**: This required element gives the **PreparedStatement** template used to create a primary key constraint in the create table statement. The template must support two arguments
 1. Primary key constraint name; which is always **pk_{table-name}**
 2. Comma separated list of primary key column names

If a primary key constraint clause is not supported in a create table statement this element should be empty. The most common form of a primary key constraint is: **CONSTRAINT ?1 PRIMARY KEY (?2)**

- **fk-constraint-template**: This is the template used to create a foreign key constraint in separate statement. The template must support five arguments:
 1. Table name
 2. Foreign key constraint name; which is always **fk_{table-name}_{cmr-field-name}**
 3. Comma separated list of foreign key column names
 4. References table name
 5. Comma separated list of the referenced primary key column names

If the datasource does not support foreign key constraints this element should be empty. The most common form of a foreign key constraint is: **ALTER TABLE ?1 ADD CONSTRAINT ?2 FOREIGN KEY (?3) REFERENCES ?4 (?5).**

- **auto-increment-template**: This declares the SQL template for specifying auto increment columns.
- **add-column-template**: When **alter-table** is true, this SQL template specifies the syntax for adding a column to an existing table. The default value is **ALTER TABLE ?1 ADD ?2 ?3**. The parameters are:
 1. the table name

2. the column name
 3. the column type
- **alter-column-template**: When **alter-table** is true, this SQL template specifies the syntax for dropping a column to from an existing table. The default value is **ALTER TABLE ?1 ALTER ?2 TYPE ?3**. The parameters are:
 1. the table name
 2. the column name
 3. the column type
 - **drop-column-template**: When **alter-table** is true, this SQL template specifies the syntax for dropping a column to from an existing table. The default value is **ALTER TABLE ?1 DROP ?2**. The parameters are:
 1. the table name
 2. the column name
 - **alias-header-prefix**: This required element gives the prefix used in creating the alias header. An alias header is prepended to a generated table alias by the EJB-QL compiler to prevent name collisions. The alias header is constructed as follows: alias-header-prefix + int_counter + alias-header-suffix. An example alias header would be **t0_** for an alias-header-prefix of **"t"** and an alias-header-suffix of **"_"**.
 - **alias-header-suffix**: This required element gives the suffix portion of the generated alias header.
 - **alias-max-length**: This required element gives the maximum allowed length for the generated alias header.
 - **subquery-supported**: This required element specifies if this **type-mapping** subqueries as either true or false. Some EJB-QL operators are mapped to exists subqueries. If **subquery-supported** is false, the EJB-QL compiler will use a left join and is null.
 - **true-mapping**: This required element defines *true* identity in EJB-QL queries. Examples include **TRUE**, **1**, and **(1=1)**.
 - **false-mapping**: This required element defines *false* identity in EJB-QL queries. Examples include **FALSE**, **0**, and **(1=0)**.
 - **function-mapping**: This optional element specifies one or more the mappings from an EJB-QL function to an SQL implementation. See [Section 21.13.2, "Function Mapping"](#) for the details.
 - **mapping**: This required element specifies the mappings from a Java type to the corresponding JDBC and SQL type. See [Section 21.13.3, "Mapping"](#) for the details.

21.13.2. Function Mapping

The function-mapping element model is show below.

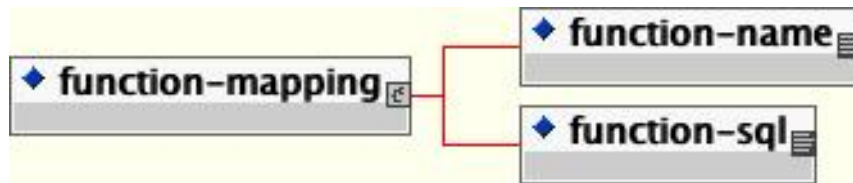


Figure 21.18. The jbosscmp-jdbc function-mapping element content model

The allowed child elements are:

- **function-name**: This required element gives the EJB-QL function name, e.g., **concat**, **substring**.
- **function-sql**: This required element gives the SQL for the function as appropriate for the underlying database. Examples for a **concat** function include: **(?1 || ?2)**, **concat(?1, ?2)**, **(?1 + ?2)**.

21.13.3. Mapping

A **type-mapping** is simply a set of mappings between Java class types and database types. A set of type mappings is defined by a set of **mapping** elements, the content model for which is shown in Figure 21.19, “The jbosscmp-jdbc mapping element content model.”

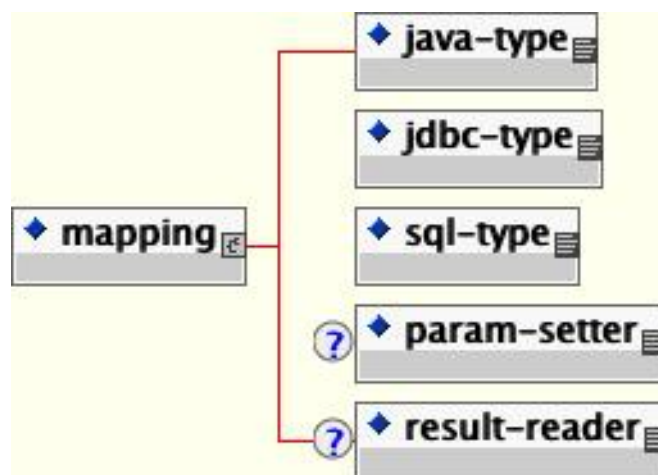


Figure 21.19. The jbosscmp-jdbc mapping element content model.

If JBoss cannot find a mapping for a type, it will serialize the object and use the **java.lang.Object** mapping. The following describes the three child elements of the mapping element:

- **java-type**: This required element gives the fully qualified name of the Java class to be mapped. If the class is a primitive wrapper class such as **java.lang.Short**, the mapping also applies to the primitive type.
- **jdbc-type**: This required element gives the JDBC type that is used when setting parameters in a JDBC **PreparedStatement** or loading data from a JDBC **ResultSet**. The valid types are defined in **java.sql.Types**.
- **sql-type**: This required element gives the SQL type that is used in create table statements. Valid types are only limited by your database vendor.
- **param-setter**: This optional element specifies the fully qualified name of the **JDBCParameterSetter** implementation for this mapping.

- **result-reader**: This option element specifies the fully qualified name of the **JDBCResultSetReader** implementation for this mapping.

An example mapping element for a short in Oracle9i is shown below.

```
<jbosscmp-jdbc>
  <type-mappings>
    <type-mapping>
      <name>Oracle9i</name>
      <!--...-->
      <mapping>
        <java-type>java.lang.Short</java-type>
        <jdbc-type>NUMERIC</jdbc-type>
        <sql-type>NUMBER(5)</sql-type>
      </mapping>
    </type-mapping>
  </type-mappings>
</jbosscmp-jdbc>
```

21.13.4. User Type Mappings

User type mappings allow one to map from JDBC column types to custom CMP fields types by specifying an instance of **org.jboss.ejb.plugins.cmp.jdbc.Mapper** interface, the definition of which is shown below.

```
public interface Mapper
{
    /**
     * This method is called when CMP field is stored.
     * @param fieldValue - CMP field value
     * @return column value.
     */
    Object toColumnValue(Object fieldValue);

    /**
     * This method is called when CMP field is loaded.
     * @param columnValue - loaded column value.
     * @return CMP field value.
     */
    Object toFieldValue(Object columnValue);
}
```

A prototypical use case is the mapping of an integer type to its type-safe Java enumeration instance. The content model of the **user-type-mappings** element consists of one or more **user-type-mapping** elements, the content model of which is shown in [Figure 21.20, “The user-type-mapping content model >”](#).

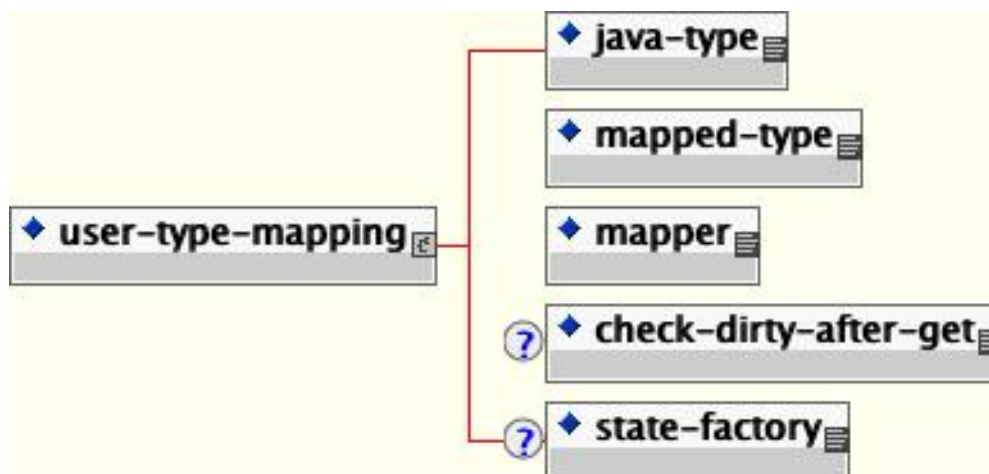


Figure 21.20. The user-type-mapping content model >

- **java-type**: the fully qualified name of the CMP field type in the mapping.
- **mapped-type**: the fully qualified name of the database type in the mapping.
- **mapper**: the fully qualified name of the **Mapper** interface implementation that handles the conversion between the **java-type** and **mapped-type**.

APPENDIX A. BOOK EXAMPLE INSTALLATION

The book comes with the source code for the examples discussed in the book. You can download the examples zip file from <http://www.redhat.com/docs>. Unzipping the example code archive creates a **server-config-guide** directory. This is the **examples** directory referred to by the book.

The only customization needed before the examples may be used is to set the location of the JBoss server distribution. This may be done by editing the **examples/build.xml** file and changing the **jboss.dist** property value. This is shown in bold below:

```
<project name="JBoss book examples" default="build-all" basedir=".">
  <!-- Allow overrides from a local properties file -->
  <property file="ant.properties" />

  <!-- Override with your JBoss server bundle dist location -->
  <property name="jboss.dist"
value="${basedir}/../../../../jboss-eap-4.3/jboss-as" />
  <property name="jboss.deploy.conf" value="default" />
  <property name="jboss.deploy.dir"
value="${jboss.dist}/server/${jboss.deploy.conf}/deploy" />
  ...
  ...
```

or by creating an **.ant.properties** file in the examples directory that contains a definition for the **jboss.dist** property. For example:

```
jboss.dist=/usr/local/jboss-eap-4.3/jboss-as
```

Part of the verification process validates that the version you are running the examples against matches what the book examples were tested against. If you have a problem running the examples first look for the output of the validate target.

APPENDIX B. USE ALTERNATIVE DATABASES WITH JBOSS AS

B.1. HOW TO USE ALTERNATIVE DATABASES



WARNING

The Hypersonic database provides default "out of the box" database functionality for evaluation and development use only. It should *not* be used as a production-use database. Technical support is not available for this component, and while we accept bugs filed against this component, we do not make any commitment to fix them within a specific timeframe.

JBoss utilizes the Hypersonic database as its default database. While this is good for development and prototyping, you or your company will probably require another database to be used for production. This chapter covers configuring JBoss AS to use alternative databases. We cover the procedures for all officially supported databases on JBoss Enterprise Application Platform. They include: MySQL 5.0, PostgreSQL 8.1, Oracle 9i and 10g R2, DB2 7.2 and 8, Sybase ASE 12.5, as well as MS SQL 2005.

Please note that in this chapter, we explain how to use alternative databases to support all services in JBoss AS. This includes all the system level services such as EJB and JMS. For individual applications (e.g., WAR or EAR) deployed in JBoss AS, you can still use any backend database by setting up the appropriate data source connection as described in [Section 7.3, "Configuring JDBC DataSources"](#).

We assume that you have already installed the external database server, and have it running. You should create an empty database named **jboss**, accessible via the username / password pair **jbossuser** / **jbosspass**. The **jboss** database is used to store JBoss AS internal data -- JBoss AS will automatically create tables and data in it.

B.2. INSTALL JDBC DRIVERS

For the JBoss Application Server and our applications to use the external database, we also need to install the database's JDBC driver. The JDBC driver is a JAR file, which you'll need to copy into your JBoss AS's **jboss-as/server/production/lib** directory. Replace **production** with the server configuration you are using if needed. This file is loaded when JBoss starts up. So if you have the JBoss AS running, you'll need to shut down and restart. The availability of JDBC drivers for different databases are as follows.

- MySQL JDBC drivers can be obtained from <http://dev.mysql.com/downloads/connector/j/5.0.html>. The download contains documentation, etc. for the JDBC connector, but you really only need the **mysql-connector-java-5.0.4-bin.jar** file to get MySQL to work with and be used by JBoss AS.
- PostgreSQL JDBC drivers can be obtained from: <http://jdbc.postgresql.org/>. For 8.2.3 version, we need the JDBC3 driver **8.2-504 JDBC 3**. The download is just the JDBC driver **postgresql-8.2-504.jdbc3.jar** file.

- Oracle thin JDBC drivers can be obtained from:
http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html for your Oracle DB versions.
- IBM DB2 JDBC drivers can be downloaded from the IBM web site <http://www-306.ibm.com/software/data/db2/java/>.
- Sybase JDBC drivers can be downloaded from the Sybase jConnect product page <http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect>
- MS SQL Server JDBC drivers can be downloaded from the MSDN web site <http://msdn.microsoft.com/data/jdbc/>.

B.2.1. Special notes on Sybase

Some of the services in JBoss uses null values for the default tables that are created. Sybase Adaptive Server should be configured to allow nulls by default.

```
sp_dboption db_name, "allow nulls by default", true
```

Refer the sybase manuals for more options.

Enable JAVA services

To use any java service like JMS, CMP, timers etc. configured with Sybase, java should be enabled on Sybase Adaptive Server. To do this use:

```
sp_configure "enable java",1
```

Refer the sybase manuals for more information.

If java is not enabled you might see this exception being thrown when you try to use any of the above services.

```
com.sybase.jdbc2.jdbc.SybSQLException: Cannot run this command because
Java services are not enabled. A user with System Administrator (SA) role
must reconfigure the system to enable Java
```

CMP Configuration

To use Container Managed Persistence for user defined Java objects with Sybase Adaptive Server Enterprise the java classes should be installed in the database. The system table 'sysxtypes' contains one row for each extended, Java-SQL datatype. This table is only used for Adaptive Servers enabled for Java. Install java classes using the installjava program.

```
installjava -f <jar-file-name> -S<sybase-server> -U<super-user> -P<super-
pass> -D<db-name>
```

Refer the installjava manual in Sybase for more options.

**NOTE**

1. You have to be a super-user with required privileges to install java classes.
2. The jar file you are trying to install should be created without compression.
3. Java classes that you install and use in the server must be compiled with JDK 1.2.2. If you compile a class with a later JDK, you will be able to install it in the server using the installjava utility, but you will get a `java.lang.ClassFormatError` exception when you attempt to use the class. This is because Sybase Adaptive Server uses an older JVM internally, and hence requires the java classes to be compiled with the same.

B.3. CREATING A DATASOURCE FOR THE EXTERNAL DATABASE

JBoss AS connects to relational databases via datasources. These datasource definitions can be found in the **`jboss-as/server/production/deploy`** directory. The datasource definitions are deployable just like WAR and EAR files. The datasource files can be recognized by looking for the XML files that end in **`*-ds.xml`**.

The datasource definition files for all supported external databases can be found in the **`jboss-as/docs/examples/jca`** directory.

- MySQL: **`mysql-ds.xml`**
- PostgreSQL: **`postgres-ds.xml`**
- Oracle: **`oracle-ds.xml`**
- DB2: **`db2-ds.xml`**
- Sybase: **`sybase-ds.xml`**
- MS SQL Server: **`mssql-ds.xml`**

The following code snippet shows the **`mysql-ds.xml`** file as an example. All the other **`*-ds.xml`** files are very similar. You will need to change the **`connection-url`**, as well as the **`user-name`** / **`password`**, to fit your own database server installation.

```
<datasources>
  <local-tx-datasource>
    <jndi-name>MySqlDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/jboss</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>jbossuser</user-name>
    <password>jbosspass</password>
    <exception-sorter-class-name>
      org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
    </exception-sorter-class-name>
    <!-- should only be used on drivers after 3.22.1 with "ping" support
    <valid-connection-checker-class-name>
      org.jboss.resource.adapter.jdbc.vendor.MySQLValidConnectionChecker
    </valid-connection-checker-class-name>
    -->
```

```

    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->
    <!-- sql to call on an existing pooled connection when it is obtained
from pool -
    MySQLValidConnectionChecker is preferred for newer drivers
    <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
    -->

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
        <type-mapping>mySQL</type-mapping>
    </metadata>
</local-tx-datasource>

</datasources>

```

Once you customized the `*-ds.xml` file to connect to your external database, you need to copy it to the `jboss-as/server/production/deploy` directory. The database connection is now available through the JNDI name specified in the `*-ds.xml` file.

B.4. CHANGE DATABASE FOR THE JMS SERVICES

The JMS service in the JBoss AS uses relational databases to persist its messages. For improved performance, you should change the JMS service to take advantage of the external database. To do that, you need to replace the file `jboss-as/server/production/deploy/jboss-messaging.sar/clustered-hsqldb-persistence-service.xml` with a file in `jboss-as/docs/examples/jms/` depending on your external database and restart your server.

- MySQL: `mysql-persistence-service.xml`
- PostgreSQL: `postgresql-persistence-service.xml`
- Oracle: `oracle-persistence-service.xml`
- Sybase: `sybase-persistence-service.xml`
- MS SQL Server: `mssql-persistence-service.xml`

For the `default` and `all` configurations, replace the files `jboss-as/server/default/deploy/jboss-messaging.sar/hsqldb-persistence-service.xml` and `jboss-as/server/all/deploy/jboss-messaging.sar/clustered-hsqldb-persistence-service.xml` respectively.

B.5. SUPPORT FOREIGN KEYS IN CMP SERVICES

Next, we need to go change the `jboss-as/server/production/conf/standardjbosscmp-jdbc.xml` file so that the `fk-constraint` property is `true`. That is needed for all external databases we support on the JBoss Enterprise Application Platform. This file configures the database connection settings for the EJB2 CMP beans deployed in the JBoss AS.

```
<fk-constraint>true</fk-constraint>
```

B.6. SPECIFY DATABASE DIALECT FOR JAVA PERSISTENCE API

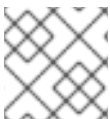
The Java Persistence API (JPA) entity manager can save EJB3 entity beans to any backend database. Hibernate provides the JPA implementation in JBoss AS. In order for Hibernate to work correctly with alternative databases, we recommend you configure the database dialect in the **jboss-as/server/production/deploy/ejb3.deployer/META-INF/persistence.properties** file. You need to un-comment the **hibernate.dialect** property and change its value to the following based on the database you setup.

- Oracle 9i: org.hibernate.dialect.Oracle9iDialect
- Oracle 10g: org.hibernate.dialect.Oracle10gDialect
- Microsoft SQL Server 2005: org.hibernate.dialect.SQLServerDialect
- PostgreSQL 8.1: org.hibernate.dialect.PostgreSQLDialect
- MySQL 5.0: org.hibernate.dialect.MySQL5Dialect
- DB2 8.0: org.hibernate.dialect.DB2Dialect
- Sybase ASE 12.5: org.hibernate.dialect.SybaseDialect



NOTE

Large Objects (LOBs) are supported only with DB2 Version 8 servers and above with the universal JDBC driver. Hence JMS services which stores messages as BLOBS and Timer services which uses BLOB fields for storing objects do not work with the JDBC Type 4 driver and DB2 7.2.



NOTE

All JBoss services work with the JDBC Type 2 driver and DB2 Version 7.2 servers.

B.7. CHANGE OTHER JBOSS AS SERVICES TO USE THE EXTERNAL DATABASE

Besides JMS, CMP, and JPA, we still need to hook up the rest of JBoss services with the external database. There are two ways to do it. One is easy but inflexible. The other is flexible but requires more steps. Now, let's discuss those two approaches respectively.

B.7.1. The Easy Way

The easy way is just to change the JNDI name for the external database to **DefaultDS**. Most JBoss services are hard-wired to use the **DefaultDS** by default. So, by changing the datasource name, we do not need to change the configuration for each service individually.

To change the JNDI name, just open the ***-ds.xml** file for your external database, and change the value of the **jndi-name** property to **DefaultDS**. For instance, in **mysql-ds.xml**, you'd change **MySqlDS** to

DefaultDS and so on. You will need to remove the **jboss-as/server/production/deploy/hsqldb-ds.xml** file after you are done to avoid duplicated **DefaultDS** definition.

In the **jms/*-jdbc2-service.xml** file, you should also change the datasource name in the **depends** tag for the **PersistenceManagers** MBean to **DefaultDS**. For instance, for **mysql-jdbc2-service.xml** file, we change the **MySqlDS** to **DefaultDS**.

```
... ..
<mbean code="org.jboss.mq.pm.jdbc2.PersistenceManager"
      name="jboss.mq:service=PersistenceManager">
  <depends optional-attribute-name="ConnectionManager">
    jboss.jca:service=DataSourceBinding,name=DefaultDS
  </depends>
... ..
```

B.7.2. The More Flexible Way

Changing the external datasource to **DefaultDS** is convenient. But if you have applications that assume the **DefaultDS** always points to the factory-default HSQL DB, that approach could break your application. Also, changing **DefaultDS** destination forces all JBoss services to use the external database. What if you want to use the external database only on some services?

A safer and more flexible way to hook up JBoss AS services with the external datasource is to manually change the **DefaultDS** in all standard JBoss services to the datasource JNDI name defined in your ***-ds.xml** file (e.g., the **MySqlDS** in **mysql-ds.xml** etc.). Below is a complete list of files that contain **DefaultDS**. You can update them all to use the external database on all JBoss services or update some of them to use different combination of datasources for different services.

- **jboss-as/server/production/conf/login-config.xml**: This file is used in Java EE container managed security services.
- **jboss-as/server/production/conf/standardjbosscmp-jdbc.xml**: This file configures the CMP beans in the EJB container.
- **jboss-as/server/production/deploy/ejb-deployer.xml**: This file configures the JBoss EJB deployer.
- **jboss-as/server/production/deploy/schedule-manager-service.xml**: This file configures the EJB timer services.
- **jboss-as/server/production/deploy/snmp-adaptor.sar/attributes.xml**: This file is used by the SNMP service.
- **jboss-as/server/production/deploy/juddi-service.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- **jboss-as/server/production/deploy/juddi-service.sar/juddi.war/META-INF/jboss-web.xml**: This file configures the UUDI service.
- **jboss-as/server/production/deploy/juddi-service.sar/juddi.war/META-INF/juddi.properties**: This file configures the UUDI service.

- **jboss-as/server/production/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- **jboss-as/server/production/deploy/jboss-messaging.sar/clustered-hsqldb-persistence-service.xml**: This file configures the JMS persistence service as we discussed earlier.

B.8. A SPECIAL NOTE ABOUT ORACLE DATABASES

In our setup discussed in this chapter, we rely on the JBoss AS to automatically create needed tables in the external database upon server startup. That works most of the time. But for databases like Oracle, there might be some minor issues if you try to use the same database server to back more than one JBoss AS instance.

The Oracle database creates tables of the form **schemaname.tablename**. The **TIMERS** and **HILOSEQUENCES** tables needed by JBoss AS would not get created on a schema if the table already exists on a different schema. To work around this issue, you need to edit the **jboss-as/server/production/deploy/ejb-deployer.xml** file to change the table name from **TIMERS** to something like **schemaname2.tablename**.

```
... ..
    <mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
    <!-- DataSourceBinding ObjectName -->
    <depends optional-attribute-name="DataSource">
        jboss.jca:service=DataSourceBinding,name=DefaultDS
    </depends>
    <!-- The plugin that handles database persistence -->
    <attribute name="DatabasePersistencePlugin">
        org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin
    </attribute>
    <!-- The timers table name -->
    <attribute name="TimersTable">TIMERS</attribute>
</mbean>
```

Similarly, you need to change the **jboss-as/server/production/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml** file to change the table name from **HILOSEQUENCES** to something like **schemaname2.tablename** as well.

```
... ..
    <!-- HiLoKeyGeneratorFactory -->
    <mbean
code="org.jboss.ejb.plugins.keygenerator.hilo.HiLoKeyGeneratorFactory"
    name="jboss:service=KeyGeneratorFactory,type=HiLo">

        <depends>jboss:service=TransactionManager</depends>

        <!-- Attributes common to HiLo factory instances -->

        <!-- DataSource JNDI name -->
```

```
    <depends optional-attribute-  
name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</depe  
nds>  
  
    <!-- table name -->  
    <attribute name="TableName">HILOSEQUENCES</attribute>  
  
    . . . . .
```

APPENDIX C. VENDOR-SPECIFIC DATASOURCE DEFINITIONS

This appendix includes datasource definitions for databases supported by JBoss Enterprise Application Platform.

C.1. DEPLOYER LOCATION AND NAMING

All database deployers should be saved to the **`$JBoss_HOME/server/default/deploy/oracle-ds.xml`** directory on the server. Each deployer file needs to end with the suffix **`-ds.xml`**. For instance, an Oracle datasource deployer might be named **`oracle-ds.xml`**. If files are not named properly, they are not found by the server.

C.2. DB2

Example C.1. DB2 Local-XA

Copy the **`$db2_install_dir/java/db2jcc.jar`** and **`$db2_install_dir/java/db2jcc_license_cu.jar`** files into the **`$jboss_install_dir/server/default/lib`** directory. The **`db2java.zip`** file, which is part of the legacy CLI driver, is normally not required when using the DB2 Universal JDBC driver included in DB2 v8.1 and later.

```
<datasources>

  <local-tx-datasource>
    <jndi-name>DB2DS</jndi-name>
    <!-- Use the syntax 'jdbc:db2:yourdatabase' for jdbc type 2
connection -->
    <!-- Use the syntax 'jdbc:db2://serveraddress:port/yourdatabase' for
jdbc type 4 connection -->
    <connection-
url>jdbc:db2://serveraddress:port/yourdatabase</connection-url>
    <driver-class>com.ibm.db2.jcc.DB2Driver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
    -->

    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example C.2. DB2 XA

Copy the `$db2_install_dir/java/db2jcc.jar` and `$db2_install_dir/java/db2jcc_license_cu.jar` files into the `$jboss_install_dir/server/default/lib` directory.

The `db2java.zip` file is required when using the DB2 Universal JDBC driver (type 4) for XA on DB2 v8.1 fixpak 14 (and the corresponding DB2 v8.2 fixpak 7).

```
<datasources>
  <!--
    XADatasource for DB2 v8.x (app driver)
  -->

  <xa-datasource>
    <jndi-name>DB2XADS</jndi-name>

    <xa-datasource-class>com.ibm.db2.jcc.DB2XADataSource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">your_server_address</xa-
datasource-property>
    <xa-datasource-property name="PortNumber">your_server_port</xa-
datasource-property>
    <xa-datasource-property name="DatabaseName">your_database_name</xa-
datasource-property>
    <!-- DriverType can be either 2 or 4, but you most likely want to
use the JDBC type 4 as it doesn't require a DB" client -->
    <xa-datasource-property name="DriverType">4</xa-datasource-
property>
    <!-- If driverType 4 is used, the following two tags are needed -->
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>>false</isSameRM-override-value>

    <xa-datasource-property name="User">your_user</xa-datasource-
property>
    <xa-datasource-property name="Password">your_password</xa-
datasource-property>

    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2</type-mapping>
    </metadata>
  </xa-datasource>
</datasources>
```

Example C.3. DB2 on AS/400

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

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

<!-- $Id: db2-400-ds.xml,v 1.1.4.2 2004/10/27 18:44:10 pilhuhn Exp $ -->

<!-- You need the jt400.jar that is delivered with IBM iSeries Access or
the
OpenSource Project jtopen.

[systemname] Hostame of the iSeries
[schema]      Default schema is needed so jboss could use metadat to
test if the tables exists
-->

<datasources>
  <local-tx-datasource>
    <jndi-name>DB2-400</jndi-name>
    <connection-url>jdbc:as400://[systemname]/[schema];extended
dynamic=true;package=jbpkg;package cache=true;package
library=jboss;errors=full</connection-url>
    <driver-class>com.ibm.as400.access.AS400JDBCDriver</driver-class>
    <user-name>[username]</user-name>
    <password>[password]</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
    -->
    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2/400</type-mapping>
    </metadata>

  </local-tx-datasource>
</datasources>

```

Example C.4. DB2 on AS/400 "native"

The *Native* JDBC driver is shipped as part of the IBM Developer Kit for Java (57xxJV1). It is implemented by making native method calls to the SQL *CLI* (*Call Level Interface*), and it only runs on the i5/OS JVM. The class name to register is **com.ibm.db2.jdbc.app.DB2Driver**. The URL subprotocol is **db2**. See JDBC FAQKS at <http://www-03.ibm.com/systems/i/software/toolbox/faqjdbc.html#faqA1> for more information.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
===== --
>
<!--
-->
<!-- JBoss Server Configuration
-->
<!--
-->
<!--
===== --
>
<!-- $Id: db2-400-ds.xml,v 1.1.4.2 2004/10/27 18:44:10 pilhuhn Exp $
-->
<!-- You need the jt400.jar that is delivered with IBM iSeries Access or
the
OpenSource Project jtopen.
[systemname] Hostame of the iSeries
[schema]      Default schema is needed so jboss could use metadat to
test if the tables exists -->
<datasources>
  <local-tx-datasource>
    <jndi-name>DB2-400</jndi-name>
    <connection-url>jdbc:db2://[systemname]/[schema];extended
dynamic=true;package=jbpkg;package cache=true;package
library=jboss;errors=full</connection-url>
    <driver-class>com.ibm.db2.jdbc.app.DB2Driver</driver-class>
    <user-name>[username]</user-name>
    <password>[password]</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>      --
  >

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>      -->
    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2/400</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Tips

- This driver is sensitive to the job's CCSID, but works fine with **CCSID=37**.

- `[systemname]` must be defined as entry **WRKRDBDIRE** like ***local**.

C.3. ORACLE

Example C.5. Oracle Local-TX Datasource

```
<?xml version="1.0" encoding="UTF-8"?>

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

<!-- $Id: oracle-ds.xml,v 1.6 2004/09/15 14:37:40 loubiansky Exp $ -->
<!--
===== -->
<!-- Datasource config for Oracle originally from Steven Coy
-->
<!--
===== -->

<datasources>
  <local-tx-datasource>
    <jndi-name>OracleDS</jndi-name>
    <connection-
url>jdbc:oracle:thin:@youroraclehost:1521:yoursid</connection-url>
    <!--
    See on WIKI page below how to use Oracle's thin JDBC driver to connect
    with enterprise RAC.
    -->
    <!--
    Here are a couple of the possible OCI configurations.
    For more information, see
    http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/java.9
    20/a96654/toc.htm

    <connection-url>jdbc:oracle:oci:@youroracle-tns-name</connection-url>
    or
    <connection-url>jdbc:oracle:oci:@(description=(address=
    (host=youroraclehost)(protocol=tcp)(port=1521))(connect_data=
    (SERVICE_NAME=yourservicename)))</connection-url>

    Clearly, its better to have TNS set up properly.
    -->
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
```



```

<user-name>x</user-name>
<password>y</password>

<min-pool-size>5</min-pool-size>
<max-pool-size>100</max-pool-size>

  <!-- Uses the pingDatabase method to check a connection is still
  valid before handing it out from the pool -->
  <!--valid-connection-checker-class-
  name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker
  </valid-connection-checker-class-name-->
  <!-- Checks the Oracle error codes and messages for fatal errors -->
  <exception-sorter-class-
  name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</except
  ion-sorter-class-name>
  <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
  -->

  <!-- sql to call on an existing pooled connection when it is
  obtained from pool - the OracleValidConnectionChecker is preferred
  <check-valid-connection-sql>some arbitrary sql</check-valid-
  connection-sql>
  -->

  <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
  (optional) -->
  <metadata>
    <type-mapping>Oracle9i</type-mapping>
  </metadata>
</local-tx-datasource>

</datasources>

```

Example C.6. Oracle XA Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

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

<!-- $Id: oracle-xa-ds.xml,v 1.13 2004/09/15 14:37:40 loubiansky Exp $ -
-->

```

```

<!--
===== --
>
<!-- ATTENTION: DO NOT FORGET TO SET Pad=true IN transaction-
service.xml -->
<!--
===== --
>

<datasources>
  <xa-datasource>
    <jndi-name>XAOracleDS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>
    <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-
datasource-class>
    <xa-datasource-property name="URL">jdbc:oracle:oci8:@tc</xa-
datasource-property>
    <xa-datasource-property name="User">scott</xa-datasource-property>
    <xa-datasource-property name="Password">tiger</xa-datasource-
property>
    <!-- Uses the pingDatabase method to check a connection is still
valid before handing it out from the pool -->
    <!--valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker
</valid-connection-checker-class-name-->
    <!-- Checks the Oracle error codes and messages for fatal errors -->
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</excep
tion-sorter-class-name>
    <!-- Oracles XA datasource cannot reuse a connection outside a
transaction once enlisted in a global transaction and vice-versa -->
    <no-tx-separate-pools></no-tx-separate-pools>

    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>Oracle9i</type-mapping>
    </metadata>
  </xa-datasource>

  <mbean
code="org.jboss.resource.adapter.jdbc.vendor.OracleXAExceptionFormatter"
name="jboss.jca:service=OracleXAExceptionFormatter">
    <depends optional-attribute-
name="TransactionManagerService">jboss:service=TransactionManager</depen
ds>
  </mbean>

</datasources>

```

Example C.7. Oracle's Thin JDBC Driver with Enterprise RAC

The extra configuration to use Oracle's Thin JDBC driver to connect with Enterprise RAC involves the `<connection-url>`. The two hostnames provide load balancing and failover to the underlying physical database.

```
...
<connection-url>jdbc:oracle:thin:@(description=(address_list=
(load_balance=on)(failover=on)(address=(protocol=tcp)(host=xxxxhost1)
(port=1521))(address=(protocol=tcp)(host=xxxxhost2)(port=1521)))
(connect_data=(service_name=xxxxsid)(failover_mode=(type=select)
(method=basic))))</connection-url>
...
```



NOTE

This example has only been tested against Oracle 10g.

C.3.1. Changes in Oracle 10g JDBC Driver

It is no longer necessary to enable the **Pad** option in your `jboss-service.xml` file. Further, you no longer need the `<no-tx-seperate-pool/>`.

C.3.2. Type Mapping for Oracle 10g

You need to specify Oracle9i type mapping for Oracle 10g datasource configurations.

Example C.8. Oracle9i Type Mapping

```
....
<metadata>
  <type-mapping>Oracle9i</type-mapping>
</metadata>
....
```

C.3.3. Retrieving the Underlying Oracle Connection Object

Example C.9. Oracle Connection Object

```
Connection conn = myJBossDataSource.getConnection();
WrappedConnection wrappedConn = (WrappedConnection)conn;
Connection underlyingConn = wrappedConn.getUnderlyingConnection();
OracleConnection oracleConn = (OracleConnection)underlyingConn;
```

C.4. SYBASE

Example C.10. Sybase Datasource

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/SybaseDB</jndi-name>
    <!-- Sybase jConnect URL for the database.
    NOTE: The hostname and port are made up values. The optional
    database name is provided, as well as some additinal Driver
    parameters.
    -->
    <connection-url>jdbc:sybase:Tds:host.at.some.domain:5000/db_name?
    JCONNECT_VERSION=6</connection-url>
    <driver-class>com.sybase.jdbc2.jdbc.SybDataSource</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <exception-sorter-class-
    name>org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter</excep
    tion-sorter-class-name>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
    obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
    connection-sql>
    -->

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
    (optional) -->
    <metadata>
      <type-mapping>Sybase</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

[1]

C.5. MICROSOFT SQL SERVER

To evaluate those drivers, you can use a simple JSP page to query the **pubs** database shipped with Microsoft SQL Server.

Move the WAR archive located in [files/mssql-test.zip](#) to the **/deploy**, start the server, and navigate your web browser to <http://localhost:8080/test/test.jsp>.

Example C.11. Local-TX Datasource Using DataDirect Driver

This example uses the DataDirect Connect for JDBC drivers from <http://www.datadirect.com>.

```

<datasources>
  <local-tx-datasource>
    <jndi-name>MerliaDS</jndi-name>
    <connection-

```

```

url>jdbc:datadirect:sqlserver://localhost:1433;DatabaseName=jboss</conne
ction-url>
  <driver-class>com.ddtek.jdbc.sqlserver.SQLServerDriver</driver-
class>
  <user-name>sa</user-name>
  <password>sa</password>

  <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
  <metadata>
    <type-mapping>MS SQLSERVER2000</type-mapping>
  </metadata>
</local-tx-datasource>

</datasources>

```

Example C.12. Local-TX Datasource Using Merlia Driver

This example uses the Merlia JDBC Driver drivers from <http://www.inetsoftware.de>.

```

<datasources>
  <local-tx-datasource>
    <jndi-name>MerliaDS</jndi-name>
    <connection-url>jdbc:inetdae7:localhost:1433?
database=pubs</connection-url>
    <driver-class>com.inet.tds.TdsDataSource</driver-class>
    <user-name>sa</user-name>
    <password>sa</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>

```

Example C.13. XA Datasource Using Merlia Driver

This example uses the Merlia JDBC Driver drivers from <http://www.inetsoftware.de>.

```

<datasources>
  <xa-datasource>
    <jndi-name>MerliaXADS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>>false</isSameRM-override-value>
    <xa-datasource-class>com.inet.tds.DTCDataSource</xa-datasource-
class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-

```

```

property>
  <user-name>sa</user-name>
  <password>sa</password>

  <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
  <metadata>
    <type-mapping>MS SQLSERVER2000</type-mapping>
  </metadata>
</xa-datasource>

</datasources>

```

C.5.1. Microsoft JDBC Drivers

The Microsoft JDBC driver for MS SQL Server comes now in two flavors:

- SQL Server 2000 Driver for JDBC Service Pack 3 which can be used with SQL Server 2000
- Microsoft SQL Server 2005 JDBC Driver which be used with either SQL Server 2000 or 2005. This version contains numerous fixes and has been certified for JBoss Hibernate. This driver runs under JDK 5.

Make sure to read the **release.txt** included in the driver distribution to understand the differences between these drivers, especially the new package name introduced with 2005 and the potential conflicts when using both drivers in the same app server.

Example C.14. Microsoft SQL Server 2000 Local-TX Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>MSSQL2000DS</jndi-name>
    <connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;Databa
seName=pubs</connection-url>
    <driver-class>com.microsoft.jdbc.sqlserver.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>

```

Example C.15. Microsoft SQL Server 2005 Local-TX Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>MSSQL2005DS</jndi-name>
    <connection-
url>jdbc:sqlserver://localhost:1433;DatabaseName=pubs</connection-url>
    <driver-class>com.microsoft.sqlserver.jdbc.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

Example C.16. Microsoft SQL Server 2005 XA Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <xa-datasource>
    <jndi-name>MSSQL2005XADS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>>false</isSameRM-override-value>
    <xa-datasource-
class>com.microsoft.sqlserver.jdbc.SQLServerXADataSource</xa-datasource-
class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
    <xa-datasource-property name="SelectMethod">cursor</xa-datasource-
property>
    <xa-datasource-property name="User">sa</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-
property>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>
</datasources>

```

C.5.2. JSQL Drivers

Example C.17. JSQL Driver

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>JSQLDS</jndi-name>
    <connection-
url>jdbc:JSQLConnect://localhost:1433/databaseName=testdb</connection-
url>
    <driver-class>com.jnetdirect.jsql.JSQLDriver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
    obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
    connection-sql>
    -->

  </local-tx-datasource>
</datasources>
```

C.5.3. jTDS JDBC Driver

jTDS is an open source 100% pure Java (type 4) JDBC 3.0 driver for Microsoft SQL Server (6.5, 7, 2000 and 2005) and Sybase (10, 11, 12, 15). jTDS is based on FreeTDS and is currently the fastest production-ready JDBC driver for microsoft SQL Server and Sybase. jTDS is 100% JDBC 3.0 compatible, supporting forward-only and scrollable/updateable ResultSets, concurrent (completely independent) Statements and implementing all the **DatabaseMetaData** and **ResultSetMetaData** methods.

Download jTDS from <http://jtds.sourceforge.net/>.

Example C.18. jTDS Local-TX Datasource

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>jtdsDS</jndi-name>
    <connection-
url>jdbc:jtds:sqlserver://localhost:1433;databaseName=pubs</connection-
url>
    <driver-class>net.sourceforge.jtds.jdbc.Driver</driver-class>
```



```

<user-name>sa</user-name>
<password>jboss</password>

<!-- optional parameters -->
<transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
<min-pool-size>10</min-pool-size>
<max-pool-size>30</max-pool-size>
<idle-timeout-minutes>15</idle-timeout-minutes>
<blocking-timeout-millis>5000</blocking-timeout-millis>
<new-connection-sql>select 1</new-connection-sql>
<check-valid-connection-sql>select 1</check-valid-connection-sql>
<set-tx-query-timeout></set-tx-query-timeout>
<metadata>
  <type-mapping>MS SQLSERVER2000</type-mapping>
</metadata>
</local-tx-datasource>
</datasources>

```

Example C.19. jTDS XA Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <xa-datasource>
    <jndi-name>jtdsXADS</jndi-name>
    <xa-datasource-class>net.sourceforge.jtds.jdbcx.JtdsDataSource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
    <xa-datasource-property name="User">sa</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-
property>

    <!--
      When set to true, emulate XA distributed transaction support. Set to
      false to use experimental
      true distributed transaction support. True distributed transaction
      support is only available for
      SQL Server 2000 and requires the installation of an external stored
      procedure in the target server
      (see the README.XA file in the distribution for details).
    -->
    <xa-datasource-property name="XaEmulation">true</xa-datasource-
property>

    <track-connection-by-tx></track-connection-by-tx>

    <!-- optional parameters -->
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
    <min-pool-size>10</min-pool-size>

```

```

<max-pool-size>30</max-pool-size>
<idle-timeout-minutes>15</idle-timeout-minutes>
<blocking-timeout-millis>5000</blocking-timeout-millis>
<new-connection-sql>select 1</new-connection-sql>
<check-valid-connection-sql>select 1</check-valid-connection-sql>
<set-tx-query-timeout></set-tx-query-timeout>
<metadata>
  <type-mapping>MS SQLSERVER2000</type-mapping>
</metadata>
</xa-datasource>
</datasources>

```

C.5.4. "Invalid object name 'JMS_SUBSCRIPTIONS' Exception

If you receive an exception like the one in [Example C.20, “JMS_SUBSCRIPTIONS Exception”](#) during startup, specify a **SelectMethod** in the connection URL, as shown in [Example C.21, “Specifying a SelectMethod”](#).

Example C.20. JMS_SUBSCRIPTIONS Exception

```

17:17:57,167 WARN [ServiceController] Problem starting service
jboss.mq.destination:name=testTopic,service=Topic
org.jboss.mq.SpyJMSException: Error getting durable subscriptions for
topic TOPIC.testTopic; - nested throwable: (java.sql.SQLException:
[Microsoft][SQLServer 2000 Driver for JDBC][SQLServer]Invalid object
name 'JMS_SUBSCRIPTIONS'.)
    at
org.jboss.mq.sm.jdbc.JDBCStateManager.getDurableSubscriptionIdsForTopic(
JDBCStateManager.java:290)
    at
org.jboss.mq.server.JMSDestinationManager.addDestination(JMSDestinationM
anager.java:656)

```

Example C.21. Specifying a SelectMethod

```

<connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;Databa
seName=jboss</connection-url>

```

C.6. MYSQL DATASOURCE

C.6.1. Installing the Driver

Procedure C.1. Installing the Driver

1. Download the driver from <http://www.mysql.com/products/connector/j/>. Make sure to choose the driver based on your version of MySQL.

2. Expand the driver ZIP or TAR file, and locate the `.jar` file.
3. Move the `.jar` file into `$JBOSS_HOME/server/config_name/lib`.
4. Copy the `$JBOSS_HOME/docs/examples/jca/mysql-ds.xml` example datasource deployer file to `$JBOSS_HOME/server/config_name/deploy/`, for use as a template.

C.6.2. MySQL Local-TX Datasource

Example C.22. MySQL Local-TX Datasource

This example uses a database hosted on `localhost`, on port 3306, with `autoReconnect` enabled. This is not a recommended configuration, unless you do not need any Transactions support.

```
<datasources>
  <local-tx-datasource>

    <jndi-name>MySQLDS</jndi-name>

    <connection-url>jdbc:mysql://localhost:3306/database</connection-
url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>

    <user-name>username</user-name>
    <password>secret</password>

    <connection-property name="autoReconnect">true</connection-
property>

    <!-- Typemapping for JBoss 4.0 -->
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>

  </local-tx-datasource>
</datasources>
```

C.6.3. MySQL Using a Named Pipe

Example C.23. MySQL Using a Named Pipe

This example uses a database hosted locally, but uses a named pipe instead of TCP/IP.

```
<datasources>
  <local-tx-datasource>

    <jndi-name>MySQLDS</jndi-name>
    <connection-url>jdbc:mysql://./database</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>

    <user-name>username</user-name>
    <password>secret</password>
```

```

        <connection-property
name="socketFactory">com.mysql.jdbc.NamedPipeSocketFactory</connection-
property>

        <metadata>
        <type-mapping>mySQL</type-mapping>
        </metadata>

    </local-tx-datasource>
</datasources>

```

C.7. POSTGRESQL

Example C.24. PostgreSQL Local-TX Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>PostgresDS</jndi-name>
    <connection-url>jdbc:postgresql://[servername]:[port]/[database
name]</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
    obtained from pool
    <check-valid-connection-sql>some arbitrary sql</check-valid-
    connection-sql>
    -->

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
    (optional) -->
    <metadata>
      <type-mapping>PostgreSQL 7.2</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

Example C.25. PostgreSQL XA Datasource

This configuratino works for PostgreSQL 8.x and later.

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <xa-datasource>

```

```

<jndi-name>PostgresDS</jndi-name>

<xa-datasource-class>org.postgresql.xa.PGXADatasource</xa-
datasource-class>
<xa-datasource-property name="ServerName">[servername]</xa-
datasource-property>
<xa-datasource-property name="PortNumber">5432</xa-datasource-
property>

<xa-datasource-property name="DatabaseName">[database name]</xa-
datasource-property>
<xa-datasource-property name="User">[username]</xa-datasource-
property>
<xa-datasource-property name="Password">[password]</xa-datasource-
property>

<track-connection-by-tx></track-connection-by-tx>
</xa-datasource>
</datasources>

```

C.8. INGRES

Example C.26. Ingres Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>IngresDS</jndi-name>
    <use-java-context>false</use-java-context>
    <driver-class>com.ingres.jdbc.IngresDriver</driver-class>
    <connection-url>jdbc:ingres://localhost:II7/testdb</connection-url>
    <datasource-class>com.ingres.jdbc.IngresDataSource</datasource-
class>
    <datasource-property name="ServerName">localhost</datasource-
property>
    <datasource-property name="PortName">II7</datasource-property>
    <datasource-property name="DatabaseName">testdb</datasource-
property>
    <datasource-property name="User">testuser</datasource-property>
    <datasource-property name="Password">testpassword</datasource-
property>
    <new-connection-sql>select count(*) from iitables</new-connection-
sql>

    <check-valid-connection-sql>select count(*) from iitables</check-
valid-connection-sql>
    <metadata>
      <type-mapping>Ingres</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

[2]

[1] Source: <http://community.jboss.org/wiki/SetUpASybaseDatasource>

[2] Source: <http://community.ingres.com>

APPENDIX D. REVISION HISTORY

Revision 4.3.10-101.400 Rebuild with publican 4.0.0	2013-10-31	Rüdiger Landmann
Revision 4.3.10-101 Rebuild for Publican 3.0	2012-07-18	Anthony Towns
Revision 4.3.10-100 Incorporated changes for the Enterprise Application Platform 4.3.0CP10 release. For more information, refer to the Documentation Resolved Issues in the <i>Release Notes CP10</i> .	Mon Aug 29 2011	Jared Morgan
Revision 4.3.9-100 Incorporated changes for the Enterprise Application Platform 4.3.0CP09 release. For more information, refer to the Documentation Resolved Issues in the <i>Release Notes CP09</i> .	Tue Nov 30 2010	Jared Morgan