



JBoss Enterprise Web Platform 5 Administration And Configuration Guide

for use with JBoss Enterprise Web Platform 5
Edition 5.1.1

JBoss Community

JBoss Enterprise Web Platform 5 Administration And Configuration Guide

for use with JBoss Enterprise Web Platform 5
Edition 5.1.1

JBoss Community

Edited by

JBoss Community

Isaac Rooskov

Laura Bailey

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, MetaMatrix, 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 the administration and configuration of JBoss Enterprise Web Platform 5 and its patch releases.

Table of Contents

Preface	10
1. Document Conventions	10
1.1. Typographic Conventions	10
1.2. Pull-quote Conventions	11
1.3. Notes and Warnings	12
2. Getting Help and Giving Feedback	12
2.1. Do You Need Help?	12
2.2. Give us Feedback	13
What this Book Covers	14
Chapter 1. Introduction	15
1.1. JBoss Enterprise Web Platform Use Cases	15
Part I. JBoss Enterprise Web Platform Infrastructure	16
Chapter 2. JBoss Enterprise Web Platform 5 Architecture	17
2.1. The JBoss Enterprise Web Platform Bootstrap	17
2.2. Hot Deployment	18
Part II. JBoss Enterprise Web Platform 5 Configuration	19
Chapter 3. Network	20
3.1. IPv6 Support	20
Chapter 4. Logging	21
4.1. Logging Defaults	21
4.2. Component-Specific Logging	21
4.2.1. SQL Logging with Hibernate	22
4.2.2. Transaction Service Logging	22
Chapter 5. Deployment	23
5.1. Deployable Application Types	23
5.1.1. Exploded Deployment	24
5.2. Standard Server Profiles	24
Chapter 6. Microcontainer	26
6.1. An overview of the Microcontainer modules	26
6.2. Configuration	27
6.3. References	29
Chapter 7. The JNDI Naming Service	30
7.1. An Overview of JNDI	30
7.1.1. Names	30
7.1.2. Contexts	31
7.1.2.1. Obtaining a Context using InitialContext	31
7.2. The JBoss Naming Service Architecture	32
7.3. The Naming InitialContext Factories	34
7.3.1. The standard naming context factory	34
7.3.2. The org.jboss.naming.NamingContextFactory	35
7.3.3. Naming Discovery in Clustered Environments	35
7.3.4. The HTTP InitialContext Factory Implementation	36
7.3.5. The Login InitialContext Factory Implementation	36
7.3.6. The ORBInitialContextFactory	37

7.4. JNDI over HTTP	37
7.4.1. Accessing JNDI over HTTP	37
7.4.2. Accessing JNDI over HTTPS	40
7.4.3. Securing Access to JNDI over HTTP	42
7.4.4. Securing Access to JNDI with a Read-Only Unsecured Context	44
7.5. Additional Naming MBeans	46
7.5.1. JNDI Binding Manager	46
7.5.2. The org.jboss.naming.NamingAlias MBean	47
7.5.3. org.jboss.naming.ExternalContext MBean	47
7.5.4. The org.jboss.naming.JNDIView MBean	49
7.6. J2EE and JNDI - The Application Component Environment	51
7.6.1. ENC Usage Conventions	52
7.6.1.1. Environment Entries	52
7.6.1.2. EJB References	54
7.6.1.3. EJB References with jboss.xml and jboss-web.xml	55
7.6.1.4. EJB Local References	56
7.6.1.5. Resource Manager Connection Factory References	58
7.6.1.6. Resource Manager Connection Factory References with jboss.xml and jboss-web.xml	60
7.6.1.7. Resource Environment References	59
7.6.1.8. Resource Environment References and jboss.xml, jboss-web.xml	61
Chapter 8. Web Services	62
8.1. The need for web services	62
8.2. What web services are not	62
8.3. Document/Literal	62
8.4. Document/Literal (Bare)	63
8.5. Document/Literal (Wrapped)	64
8.6. RPC/Literal	64
8.7. RPC/Encoded	65
8.8. Web Service Endpoints	66
8.9. Plain old Java Object (POJO)	66
8.10. The endpoint as a web application	66
8.11. Packaging the endpoint	66
8.12. Accessing the generated WSDL	67
8.13. EJB3 Stateless Session Bean (SLSB)	67
8.14. Endpoint Provider	68
8.15. WebServiceContext	68
8.16. Web Service Clients	69
8.16.1. Service	69
8.16.1.1. Service Usage	69
8.16.1.2. Handler Resolver	70
8.16.1.3. Executor	70
8.16.2. Dynamic Proxy	70
8.16.3. WebServiceRef	71
8.16.4. Dispatch	73
8.16.5. Asynchronous Invocations	74
8.16.6. Oneway Invocations	74
8.17. Common API	75
8.17.1. Handler Framework	75
8.17.1.1. Logical Handler	75
8.17.1.2. Protocol Handler	75
8.17.1.3. Service endpoint handlers	75
8.17.1.4. Service client handlers	76
8.17.2. Message Context	76
8.17.2.1. Accessing the message context	76
8.17.2.2. Logical Message Context	76

8.17.2.3. SOAP Message Context	76
8.17.3. Fault Handling	77
8.18. DataBinding	77
8.18.1. Using JAXB with non annotated classes	77
8.19. Attachments	77
8.19.1. MTOM/XOP	77
8.19.1.1. Supported MTOM parameter types	78
8.19.1.2. Enabling MTOM per endpoint	78
8.19.2. SwaRef	79
8.19.2.1. Using SwaRef with JAX-WS endpoints	79
8.19.2.2. Starting from WSDL	80
8.20. Tools	80
8.20.1. Bottom-Up (Using wsprovide)	81
8.20.2. Top-Down (Using wsconsume)	83
8.20.3. Client Side	84
8.20.4. Command-line & Ant Task Reference	86
8.20.5. JAX-WS binding customization	87
8.21. Web Service Extensions	87
8.21.1. WS-Addressing	87
8.21.1.1. Specifications	87
8.21.1.2. Addressing Endpoint	87
8.21.1.3. Addressing Client	87
8.21.2. WS-Security	89
8.21.2.1. Endpoint configuration	90
8.21.2.2. Server side WSSE declaration (jboss-wsse-server.xml)	90
8.21.2.3. Client side WSSE declaration (jboss-wsse-client.xml)	91
8.21.2.3.1. Client side key store configuration	91
8.21.2.4. Installing the BouncyCastle JCE provider	92
8.21.2.5. Username Token Authentication	92
8.21.2.5.1. Secure Transport	95
8.21.2.6. X509 Certificate Token	95
8.21.2.7. JAAS Integration	98
8.21.2.8. POJO Endpoint Authentication and Authorization	100
8.21.3. XML Registries	102
8.21.3.1. Apache jUDDI Configuration	102
8.21.3.2. JBoss JAXR Configuration	103
8.21.3.3. JAXR Sample Code	103
8.21.3.4. Troubleshooting	106
8.21.3.5. Resources	107
8.22. JBossWS Extensions	107
8.22.1. Proprietary Annotations	107
8.22.1.1. EndpointConfig	107
8.22.1.2. WebContext	107
8.22.1.3. SecurityDomain	109
8.23. Web Services Appendix	109
8.24. References	109
Chapter 9. JBoss AOP	110
9.1. Some key terms	110
9.2. Creating Aspects in JBoss AOP	111
9.3. Applying Aspects in JBoss AOP	112
9.4. Packaging AOP Applications	113
9.5. The JBoss AspectManager Service	114
9.6. Loadtime transformation in the JBoss Enterprise Web Platform Using Sun JDK	115
9.7. JRockit	116
9.8. Improving Loadtime Performance in the JBoss Enterprise Web Platform Environment	116

9.9. Scoping the AOP to the classloader	116
9.9.1. Deploying as part of a scoped classloader	117
9.9.2. Attaching to a scoped deployment	117
Chapter 10. Transaction Management	118
10.1. Overview	118
10.2. Configuration Essentials	118
10.3. Transactional Resources	119
10.4. Last Resource Commit Optimization (LRCO)	120
10.5. Transaction Timeout Handling	121
10.6. Recovery Configuration	121
10.7. Transaction Service FAQ	121
Chapter 11. Use Alternative Databases with JBoss Enterprise Platform	123
11.1. How to Use Alternative Databases	123
11.2. Install JDBC Drivers	123
11.2.1. Special Notes on Sybase	124
11.2.1.1. Enable JAVA services	124
11.2.1.2. CMP Configuration	124
11.2.1.3. Installing Java Classes	124
11.2.2. Configuring JDBC DataSources	125
11.3. Common Database-Related Tasks	125
11.3.1. Security and Pooling	125
11.3.2. Change Database for the JMS Services	125
11.3.3. Support Foreign Keys in CMP Services	125
11.3.4. Specify Database Dialect for Java Persistence API	126
11.3.5. Change Other JBoss Enterprise Platform Services to use the External Database	126
11.3.5.1. The Easy Way	126
11.3.5.2. The More Flexible Way	126
11.3.6. A Special Note About Oracle Databases	127
Chapter 12. Datasource Configuration	129
12.1. Types of Datasources	129
12.2. Datasource Parameters	129
12.3. Datasource Examples	135
12.3.1. Generic Datasource Example	135
12.3.2. Configuring a DataSource for Remote Usage	138
12.3.3. Configuring a Datasource to Use Login Modules	139
Chapter 13. Pooling	140
13.1. Strategy	140
13.2. Sticky Transactions	140
13.3. Workaround for Oracle	140
13.4. Pool Access	140
13.5. Pool Filling	141
13.6. Idle Connections	141
13.7. Dead connections	141
13.7.1. Valid connection checking	142
13.7.2. Errors during SQL queries	142
13.7.3. Changing/Closing/Flushing the pool	142
Chapter 14. Frequently Asked Questions	143
14.1. I have problems with Oracle XA	143
Part III. Clustering Guide	144
Chapter 15. Introduction and Quick Start	145

15.1. Quick Start Guide	145
15.1.1. Initial Preparation	145
15.1.2. Launching a JBoss Enterprise Web Platform Cluster	146
15.1.3. Web Application Clustering Quick Start	148
Chapter 16. Clustering Concepts	149
16.1. Cluster Definition	149
16.2. Service Architectures	150
16.2.1. Client-side interceptor architecture	150
16.2.2. External Load Balancer Architecture	151
16.3. Load Balancing Policies	152
16.3.1. Client-side interceptor architecture	152
16.3.2. External load balancer architecture	153
Chapter 17. Clustering Building Blocks	154
17.1. Group Communication with JGroups	154
17.1.1. The Channel Factory Service	155
17.1.1.1. Standard Protocol Stack Configurations	155
17.1.2. The JGroups Shared Transport	156
17.1.3. Distributed Caching with JBoss Cache	157
17.1.3.1. The JBoss Enterprise Web Platform CacheManager Service	158
17.1.3.1.1. Standard Cache Configurations	158
17.1.3.1.2. Cache Configuration Aliases	160
17.1.4. The HAPartition Service	160
17.1.4.1. DistributedReplicantManager Service	163
17.1.4.2. Custom Use of HAPartition	163
Chapter 18. Clustered JNDI Services	164
18.1. How it works	164
18.2. Client configuration	165
18.2.1. For clients running inside the Enterprise Web Platform	165
18.2.1.1. Accessing HA-JNDI Resources from WARs -- Environment Naming Context	166
18.2.1.2. How can I tell if things are being bound into HA-JNDI that shouldn't be?	166
18.2.2. For clients running outside the Enterprise Web Platform	167
18.3. JBoss configuration	168
18.3.1. Adding a Second HA-JNDI Service	172
Chapter 19. Clustered Session EJBs	173
19.1. Stateless Session Bean in EJB 3.0	173
19.2. Stateful Session Beans in EJB 3.0	174
19.2.1. The EJB application configuration	175
19.2.2. Optimize state replication	176
19.2.3. CacheManager service configuration	176
19.3. Stateless Session Bean in EJB 2.x	179
19.4. Stateful Session Bean in EJB 2.x	180
19.4.1. The EJB application configuration	180
19.4.2. Optimize state replication	181
19.4.3. The HASessionStateService configuration	181
19.4.4. Handling Cluster Restart	182
19.4.5. JNDI Lookup Process	182
19.4.6. SingleRetryInterceptor	183
Chapter 20. Clustered Entity EJBs	184
20.1. Entity Bean in EJB 3.0	184
20.1.1. Configure the distributed cache	184
20.1.2. Configure the entity beans for cache	187
20.1.3. Query result caching	189

20.2. Entity Bean in EJB 2.x	193
Chapter 21. HTTP Services	195
21.1. Configuring load balancing using Apache and mod_jk	195
21.1.1. Download the software	195
21.1.2. Configure Apache to load mod_jk	195
21.1.3. Configure worker nodes in mod_jk	197
21.1.4. Configuring JBoss to work with mod_jk	198
21.1.5. Configuring the NSAPI connector on Solaris	199
21.1.5.1. Prerequisites	199
21.1.5.2. Configure JBoss Enterprise Platform as a Worker Node	199
21.1.5.3. Configure Sun Java System Web Server for Clustering	200
21.1.5.3.1. Configure a basic cluster with NSAPI	201
21.1.5.3.2. Configure a Load-balanced Cluster with NSAPI	202
21.1.5.3.3. Restart Sun Java System Web Server	203
21.2. Configuring HTTP session state replication	204
21.2.1. Enabling session replication in your application	204
21.2.2. HttpSession Passivation and Activation	207
21.2.2.1. Configuring HttpSession Passivation	208
21.2.3. Configuring the JBoss Cache instance used for session state replication	209
21.3. Using FIELD-level replication	210
21.4. Using Clustered Single Sign-on (SSO)	212
21.4.1. Configuration	212
21.4.2. SSO Behavior	213
21.4.3. Limitations	213
21.4.4. Configuring the Cookie Domain	214
Chapter 22. Clustered Deployment Options	215
22.1. Clustered Singleton Services	215
22.1.1. HASingleton Deployment Options	215
22.1.1.1. HASingletonDeployer service	216
22.1.1.2. POJO deployments using HASingletonController	216
22.1.1.3. HASingleton deployments using a Barrier	218
22.1.2. Determining the master node	218
22.1.2.1. HA singleton election policy	219
22.2. Farming Deployment	219
Chapter 23. JGroups Services	222
23.1. Configuring a JGroups Channel's Protocol Stack	222
23.1.1. Common Configuration Properties	225
23.1.2. Transport Protocols	225
23.1.2.1. UDP configuration	225
23.1.2.2. TCP configuration	229
23.1.2.3. TUNNEL configuration	230
23.1.3. Discovery Protocols	231
23.1.3.1. PING	231
23.1.3.2. TCPGOSSIP	232
23.1.3.3. TCPPING	233
23.1.3.4. MPING	233
23.1.4. Failure Detection Protocols	234
23.1.4.1. FD	234
23.1.4.2. FD_SOCKET	235
23.1.4.3. VERIFY_SUSPECT	235
23.1.4.4. FD versus FD_SOCKET	235
23.1.5. Reliable Delivery Protocols	236
23.1.5.1. UNICAST	237

23.1.5.2. NAKACK	237
23.1.6. Group Membership (GMS)	238
23.1.7. Flow Control (FC)	238
23.2. Fragmentation (FRAG2)	240
23.3. State Transfer	240
23.4. Distributed Garbage Collection (STABLE)	241
23.5. Merging (MERGE2)	241
23.6. Other Configuration Issues	242
23.6.1. Binding JGroups Channels to a Particular Interface	242
23.6.2. Isolating JGroups Channels	243
23.6.2.1. Isolating sets of Application Server instances from each other	243
23.6.2.2. Isolating Channels for Different Services on the Same Set of AS Instances	244
23.6.2.2.1. Changing the Group Name	244
23.6.2.2.2. Changing the multicast address and port	244
23.6.2.2.3. Changing the Multicast Port	245
23.6.2.3. Improving UDP Performance by Configuring OS UDP Buffer Limits	245
23.6.3. JGroups Troubleshooting	246
23.6.3.1. Nodes do not form a cluster	246
23.6.3.2. Causes of missing heartbeats in FD	247
Chapter 24. JBoss Cache Configuration and Deployment	248
24.1. Key JBoss Cache Configuration Options	248
24.1.1. Editing the CacheManager Configuration	248
24.1.2. Cache Mode	253
24.1.3. Transaction Handling	255
24.1.4. Concurrent Access	255
24.1.5. JGroups Integration	256
24.1.6. Eviction	257
24.1.7. Cache Loaders	257
24.1.7.1. CacheLoader Configuration for Web Session and SFSB Caches	258
24.1.8. Buddy Replication	259
24.2. Deploying Your Own JBoss Cache Instance	260
24.2.1. Deployment Via the CacheManager Service	260
24.2.1.1. Accessing the CacheManager	261
24.2.2. Deployment Via a -service.xml File	263
24.2.3. Deployment Via a -jboss-beans.xml File	264
Part IV. Performance Tuning	266
Chapter 25. JBoss Enterprise Web Platform 5 Performance Tuning	267
25.1. Introduction	267
25.2. Hardware tuning	267
25.2.1. CPU (Central Processing Unit)	267
25.2.2. RAM (Random Access Memory)	267
25.2.3. Hard Disk	267
25.3. Operating System Performance Tuning	268
25.3.1. Networking	268
25.4. Tuning the JVM	268
25.5. Tuning your applications	268
25.5.1. Instrumentation	269
25.6. Tuning JBoss Enterprise Web Platform	269
25.6.1. Memory usage	270
25.6.1.1. VFS Tuning	270
25.6.1.1.1. VFS Cache Tuning	271
25.6.1.1.2. Annotation Scanning Tuning	272
25.6.2. Database Connection	273

25.6.3. Clustering Tuning	274
25.6.3.1. Ensuring Adequate Network Buffers	274
25.6.3.2. Isolating Intra-Cluster Traffic	274
25.6.3.3. JGroups Message Bundling	274
25.6.3.4. Enabling Buddy Replication for Session Caches	275
25.6.3.5. Reducing the Volume of Web Session Replication	276
25.6.3.6. Reducing the Volume of EJB3 Stateful Session Bean Replication	276
25.6.3.7. Be Cautious with JPA/Hibernate Second Level Caching	276
25.6.3.8. Monitoring JGroups via JMX	276
25.6.4. Other key configurations	277
Part V. Appendices	279
Vendor-Specific Datasource Definitions	280
A.1. Deployer Location and Naming	280
A.2. DB2	280
A.3. Oracle	283
A.3.1. Changes in Oracle 10g JDBC Driver	288
A.3.2. Type Mapping for Oracle 10g	288
A.3.3. Retrieving the Underlying Oracle Connection Object	288
A.3.4. Limitations of Oracle 11g	288
A.4. Sybase	288
A.4.1. Sybase Limitations	289
A.5. Microsoft SQL Server	290
A.5.1. Microsoft JDBC Drivers	291
A.5.2. JSQL Drivers	293
A.5.3. jTDS JDBC Driver	293
A.5.4. "Invalid object name 'JMS_SUBSCRIPTIONS' Exception	295
A.5.5. New dialect for Microsoft SQL Server 2008 with JDBC 3.x drivers	296
A.6. MySQL Datasource	296
A.6.1. Installing the Driver	296
A.6.2. MySQL Local-TX Datasource	297
A.6.3. MySQL Using a Named Pipe	297
A.7. PostgreSQL	297
A.8. Ingres	299
Logging Information and Recipes	301
B.1. Log Level Descriptions	301
B.2. Separate Log Files Per Application	301
B.3. Redirecting Category Output	302
Revision History	304

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](#) set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later include the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, select the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy**

button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic* or *Proportional Bold Italic

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — *username*, *domain.name*, *file-system*, *package*, *version* and *release*. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```

package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome         home   = (EchoHome) ref;
        Echo              echo   = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}

```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at <http://access.redhat.com>. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **JBoss Enterprise Application Platform 5** and the component **doc-Admin_and_Config_Guide**. The following link will take you to a pre-filled bug report for this product: <http://bugzilla.redhat.com/>.

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:

Be sure to give us your name so that you can receive full credit for reporting the issue.

What this Book Covers

The primary focus of this book is the presentation of the standard JBoss Enterprise Web Platform 5.0 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. This book is not an introduction to JavaEE or how to use JavaEE in applications. It focuses on the internal details of the JBoss server architecture and how our implementation of a given JavaEE 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.

Chapter 1. Introduction

JBoss Enterprise Web Platform is built on top of the new JBoss Microcontainer. The JBoss Microcontainer is a lightweight container that supports direct deployment, configuration and lifecycle of plain old Java objects (POJOs). JBoss Microcontainer replaces the JBoss JMX Microkernel.

The JBoss Microcontainer integrates nicely with the JBoss Aspect Oriented Programming framework (JBoss AOP). JBoss AOP is discussed in [Chapter 9, JBoss AOP](#). Support for JMX in JBoss Enterprise Web Platform remains strong and MBean services written against the old Microkernel are expected to work.

The Seam Booking sample application is Java EE 5 compliant. It demonstrates many of the technologies available on the Enterprise Web Platform, such as:

- EJB3
- Stateful Session Beans
- Stateless Session Beans
- JPA (w/ Hibernate validation)
- JSF
- Facelets
- Ajax4JSF
- Seam

Many key features of JBoss Enterprise Web Platform 5 are provided by integrating standalone JBoss projects which include:

- JBoss EJB3 included with JBoss Enterprise Web Platform 5 provides the implementation of the latest revision of the Enterprise Java Beans (EJB) specification. EJB 3.0 is a deep overhaul and simplification of the EJB specification. EJB 3.0's goals are to simplify development, facilitate a test driven approach, and focus more on writing plain old java objects (POJOs) rather than coding against complex EJB APIs.
- JBoss Cache comes in two flavors: a traditional tree-structured node-based cache, and a PojoCache, an in-memory, transactional, and replicated cache system that allows users to operate on simple POJOs transparently without active user management of either replication or persistency aspects.
- JBoss Web Services 3.x is the web services stack for JBoss Enterprise Web Platform, providing Java EE compatible web services, JAXWS-2.x, etc.
- JBoss Transactions is the default transaction manager for JBoss Enterprise Web Platform 5. JBoss Transactions is founded on industry proven technology and 18 year history as a leader in distributed transactions, and is one of the most interoperable implementations available.
- JBoss Web is the Web container in JBoss Enterprise Web Platform 5, an implementation based on Apache Tomcat that includes the Apache Portable Runtime (APR) and Tomcat native technologies to achieve scalability and performance characteristics that match and exceed the Apache Http server.

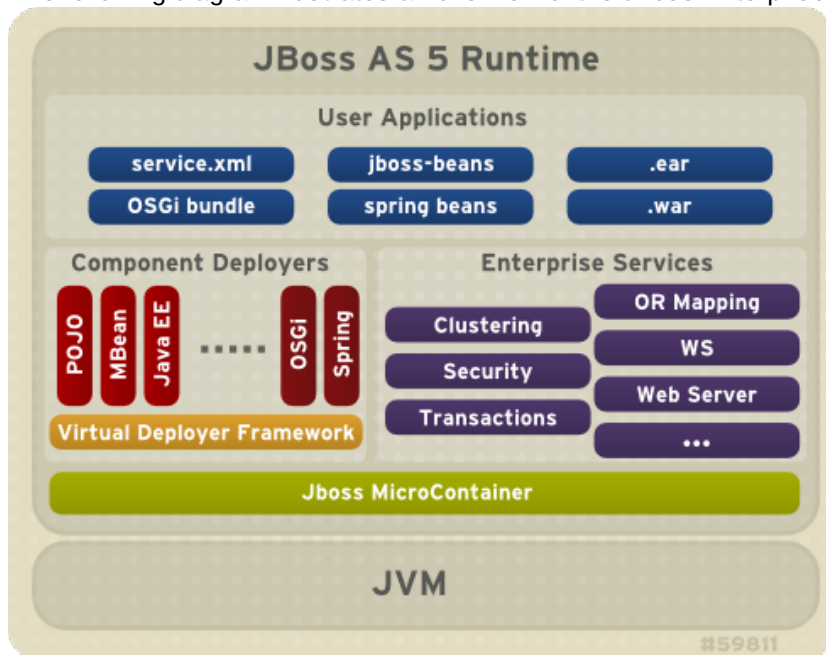
1.1. JBoss Enterprise Web Platform Use Cases

- 99% of web applications involving a database
- Mission critical web applications likely to be clustered.
- Simple web applications with JSPs/Servlets upgrades to JBoss Enterprise Web Platform with Tomcat Embedded.
- Intermediate web applications with JSPs/Servlets using a web framework such as Struts, Java Server Faces, Cocoon, Tapestry, Spring, Expresso, Avalon, Turbine.
- Complex web applications with JSPs/Servlets, Seam, Enterprise Java Beans (EJB), caching, etc.
- Cross application middleware (Corba, JMX, etc.)

Part I. JBoss Enterprise Web Platform Infrastructure

Chapter 2. JBoss Enterprise Web Platform 5 Architecture

The following diagram illustrates an overview of the JBoss Enterprise Web Platform and its components.



The directory structure of JBoss Enterprise Web Platform is outlined here:

```
-jboss-as-web - the path to your JBoss Enterprise Web Platform application server.
|-- bin - contains start scripts and run.jar
|-- client - client jars
|-- common/lib - static jars shared across server configuration
|-- docs - schemas/dtds, examples
|-- lib - core bootstrap jars
|   lib/endorsed - added to the server JVM java.endorsed.dirs path
|-- server - server configuration/profile directories. See Section 3.2
               for details of the server profiles included in this release.
```

```
-seam - the path to JBoss Seam application framework
|-- bootstrap
|-- build
|-- examples - examples demonstrating uses of Seam's features
|-- extras
|-- lib - library directory
|-- seam-gen - command-line utility used to generate simple skeletal Seam code
to get your project started
|-- ui -
```

```
-resteasy - RESTEasy - a portable implementation of JSR-311 JAX-RS Specification
|-- embedded-lib
|-- lib
|-- resteasy-jaxrs.war
```

2.1. The JBoss Enterprise Web Platform Bootstrap

The `org.jboss.Main` entry point in JBoss Enterprise Web Platform 5 loads an `org.jboss.system.server.Server` implementation. This is a JBoss Microcontainer.

The default JBoss Enterprise Web Platform 5 `org.jboss.system.server.Server` implementation is `org.jboss.bootstrap.microcontainer.ServerImpl`. This implementation is an extension of the kernel basic bootstrap that boots the MC from the bootstrap beans declared in

<**jboss.server.config.url**>/**bootstrap.xml** descriptors using a **BasicXMLDeployer**. In addition, the **ServerImpl** registers install callbacks for any beans that implement the **org.jboss.bootstrap.spi.Bootstrap** interface. The **bootstrap/profile.xml** configurations include a **ProfileServiceBootstrap** bean that implements the **Bootstrap** interface.

The **org.jboss.system.server.profileservice.ProfileServiceBootstrap** is an implementation of the **org.jboss.bootstrap.spi.Bootstrap** interface that loads the deployments associated with the current profile. The **\$PROFILE** is the name of the server configuration profile being loaded and corresponds to the **server -c** command line argument. The default **\$PROFILE** is **default**.

2.2. Hot Deployment

Hot deployment in JBoss Enterprise Web Platform 5 is controlled by the **Profile** implementations associated with the **ProfileService**. The **HDScanner** bean deployed via the **deploy/hdscanner-jboss-beans.xml** MC deployment, queries the profile service for changes in application directory contents and redeploys updated content, undeploys removed content, and adds new deployment content to the current profile via the **ProfileService**.

Disabling hot deployment is achieved by removing the **hdscanner-jboss-beans.xml** file from your deployment.

Part II. JBoss Enterprise Web Platform 5 Configuration

Chapter 3. 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 3.1. Network-related start-up options

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

3.1. IPv6 Support

Enterprise Web Platform 5 does not include support for IPv6, although this support is planned for the future.

Chapter 4. Logging

Logging is the most important tool to troubleshoot errors and monitor the status of the components of the Platform. **log4j** provides a familiar, flexible framework, familiar to Java developers.

[Section 4.1, “Logging Defaults”](#) contains information about customizing the default logging behavior for the Platform. See [Section 4.2, “Component-Specific Logging”](#) for additional customization. [Appendix B, Logging Information and Recipes](#) provides some logging *recipes*, which you can customize to your needs.

4.1. Logging Defaults

The **log4j** configuration is loaded from the ***JBOSS_HOME/server/PROFILE/conf/jboss-log4j.xml*** deployment descriptor. **log4j** uses *appenders* to control its logging behavior. An appender is a directive for where to log information, and how to do it. The ***jboss-log4j.xml*** file contains many sample appenders, including FILE, CONSOLE, and SMTP.

Table 4.1. Common log4j Configuration Directives

Configuration Option	Description
appender	The main appender. Gives the name and the implementing class.
errorHandler	Delegates an external class to handle exceptions passed to the logger, especially if the appender cannot write the log for some reason.
param	Options specific to the type of appender. In this instance, the <code><param></code> is the name of the file that stores the logs for the FILE appender.
layout	Controls the logging format. Tweak this to work with your log-parsing software of choice.

Example 4.1. Sample Appender

```
<appender name="FILE"
class="org.jboss.logging.appender.DailyRollingFileAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="File" value="${jboss.server.log.dir}/server.log"/>
  <param name="Append" value="true"/>
  <!-- In AS 5.0.x the server log threshold was set by a system property.
       In 5.1 and later, the system property sets the priority on the root
       logger (see <root/> below)
  <param name="Threshold" value="${jboss.server.log.threshold}"/> -->

  <!-- Rollover at midnight each day -->
  <param name="DatePattern" value="'.'yyyy-MM-dd"/>
  <layout class="org.apache.log4j.PatternLayout">
    <!-- The default pattern: Date Priority [Category] (Thread) Message\n -->
    <param name="ConversionPattern" value="%d %-5p [%c] (%t) %m%n"/>
  </layout>
</appender>
```

For more information on configuring **log4j**, see <http://logging.apache.org/log4j/1.2/>.

4.2. Component-Specific Logging

Some Platform components have extra logging options available, or extra mechanisms for customizing

logging.

4.2.1. SQL Logging with Hibernate

Hibernate has two ways to enable logging of SQL statements. These statements are most useful during the testing and debugging phases of application development.

The first way is to explicitly enable it in your code.

```
SessionFactory sf = new Configuration()
    .setProperty("hibernate.show_sql", "true")
    // ...
    .buildSessionFactory();
```

Alternately, you can configure Hibernate to send all SQL messages to **log4j**, using a specific facility:

```
log4j.logger.org.hibernate.SQL=DEBUG, SQL_APPENDER
log4j.additivity.org.hibernate.SQL=false
```

The **additivity** option controls whether these log messages are propagated upward to parent handlers, and is a matter of preference.

4.2.2. Transaction Service Logging

The TransactionManagerService included with the Enterprise Platform handles logging differently than the stand-alone Transaction Service. Specifically, it overrides the value of the `com.arjuna.common.util.logger` property given in the **jbossjta-properties.xml** file, forcing use of the **log4j_releveller** logger. All **INFO** level messages in the transaction code behave as **DEBUG** messages. Therefore, these messages are only present in log files if the filter level is **DEBUG**. All other log messages behave as normal.

Chapter 5. Deployment

Deploying applications on JBoss Enterprise Web Platform is achieved by copy the application into the **\$JBOSS_HOME/server/\$PROFILE/deploy** directory, where **\$PROFILE** is the server profile you wish to use. The JBoss Enterprise Web Platform constantly scans the **deploy** directory to pick up new applications or any changes to existing applications. This enables the *hot deployment* of applications on the fly, while JBoss Enterprise Web Platform is still running.

5.1. Deployable Application Types

JBoss Enterprise Web Platform uses mutiple deployers to transform the metadata associated with a deployment until the deployment is processed by a deployer that creates a runtime component from the metadata. Deployments must contain a descriptor that cause component metadata to be added to the deployment. Deployers for the following deployment types are available in JBoss Enterprise Web Platform by default:

WAR

The WAR application archive (e.g., myapp.war) packages Java EE web applications in a JAR file. It contains servlet classes, view pages, libraries, and deployment descriptors in WEB-INF such as **web.xml**, **faces-config.xml**, and **jboss-web.xml** etc..

EAR

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 META-INF deployment descriptors such as application.xml and jboss-app.xml etc.



Persistence Units in EAR Exploded Deployment

According to EJB3 specification, deployment of a persistence unit into an EAR should fail when the unit is outside of the EAR file and the bean attempting to inject the persistence unit is within the EAR. To follow the specification, you need to deploy the persistence unit packaged within the EAR file.

However, JBoss EAP persistence units can exist outside of their EARs. To allow this behavior, modify the bean class of the **PersistenceUnitDependencyResolver** bean in the file **deployers/ejb3.deployer/META-INF/jpa-deployer-jboss-beans.xml** under the respective JBoss AS server profile:

```
<!--
Can be DefaultPersistenceUnitDependencyResolver for spec compliant
resolving,
InterApplicationPersistenceUnitDependencyResolver for resolving
beyond EARs,
or DynamicPersistencePersistenceUnitDependencyResolver which allows
configuration via JMX.
-->
<bean name="PersistenceUnitDependencyResolver"
class="org.jboss.jpa.resolvers.DynamicPersistenceUnitDependencyResolve
r"/>
```

The bean default value is **DynamicPersistenceUnitDependencyResolver**. This resolver allows you to specify the specification-compliant behavior, which can be additionally monitored through an MBean in the JMX Console. To use the spec-noncompliant JBoss variant, set the bean to **InterApplicationPersistenceUnitDependencyResolver**.

JBoss Microcontainer

The JBoss Microcontainer (MC) beans archive (typical suffixes include, `.beans`, `.deployer`) packages a POJO deployment in a JAR file with a **META-INF/jboss-beans.xml** descriptor. This format is commonly used by the JBoss Enterprise Web Platform component deployers.

You can deploy ***-jboss-beans.xml** files with MC beans definitions. If you have the appropriate JAR files available in the `deploy` or `lib` directories, the MC beans can be deployed using such a standalone XML file.

SAR

The SAR application archive (e.g., `myservice.sar`) packages a JBoss service in a JAR file. It is mostly used by JBoss Enterprise Web Platform internal services that have not been updated to support MC beans style deployments.

You can deploy ***-service.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 you deploy many JBoss Enterprise Web Platform internal services that have not been updated to support POJO style deployment, such as the JMS queues.

DataSource

The ***-ds.xml** file defines connections to external databases. The data source can then be reused by all applications and services in JBoss Enterprise Web Platform via the internal JNDI.

*AR

You can also deploy JAR files containing EJBs or other service objects directly in JBoss Enterprise Web Platform. The list of suffixes that are recognized as JAR files is specified in the **conf/bootstrap/deployers.xml** JARStructure bean constructor set.

5.1.1. Exploded Deployment

The WAR, EAR, MC beans and SAR deployment packages are JAR files with special XML deployment descriptors in directories like `META-INF` and `WEB-INF`. JBoss Enterprise Application Platform allows you to deploy the archives also as expanded directories instead of JAR files. This is called exploded deployment and allows you to make application changes on the fly, that is without re-deploying the entire application. If you need to re-deploy an exploded directory without restart the server, just **touch** the deployment descriptors (that is the **WEB-INF/web.xml** in a WAR and the **META-INF/application.xml** in an EAR) to update their timestamps.

5.2. Standard Server Profiles

The JBoss Enterprise Web Platform ships with two server profiles. You can choose which configuration to start by passing the **-c** parameter to the server startup script. For instance, the **run.sh -c production** command would start the server in the **production** profile. Each profile is contained in a directory named **JBOSS_HOME/server/\$PROFILE/**, where **\$PROFILE** represents the profile name. You can look into each server profile's directory to see the services, applications, and libraries included in the profile.



Note

The exact contents of the **server/\$PROFILE** directory depends on the profile service implementation and is subject to change as the management layer and embedded server evolve.

- ▶ The **default** profile is the mostly common used profile for application developers. It supports the standard Java EE 5.0 programming APIs (for example, Annotations, JPA, and EJB3).
- ▶ The **production** profile has the features of the **default** profile with added clustering support and enterprise extensions. It is optimized for production environments.

The detailed services and APIs supported in each of these profiles will be discussed throughout.

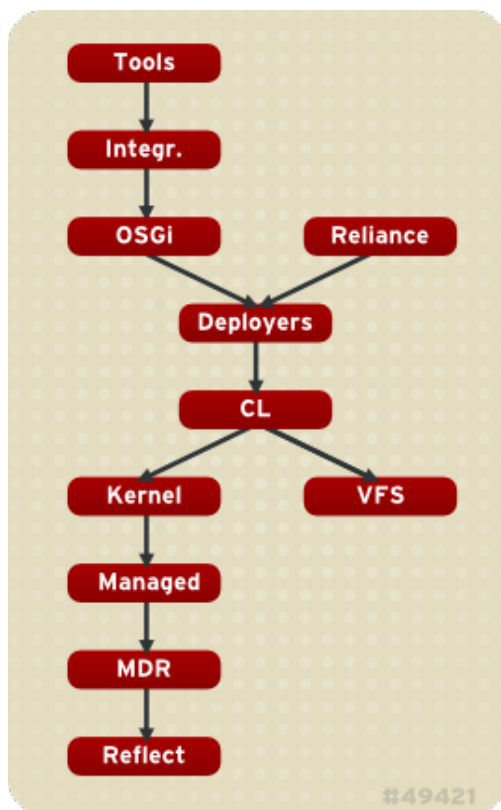
Chapter 6. Microcontainer

JBoss Enterprise Web Platform 5.0 uses JBoss Microcontainer to integrate enterprise services together with a Servlet/JSP container, EJB container, deployers and management utilities in order to provide a standard Java EE environment. If you need additional services then you can simply deploy these on top of Java EE to provide the functionality you need. Likewise any services that you do not need can be removed by changing the configuration.

JBoss Microcontainer is very lightweight and deals with POJOs. As with other lightweight containers, JBoss Microcontainer uses dependency injection to wire individual POJOs together to create services. Configuration is performed using either annotations or XML depending on where the information is best located. Unit testing is made extremely simple thanks to a helper class that extends JUnit to setup the test environment, allowing you to access POJOs and services from your test methods using just a few lines of code.

6.1. An overview of the Microcontainer modules

This section introduces the various Microcontainer modules. The figure below gives an overview of the modules.



- **Tools** represents a variety of tools to assist users with use and development with JBoss Microcontainer.
- **Integr.** represents the **aop-mc-int** module, which handles integration between the JBoss AOP and JBoss Microcontainer.
- **OSGi** represents several integration classes that adapt the OSGi module for the Microcontainer.
- **Reliance** represents two modules: **Drools-int** and **jBPM-int**. These modules define JBoss Rules dependencies.
- **Deployers** load components from various modules (such as POJOs, JMX, Spring, Java EE) into the Microcontainer at runtime.
- **CL** represents the *ClassLoader*, a new peer classloader module that handles the OSGi bundle module.

- ▶ **VFS** represents the *Virtual File System*. This is an abstract layer used to identify known file system issues within a single module.
- ▶ **Kernel** defines the core kernel SPI, including bootstrap, configuration, POJO deployments, dependency, events, bean metadata and bean registry. It contains the following modules:
 - **Dependency**
 - **Kernel**
 - **AOP-MC-int**
 - **Spring-int**
 - **Guice-int**
- ▶ **Managed** represents two modules: **managed** and **metatype**. These modules define the base objects that define the management view of a component.
- ▶ **MDR** is the generic *Metadata Repository*. It handles scoped metadata lookups.
- ▶ **Reflect** is the integration point for manipulating class information at runtime.

6.2. Configuration

To configure the Microcontainer bootstrap you can use the `$JBOSS_HOME/server/$PROFILE/conf/bootstrap.xml` and `$JBOSS_HOME/server/$PROFILE/conf/bootstrap/*.xml` files where `$PROFILE` represents the name of the server profile; either **production** or **default**. The `bootstrap.xml` simply references Microcontainer deployment descriptors that should be loaded in the indicated order. The current **default** profile `bootstrap.xml` references are:

`logging.xml`

Logging manager and bridge configuration.

`vfs.xml`

JBoss VFS caching beans.

`classloader.xml`

The root class loading beans for the peer class loading model.

`aop.xml`

JBoss AOP integration and AspectManager beans.

`jmx.xml`

JBoss JMX kernel initialization.

`deployers.xml`

Core deployers for `-jboss-beans.xml` and `-service.xml`.

`profile.xml`

Full featured repository based profile service referenced by `bootstrap.xml`

The main beans are:

ProfileService

Loads the deployments associated with the specified server profile. Constantly examines

server/\$PROFILE/conf/jboss-service.xml, **server/\$PROFILE/deployers** and **server/\$PROFILE/deploy** for deployments.

AspectManager

AOP aspects.

MainDeployer

Deployer aspects are registered with the **MainDeployer** as an ordered list via an injection of the **deployers** property.

ServiceClassLoaderDeployer

Manages the class loading aspect of deployment.

JARDeployer

A structural deployment aspect which handles the legacy nested deployment behavior of adding non-deployable JARs to the current deployment classpath.

FileStructure

A structural deployment aspect which recognizes common deployment file types by suffix, as specified by **FileManager**.

AspectDeployer

Handles AOP descriptor deployments.

BeanDeployer

Translates * -**jboss-beans.xml** into **KernelDeployment** for the descriptor beans.

KernelDeploymentDeployer

Translates a **KernelDeployment** into the constituent **BeanMetaData** instances for the kernel beans.

SARDeployer

Handles legacy * -**service.xml** MBean descriptors and maps them as **ServiceDeployment** POJOs.

ServiceDeploymentDeployer

Translates the **ServiceDeployment** POJO into the constituent **ServiceMetaData** that represent the various MBeans.

ServiceDeployer

Creates the MBean services from deployment **ServiceMetaData** instances.

JMXKernel

Manages JMX kernel and **MBeanServer** instantiation in the JBoss domain. It is used by **SARDeployer** and will be used to expose kernel beans via JMX in the future.

HDSscanner

Queries the profile service for changes to the **deploy** directory contents. It then redeploys updated content, undeploys removed content, and adds new deployment content to the profile service.

6.3. References

More information on the JBoss Microcontainer project can be obtained from <http://www.jboss.org/jbossmc/>.

Chapter 7. 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 Enterprise Application Platform instance, needs only know that it needs to talk to the a message queue named **queue/IncomingOrders** and need not worry about any of the queue's configuration details.

In a clustered environment, naming services are even more valuable. A client of a service must be able to look up a **ProductCatalog** session bean from the cluster without needing to know which machine it resides on. Whether it is a large clustered service, a local resource or an application component that is needed, the JNDI naming service provides the glue that lets code find the objects in the system by name.

7.1. An Overview of JNDI

JNDI is a standard Java API that is bundled with the Java Development Kit. 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.

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**.

7.1.1. 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 Enterprise Application Platform 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.

7.1.2. 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.

7.1.2.1. 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 **jndi.properties** resource files found on the classpath.

For each property found in both of these two sources, the property's value is determined as follows. If the property is one of the standard JNDI properties that specify a list of JNDI factories, all of the values are concatenated into a single colon-separated list. For other properties, only the first value found is used. The preferred method of specifying the JNDI environment properties is through a **jndi.properties** file, 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 7.1, “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 7.1. 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
```

7.2. The JBoss Naming Service Architecture

The JBoss Naming Service (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 7.1, “Key components in the JBoss Naming Service architecture.”](#) illustrates some of the key classes in the JBossNS implementation and their relationships.

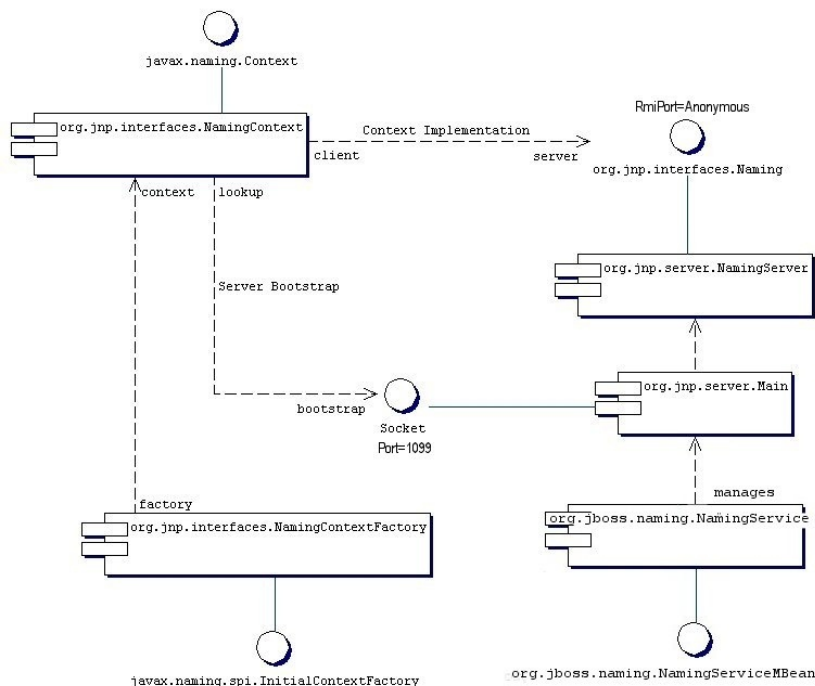


Figure 7.1. Key components in the JBoss Naming Service architecture.

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 JBoss Naming Service **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 JBoss Naming Service 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 JBoss Naming Service **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.

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.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.

7.3. The Naming InitialContext Factories

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

7.3.1. 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**.

7.3.2. 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.

7.3.3. 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.

7.3.4. 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 7.4.1, "Accessing JNDI over HTTP"](#).



Note

If a cluster partition uses the default partition name, the discovery process ignores other clusters. Therefore, make sure to specify unique partition names: `props.put("jnp.partitionName", "ClusterBPartition")` when using several clusters.

7.3.5. 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.

7.3.6. 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.

7.4. 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.

7.4.1. 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.

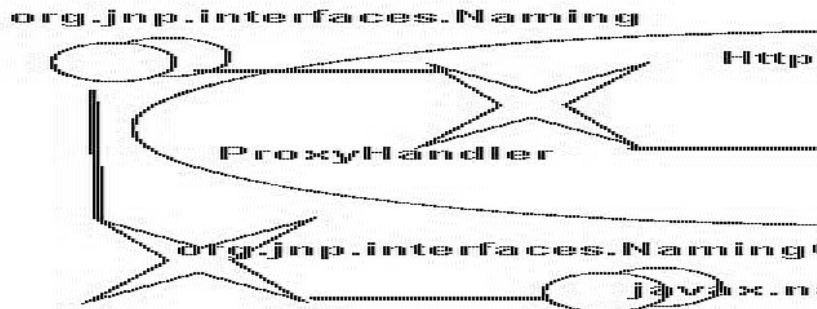


Figure 7.2. The HTTP invoker proxy/server structure for a JNDI Context

Before looking at the configurations let's look at the operation of the **http-invoker** services.

[Figure 7.2, "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 7.3, "The relationship between configuration files and JNDI/HTTP component"](#) illustrates the relationship between configuration files and the corresponding components.

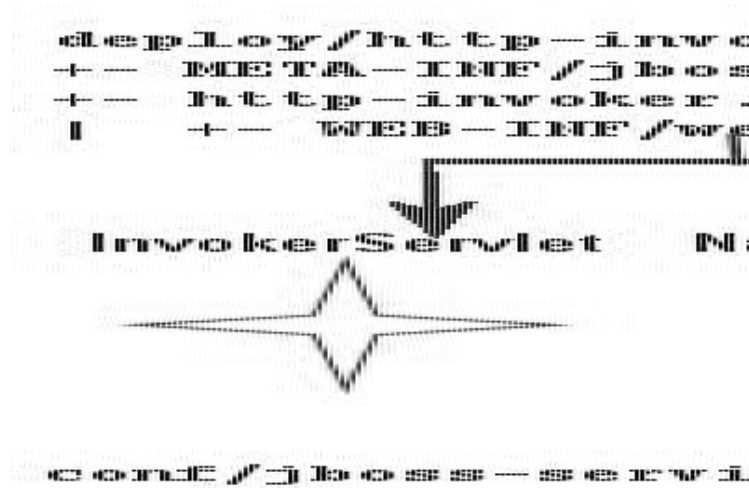


Figure 7.3. The relationship between configuration files and JNDI/HTTP component

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:

- ▶ **ObjectName**: 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.

7.4.2. 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.sar/jboss-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 7.2, "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 programatically. 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 7.2. 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

```

7.4.3. 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 7.3, "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 7.3. 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 sevlets are secured, and which roles are allowed to access

the secured servlets. You must additionally define the security domain that will handle the authentication and authorization for the war. This is done through the **jboss-web.xml** descriptor, and an example that uses the **http-invoker** security domain is given 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.

7.4.4. 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. The rest of this section explains how read-only works in JBoss Enterprise Application Platform.

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</attribute>
  <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</interceptor>
      <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/invoke/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.

7.5. 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**.

7.5.1. 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"b">
    <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>
```

7.5.2. 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>
```

7.5.3. **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");
```

7.5.4. 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 7.4, "The JMX Console view of the configured JBoss MBeans"](#).



Figure 7.4. 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 7.5, "The JMX Console view of the JNDIView MBean"](#).

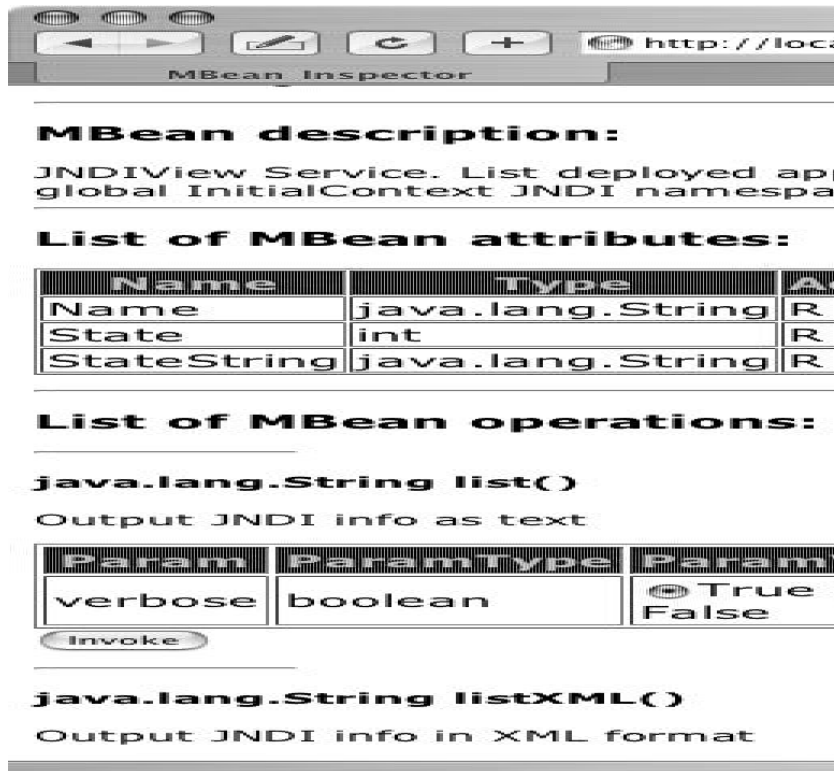


Figure 7.5. The JMX Console view of the JNDIView MBean

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 7.6, “The JMX Console view of the JNDIView list operation output”](#).

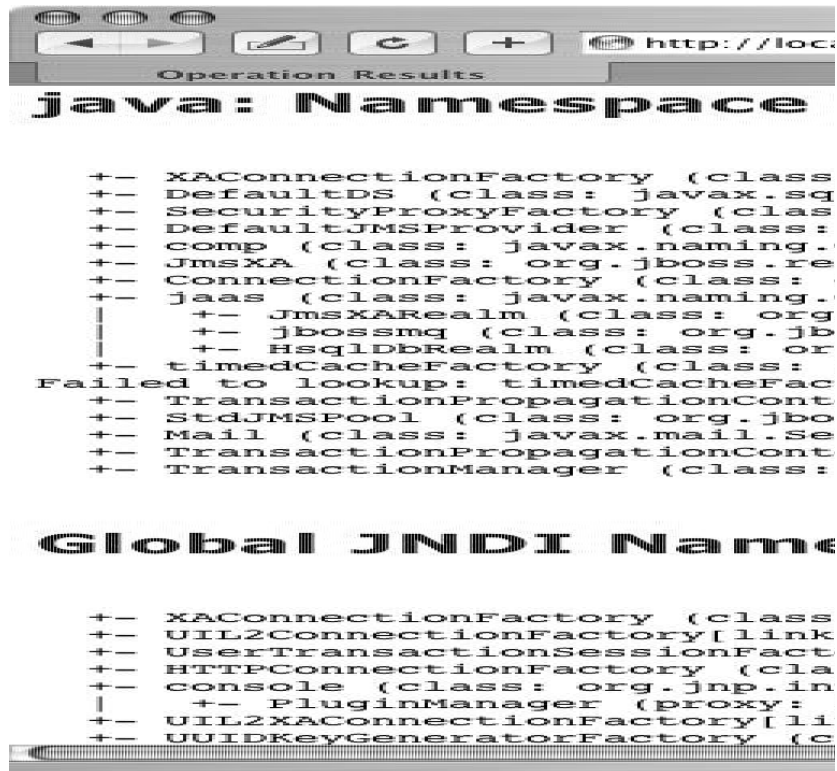


Figure 7.6. The JMX Console view of the JNDIView list operation output

7.6. 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.

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 7.4, "ENC access sample code"](#) illustrates the prototypical lines of code a component uses to access its ENC.

Example 7.4. 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 7.2, "The JBoss Naming Service 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.

7.6.1. 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 deployment descriptor element, there is a JBoss server specific deployment descriptor element that maps the JNDI name as used by the application component to the deployment environment JNDI name.

7.6.1.1. 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 7.5, “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 7.6, “ENC env-entry access code fragment”](#) shows a sample code fragment for accessing the **maxExemptions** and **taxRate** and **env-entry** values declared in the deployment descriptor.

Example 7.5. 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 7.6. 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");

```

7.6.1.2. 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 7.7, “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 7.7, “An example ejb-jar.xml ejb-ref descriptor fragment”](#) is given in [Example 7.8, “ENC ejb-ref access code fragment”](#).

Example 7.7. 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 7.8. 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");

```

7.6.1.3. 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 7.7, “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 7.9, “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 7.9. 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>
<!-- ... -->
```

7.6.1.4. 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 7.10, “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 7.10, “An example ejb-jar.xml ejb-local-ref descriptor fragment”](#) is given in [Example 7.11, “ENC ejb-local-ref access code fragment”](#).

Example 7.10. 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 7.11. 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");
```

7.6.1.5. 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 7.12, "A web.xml resource-ref descriptor fragment"](#) shows an example **web.xml** descriptor fragment that illustrates the **resource-ref** element usage. [Example 7.13, "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 7.12. 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 7.13. 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");

```

7.6.1.6. 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 7.14, “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 7.12, “A web.xml resource-ref descriptor fragment”](#).

Example 7.14. 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>

```

7.6.1.7. 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 7.15, “An example ejb-jar.xml resource-env-ref fragment”](#) provides an example **resource-ref-env** element declaration by a session bean. [Example 7.16, “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 7.15. 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 7.16. 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");

```

7.6.1.8. 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 7.17, "A sample jboss.xml resource-env-ref descriptor fragment"](#) provides a sample **jboss.xml** descriptor fragment that shows a sample mapping for the **StockInfo** **resource-env-ref**.

Example 7.17. 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 8. Web Services

Web services are a key contributing factor in the way Web commerce is conducted today. Web services enable applications to communicate by sending small and large chunks of data to each other.

A web service is essentially a software application that supports interaction of applications over a computer network or the world wide web. Web services usually interact through XML documents that map to an object, computer program, business process or database. To communicate, an application sends a message in XML document format to a web service which sends this message to the respective programs. Responses may be received based on requirements, the web service receives and then sends them in XML document format to the required program or applications. Web services can be used in many ways, examples include supply chain information management and business integration.

JBossWS is a web service framework included as part of the JBoss Enterprise Web Platform. It implements the JAX-WS specification that defines a programming model and run-time architecture for implementing web services in Java, targeted at the Java Platform, Enterprise Edition 5 (Java EE 5). Even though JAX-RPC is still supported (the web service specification for J2EE 1.4), JBossWS does put a clear focus on JAX-WS.



Warning

JAX-RPC is not supported for JBoss Web Services CXF Stack.

8.1. The need for web services

Enterprise systems communication may benefit from a wise adoption of web service technologies. Focusing attention on well designed contracts allows developers to establish an abstract view of their service capabilities. Considering the standardized way contracts are written, this definitely helps communication with third-party systems and eventually supports business-to-business integration; everything is clear and standardized in the contract the provider and consumer agree on. This also reduces the dependencies between implementations allowing other consumers to easily use the provided service without major changes.

Other benefits exist for enterprise systems that incorporate web service technologies for internal heterogenous subsystems communication as web service interoperability boosts service reuse and composition. Web services eliminates the need to rewrite whole functionalities because they were developed by another enterprise department using a different software language.

8.2. What web services are not

Web services are not the solution for every software system communication.

Nowadays they are meant to be used for loosely-coupled coarse-grained communication, message (document) exchange. Recent times has seen many specifications (WS-*) discussed and finally approved to establish standardized ws-related advanced aspects, including reliable messaging, message-level security and cross-service transactions. Web service specifications also include the notion of registries to collect service contract references, to easily discover service implementations.

This all means that the web services technology platform suits complex enterprise communication and is not simply the latest way of doing remote procedure calls.

8.3. 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. The payload of the SOAP message is an XML document that can be validated against XML schema. The 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>
```

8.4. Document/Literal (Bare)

Bare is an implementation detail from the Java domain. Neither in the abstract contract (for instance, 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;

    ...
}

```

8.5. Document/Literal (Wrapped)

Wrapped is an implementation detail from the Java domain. Neither in the abstract contract (for instance, 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

With JBossWS the request and response wrapper annotations are not required, they will be generated on demand using sensible defaults.

8.6. 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.

8.7. RPC/Encoded

SOAP encoding style is defined by [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



Note

This section should not be used in conjunction with JBoss Web Services CXF Stack.

8.8. 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 (for instance, wsdl+schema) for client consumption. All marshalling/unmarshalling is delegated to JAXB.

8.9. Plain old Java Object (POJO)

Let us take a look at simple POJO endpoint implementation. All endpoint associated metadata are provided via JSR-181 annotations

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}
```

8.10. 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>
```

8.11. 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** file are required.

8.12. 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
```

It is also possible to generate the abstract contract off line using jboss tools. For details of that see [Top Down \(Using wsconsume\)](#)

8.13. 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 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 will find the links to the generated WSDL.

```
http://yourhost:8080/jboss/ws/services
```

It is also possible to generate the abstract contract offline using JbossWS tools. For details of that please see [Top Down \(Using wsconsume\)](#)

8.14. 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
    }
}
```

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 (for example, with MESSAGE the Provider can also see SOAP Headers)

8.15. 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);
    }
}

```

8.16. Web Service Clients

8.16.1. 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).

8.16.1.1. Service Usage

Static case

Most clients will start with a WSDL file, and generate some stubs using jboss ws 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 not the recommended way to use JBossWS.

8.16.1.2. 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** and **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.

8.16.1.3. 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.

8.16.2. 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 <code>serviceEndpointInterface</code>
 * 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 \(Using wsconsume\)](#).

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-webserviceref?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);
    }
}

```

8.16.3. 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 or method declaration then 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 and 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

JBoss Enterprise Web Platform offers 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.webservicesref.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>

```

8.16.4. Dispatch

XML Web 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. For example, 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. For example, 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)));
```

8.16.5. 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);
}
```

8.16.6. Oneway Invocations

@Oneway indicates that the given web method has only an input message and no output. Typically, a one-way 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;
    }
}

```

8.17. Common API

This sections describes concepts that apply equally to [Web Service Endpoints](#) and [Web Service Clients](#)

8.17.1. 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.

8.17.1.1. 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`.

8.17.1.2. 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`.

8.17.1.3. 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)

8.17.1.4. 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!
```

8.17.2. 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 associated with 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. `APPLICATION` scoped properties are also made available to client applications and service endpoint implementations. The default scope for a property is `HANDLER`.

8.17.2.1. Accessing the message context

Users can access the message context in handlers or in endpoints via `@WebServiceContext` annotation.

8.17.2.2. Logical Message Context

`LogicalMessageContext` is passed to **Logical Handlers** at invocation time. `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.

8.17.2.3. SOAP Message Context

`SOAPMessageContext` is passed to **SOAP handlers** at invocation time. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.

8.17.3. 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

8.18. DataBinding

8.18.1. Using JAXB with non annotated classes

JAXB is heavily driven by Java Annotations on the Java Bindings. It currently doesn't support an external binding configuration.

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/jbossws/projects/jaxbintros/>

Complete documentation can be found here:

- [JAXB Introductions](#)

8.19. Attachments

JBoss-WS4EE relied on a deprecated attachments technology called SwA (SOAP with Attachments). SwA required soap/encoding which is disallowed by the WS-I Basic Profile. JBossWS provides support for WS-I AP 1.0, and MTOM instead.

8.19.1. 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 \(XOP\)](#)

8.19.1.1. 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

8.19.1.2. 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.

8.19.2. SwaRef

[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](#).

8.19.2.1. 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>
```

8.19.2.2. 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 wsi:swaRef schema type would then be mapped to DataHandler.

8.20. 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 without breaking 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
wsprovide	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
wsconsume	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development
wsrunclient	Executes a Java client (that has a main method) using the JBossWS classpath.

8.20.1. 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 vender 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 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.

To create a POJO endpoint for deployment on JBoss Enterprise Web Platform, create a simple **web.xml**:

```
<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 <replaceable>$JBOSS_HOME</replaceable>/server/default/deploy
```

At deploy time JBossWS 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.

8.20.2. 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;
    }
}
```

8.20.3. Client Side

Before going into 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 achieving 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 operating system, 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>
```

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 Web Services operation can then be called by just invoking a method on the returned interface.



Note

It is 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 `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]));
...
```

8.20.4. Command-line & Ant Task Reference

- [wsconsume reference page](#)
- [wsprovide reference page](#)

- [wsrunclient reference page](#)

8.20.5. 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](#)

8.21. Web Service Extensions

8.21.1. WS-Addressing

This section describes how [WS-Addressing](#) can be used to provide a stateful service endpoint.

8.21.1.1. Specifications

WS-Addressing is defined by a combination of the following specifications from the W3C Recommendation. The WS-Addressing API is standardized by [JSR-224 - Java API for XML-Based Web Services \(JAX-WS\)](#)

- [Web Services Addressing 1.0 - Core](#)
- [Web Services Addressing 1.0 - SOAP Binding](#)

8.21.1.2. Addressing Endpoint



Note

The following information should not be used in conjunction with JBoss Web Services CXF Stack.

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")
@Addressing(enabled=true, required=true)
@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 2.1 defined `javax.xml.ws.soap.Addressing` annotation to enable the server side addressing handler.

8.21.1.3. Addressing Client

The client code uses **javax.xml.ws.soap.AddressingFeature** feature from JAX-WS 2.1 API to enable the WS-Addressing.

```
Service service = Service.create(wsdlURL, serviceName);
port1 = (StatefulEndpoint)service.getPort(StatefulEndpoint.class, new
AddressingFeature());
```

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:Action>
<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:Action>
<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>

```

8.21.2. 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 (for example, point to point, or multi-hop relay).

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 \(Still in Draft\)](#)

8.21.2.1. 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 must 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 must setup both the endpoint configuration and the WSSE declarations. These are two separate steps.

8.21.2.2. 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 using X.509v3 certificate (a standard certificate). The alias option says that the certificate and 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. Users can use 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
{
    ...
}
```

8.21.2.3. 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.

8.21.2.3.1. 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 therefore can not be manipulated in transit.

```
<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>
```

8.21.2.4. Installing the BouncyCastle JCE provider

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 and JRE distribution). You will 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 users will put the provider jar is mostly up to them, although with jdk5 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 user think he have installed it correctly and it still doesn't work then with high probability the provider installation is not used.

8.21.2.5. Username Token Authentication

If you need to authenticate clients through a Username Token, the JAAS integration will verify the received token against the configured JBoss JAAS Security Domain.

Example 8.1. Basic Username Token Configuration

To implement this feature, you must append a `<jboss-ws-security>` element to `jboss-wsse-client.xml` that contains the following information.

```
<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">
  <config>
    (1)   <username/>
    (2)   <timestamp ttl="300"/>
  </config>
</jboss-ws-security>
```

Line (2) specifies that a `<timestamp>` element must be present in the message and that the message can not be older than 300 seconds. The seconds limitation is used to prevent replay attacks.

You must then specify the same `<timestamp>` element and ***seconds*** attribute in the `jboss-wsse-server.xml` file so both headers match. You must also specify the `<requires/>` element to enforce this condition.

```
<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">
  <config>
    <timestamp ttl="300"/>
    <requires/>
  </config>
</jboss-ws-security>
```



Warning

This example configuration results in simple text user information being sent in SOAP headers. You should strongly consider implementing JBossWS Secure Transport

Password Digest, Nonces, and Timestamp

[Example 8.1, “Basic Username Token Configuration”](#) results in the client password being sent as plain text. You can use a combination of *digested passwords*, *nonces*, and *timestamps* to provide further protection from replay attacks.

To enable password digesting, you must implement the following items as described in [Example 8.2, “Enable Password Digesting”](#):

Example 8.2. Enable Password Digesting

In the `<username>` element of the `jboss-wsse-client.xml` file:

- enable the *digestPassword* attribute
- enable the *nonces* and *timestamps* attributes.

```
<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">
  <config>
  (3)   <username digestPassword="true" useNonce="true" useCreated="true"/>
        <timestamp ttl="300"/>
  </config>
</jboss-ws-security>
```

In the `login-config.xml` file, you must also implement the `UsernameTokenCallback` module option.

Example 8.3. UsernameTokenCallback Module

```
<application-policy name="JBossWSDigest">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
      flag="required">
      <module-option name="usersProperties">META-INF/jbossws-
users.properties</module-option>
      <module-option name="rolesProperties">META-INF/jbossws-
roles.properties</module-option>
      <module-option name="hashAlgorithm">SHA</module-option>
      <module-option name="hashEncoding">BASE64</module-option>
      <module-option name="hashUserPassword">false</module-option>
      <module-option name="hashStorePassword">true</module-option>
      <module-option
name="storeDigestCallback">org.jboss.ws.extensions.security.auth.callback.Username
TokenCallback</module-option>
      <module-option name="unauthenticatedIdentity">anonymous</module-option>
    </login-module>
  </authentication>
</application-policy>
```

You may wish to use a more sophisticated custom login module to provide more security against replay attacks. You can use your own custom login module provided you implement the following:

- plug the **UsernameTokenCallback** callback into your login module
- extend the **org.jboss.security.auth.spi.UsernamePasswordLoginModule**
- set the hash attributes (*hashAlgorithm*, *hashEncoding*, *hashUserPassword*, *hashStorePassword*) as shown in [Example 8.3, "UsernameTokenCallback Module"](#).

Advanced Tuning - Nonce Factory

The way nonces are created, and subsequently checked and stored on the server side, influences overall security against replay attacks. Currently JBossWS ships with a basic implementation of a nonce store that does not cache the received tokens on the server side.

More complex implementation can be plugged into your modules by implementing the **NonceFactory** and **NonceStore** interfaces. You can find these interfaces in the `org.jboss.ws.extensions.security.nonce` package.

Once included, you specify your factory class through the `<nonce-factory-class>` element in the `jboss-wsse-server.xml` file.

Advanced Tuning - Timestamp Verification

If a Timestamp is present in the `wsse:Security` header, header verification does not allow for any tolerance whatsoever in the time comparisons. If the message appears to have been created even slightly in the future or if the message has just expired it will be rejected. A new element called `<timestamp-verification>` is available for the wsse configuration. [Example 8.4, "<timestamp-verification> Configuration"](#) describes the required attributes for the `<timestamp-verification>` element.

Example 8.4. `<timestamp-verification>` Configuration

The `<timestamp-verification>` element attributes allow you to specify the tolerance in seconds that is used when verifying the 'Created' or 'Expires' element of the 'Timestamp' header.

```
<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'>
  <timestamp-verification createdTolerance="5" warnCreated="false"
expiresTolerance="10" warnExpires="false" />
</jboss-ws-security>
```

createdTolerance

Number of seconds in the future a message will be accepted. The default value is **0**.

expiresTolerance

Number of seconds a message is rejected after being classed as expired. The default value is **0**.

warnCreated

Specifies whether to log a warning message if a message is accepted with a 'Created' value in the future. The default value is **true**.

warnExpires

Specifies whether to log a warning message if a message is accepted with an 'Expired' value in the past. The default value is **true**.



Note

The ***warnCreated*** and ***warnExpires*** attributes can be used to identify accepted messages that would normally be rejected. You can use this data to identify clients that are out of sync with the server time, without rejecting the client messages.

8.21.2.5.1. Secure Transport

8.21.2.6. X509 Certificate Token

By using X509v3 certificates, you can both sign and encrypt messages.

Encryption

To configure encryption, you must specify the items in [Example 8.5, “X509 Encryption Configuration”](#). The configuration is the same for clients and servers.

Example 8.5. X509 Encryption Configuration

```
<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/bob-sign_enc.jks</key-store-file>
      <key-store-password>password</key-store-password>
      <key-store-type>jks</key-store-type>
      <trust-store-file>WEB-INF/wsse10.truststore</trust-store-file>
      <trust-store-password>password</trust-store-password>

      <config>
        <timestamp ttl="300"/>
(2)    <sign type="x509v3" alias="1" includeTimestamp="true"/>
(3)    <encrypt type="x509v3"
          alias="alice"
          algorithm="aes-256"
          keyWrapAlgorithm="rsa_oaep"
          tokenReference="keyIdentifier" />
(4)    <requires>
          <signature/>
          <encryption/>
        </requires>
      </config>
</jboss-ws-security>
```

The server configuration includes the following encryption information:

1. Keystore and Truststore information: location of each store, the password, and type of store.
2. Signature configuration: you must provide the certificate and key pair aliases to use.
includeTimestamp specifies whether the timestamp is signed to prevent tampering.
3. Encryption configuration: you must provide the certificate and key pair aliases to use. Refer to [Algorithms](#) for more information.
4. Optional security requirements: incoming messages must be both signed, and encrypted.

Dynamic Encryption

When replying to multiple clients, a service provider must encrypt a message according to its destination using the correct public key. The JBossWS native implementation of WS-Security obtains the correct key to use from the signature received (and verified) in the incoming message.

Example 8.6. Dynamic Encryption Configuration

To configure dynamic encryption, do not specify any encryption alias on the server side (1), and declare that a signature is required (2).

```
<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">
  <key-store-file>WEB-INF/bob-sign_enc.jks</key-store-file>
  <key-store-password>password</key-store-password>
  <key-store-type>jks</key-store-type>
  <trust-store-file>WEB-INF/wsse10.truststore</trust-store-file>
  <trust-store-password>password</trust-store-password>

  <config>
    <timestamp ttl="300"/>
    <sign type="x509v3" alias="1" includeTimestamp="true"/>
  (1)   <encrypt type="x509v3"
        algorithm="aes-256"
        keyWrapAlgorithm="rsa_oaep"
        tokenReference="keyIdentifier" />
        <requires>
  (2)   <signature/>
        <encryption/>
        </requires>
    </config>
  </jboss-ws-security>
```

Algorithms

Asymmetric and symmetric encryption is performed whenever the <encrypt> element is declared. Message data are encrypted using a generated symmetric secured key. This key is written in the SOAP header after being encrypted (wrapped) with the receiver public key. You can set both the encryption and key wrap algorithms.

The supported encryption algorithms include:

- AES 128 (aes-128) (default)
- AES 192 (aes-192)
- AES 256 (aes-256)
- Triple DES (triple-des)

The supported key-wrap algorithms include:

- RSA v1.5 (rsa_15) (default)
- RSA OAEP (rsa_oaep)



Note

The [Unlimited Strength Java\(TM\) Cryptography Extension](#) installation might be required to run some strong algorithms (for example, aes-256). Your country may impose limitations on the allowed cryptographic strength in applications. It is your responsibility to select the encryption level suitable for your jurisdiction.

Encryption Token Reference

For interoperability reasons, you may need to configure the type of reference to encryption token to be used. For example, Microsoft Indigo does not support direct reference to local binary security tokens which are the default reference type used by JBossWS.

To configure this reference, you specify the ***tokenReference*** attribute in the <encrypt> element. The values for the ***tokenReference*** attribute are:

- **directReference** (default)
- **keyIdentifier** - specifies the token data by means of an X509 SubjectKeyIdentifier reference.
- **x509IssuerSerial** - uniquely identifies an end entity certificate by its X509 Issuer and Serial Number



Note

Complete information about X509 Token Profiles are available in the *WSS X501 Certificate Token Profile 1.0* document, which can be obtained from the [Oasis.org docs portal](http://Oasis.org/docs/portal).

Targets Configuration

JBossWS gives you precise control over elements that must be signed or encrypted. This allows you to encrypt important data only (such as credit card numbers) instead of other, security-trivial, information exchanged by the same service (email addresses, for example). To configure this, you must specify the Qualified Name (qname) of the SOAP elements to encrypt. The default behavior is to encrypt the whole SOAP body.

```
<encrypt type="x509v3" alias="alice">
  <targets>
    <target type="qname">{http://www.my-company.com/cc}CardNumber</target>
    <target type="qname">{http://www.my-company.com/cc}CardExpiration</target>
    <target type="qname" contentOnly="true">{http://www.my-
company.com/cc}CustomerData</target>
  </targets>
</encrypt>
```

Payload Carriage Returns

Signature verification errors can occur in signed message payloads that contain carriage returns (\r) due to the way the special character is parsed by XML parsers. To prevent this issue, you can choose to implement custom encoding before sending the payload. Users can either encrypt the message, or force JBossWS to perform canonical normalization of messages.

The org.jboss.ws.DOMContentCanonicalNormalization property can normalize the payload if set to **true** in the MessageContext. The property must be set just before the invocation on the client side and in the endpoint implementation.

8.21.2.7. JAAS Integration

The WS-Security implementation allows users to achieve J2EE declarative security through JAAS integration. The calling user's identity and credentials are derived from the wsse headers of the incoming message, according to the parameters provided in the server wsse configuration file. Authentication and authorization is subsequently achieved delegating to the JAAS login modules configured for the specified security domain.

Username Token

Username Token Profile provides a mean of specifying the caller's username and password. The wsse server configuration file can be used to have those information used when performing authentication and authorization through configured login module.

```
<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">
  <config>
    <username/>
    <authenticate>
      <usernameAuth/>
    </authenticate>
  </config>
</jboss-ws-security>
```



Note

Prior to JBossWS 3.0.2 Native the username token was always used to set principal and credential of the caller whenever specified. This means that for backward compatibility reasons, this behavior is obtained also when no `authenticate` tag at all is specified and the username token is used.

X.509 Certificate Token

In previous versions of JBossWS, the username token was always used to set the principal and credential of the caller whenever specified. This behavior is retained for backward compatibility reasons where no `<authenticate>` element is specified and the username token is used.

```
<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">
  <key-store-file>META-INF/bob-sign.jks</key-store-file>
  <key-store-password>password</key-store-password>
  <key-store-type>jks</key-store-type>
  <trust-store-file>META-INF/wsse10.truststore</trust-store-file>
  <trust-store-password>password</trust-store-password>
  <config>
    <sign type="x509v3" alias="1" includeTimestamp="false"/>
    <requires>
      <signature/>
    </requires>
    <authenticate>
      (1) <signatureCertAuth
certificatePrincipal="org.jboss.security.auth.certs.SubjectCNMapping"/>
    </authenticate>
  </config>
</jboss-ws-security>
```

The optional **certificatePrincipal** attribute (1) specifies the class used to retrieve the principal from the X.509 certificate's attributes. The selected class must extend **CertificatePrincipal**. The default class used when no attribute is specified is **org.jboss.security.auth.certs.SubjectDNMapping**.

The configured security domain must have a correctly configured **BaseCertLoginModule**, as described in [Example 8.7, "BaseCertLoginModule Security Domain"](#).

Example 8.7. BaseCertLoginModule Security Domain

The following code sample shows a security domain with a **CertRolesLoginModule** that also enables authorization (using the specified **jbossws-roles.properties** file).

```
<application-policy name="JBossWSCert">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.CertRolesLoginModule"
      flag="required">
      <module-option name="rolesProperties">jbossws-roles.properties</module-
option>
      <module-option name="unauthenticatedIdentity">anonymous</module-option>
      <module-option name="securityDomain">java:/jaas/JBossWSCert</module-
option>
    </login-module>
  </authentication>
</application-policy>
```

The BaseCertLoginModule uses a central keystore to authenticate users. This store is configured through the **org.jboss.security.plugins.JaasSecurityDomain** MBean as shown in [Example 8.8, "BaseCertLoginModule Keystore"](#).

Example 8.8. BaseCertLoginModule Keystore

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
  name="jboss.security:service=SecurityDomain">
  <constructor>
    <arg type="java.lang.String" value="JBossWSCert"/>
  </constructor>
  <attribute name="KeyStoreURL">resource:META-INF/keystore.jks</attribute>
  <attribute name="KeyStorePass">password</attribute>
  <depends>jboss.security:service=JaasSecurityManager</depends>
</mbean>
```

At authentication time, the specified **CertificatePrincipal** mapping class accesses the keystore using the principal obtained from the associated wsse header. If a certificate is found and is the same as the one specified in the wsse header, the user is successfully authenticated.

8.21.2.8. POJO Endpoint Authentication and Authorization

The credentials obtained by WS-Security are generally 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.

**Important**

Authentication and Authorization should not be enabled for EJB based endpoints because the EJB container handles the security requirements of the deployed bean.

Procedure 8.1. Enabling POJO Authentication and Authorization

This procedure describes the additional configuration required to enable authentication and authorization for POJO endpoints.

1. Define Security Domain in Web Archive

You must define a security domain in the WAR containing the POJO.

Specify 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>
```

2. Configure the jboss-wsse-server.xml `<authorize>` element

Specify an `<authorize>` element within the `<config>` element.

The `<config>` element can be defined globally, be port-specific, or operation-specific.

The `<authorize>` element must contain either the `<unchecked/>` element or one or more `<role>` elements. Each `<role>` element must contain the name of a valid RoleName.

You can choose to implement two types of authentication: unchecked, and role-based authentication.

Unchecked Authentication

The authentication step is performed to validate the user's username and password, but no further role checking takes place. If the user's username and password are invalid, the request is rejected.

Example 8.9. Unchecked Authentication

```
<jboss-ws-security>

  <config>
    <authorize>
      <unchecked/>
    </authorize>
  </config>

</jboss-ws-security>
```

Role-based Authentication

The user is authenticated using their username and password as per Unchecked Authentication. Once the user's username and password is verified, user credentials are checked again to ensure at least one of the roles specified in the `<role>` element is assigned to the user.



Note

Authentication and authorization proceeds even if no username and password, or certificate was provided in the request message. In this scenario, authentication may proceed if the security domain's login module has been configured with an anonymous identity.

Example 8.10. Role-based Authentication

```
<jboss-ws-security>
  <config>
    <authorize>
      <role>friend</role>
      <role>family</role>
    </authorize>
  </config>
</jboss-ws-security>
```

8.21.3. XML Registries

J2EE 5.0 mandates support for Java API for XML Registries (JAXR). Inclusion of a XML Registry with the J2EE 5.0 certified Application Server is optional. JBoss EAP ships a UDDI v2.0 compliant registry, the Apache jUDDI registry. JAXR Capability Level 0 (UDDI Registries) is also supported through Apache Scout integration.

[Section 8.21.3. “XML Registries”](#) 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.

8.21.3.1. Apache jUDDI Configuration

jUDDI registry configuration happens via a 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');
```

8.21.3.2. 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
scout.proxy.transportClass=org.jboss.jaxr.scout.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** or **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);
```

8.21.3.3. 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 5.0 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 5.0 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. 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 5.0 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("scout.proxy.transportClass",
"org.jboss.jaxr.scout.transport.SaajTransport");
System.setProperty("scout.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 = bqmf.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.

8.21.3.4. 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](#)

8.21.3.5. Resources

- [JAXR Tutorial and Code Camps](#)
- [J2EE 1.4 Tutorial](#)
- [J2EE Web Services by Richard Monson-Haefel](#)

8.22. JBossWS Extensions

This section describes proprietary JBoss extensions to JAX-WS.

8.22.1. Proprietary Annotations

For the set of standard annotations, please have a look at [JAX-WS Annotations](#).

8.22.1.1. EndpointConfig

```
/**
 * Defines an endpoint or client configuration.
 * This annotation is valid on an endpoint implementaion bean or a SEI.
 */
@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 "";
}
```

8.22.1.2. 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;
}

```

8.22.1.3. SecurityDomain

```

/**
 * Annotation for specifying the JBoss security domain for an EJB
 */
@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 "";
}

```

8.23. Web Services Appendix



Note

This information can be used with JBoss Web Services CXF Stack.

[JAX-WS Endpoint Configuration](#)

[JAX-WS Client Configuration](#)

[JAX-WS Annotations](#)

8.24. References

1. [JSR-224 - Java API for XML-Based Web Services \(JAX-WS\) 2.0](#)
2. [JSR 222 - Java Architecture for XML Binding \(JAXB\) 2.0](#)
3. [JSR-250 - Common Annotations for the Java Platform](#)
4. [JSR 181 - Web Services Metadata for the Java Platform](#)

Chapter 9. JBoss AOP

JBoss AOP is a 100% Pure Java Aspect Oriented Framework usable in any programming environment or tightly integrated with our application server. Aspects allow you to more easily modularize your code base when regular object oriented programming just doesn't fit the bill. It can provide a cleaner separation from application logic and system code. It provides a great way to expose integration points into your software. Combined with JDK 1.5 Annotations, it also is a great way to expand the Java language in a clean pluggable way rather than using annotations solely for code generation.

JBoss AOP is not only a framework, but also a prepackaged set of aspects that are applied via annotations, pointcut expressions, or dynamically at runtime. Some of these include caching, asynchronous communication, transactions, security, remoting, and many many more.

An aspect is a common feature that is typically scattered across methods, classes, object hierarchies, or even entire object models. It is behavior that looks and smells like it should have structure, but you can't find a way to express this structure in code with traditional object-oriented techniques.

For example, metrics is one common aspect. To generate useful logs from your application, you have to (often liberally) sprinkle informative messages throughout your code. However, metrics is something that your class or object model really shouldn't be concerned about. After all, metrics is irrelevant to your actual application: it doesn't represent a customer or an account, and it doesn't realize a business rule. It's simply orthogonal.

9.1. Some key terms

Joinpoint

A *joinpoint* is any point in your Java program. The call of a method, the execution of a constructor, the access of a field; all these are joinpoints. You could also think of a joinpoint as a particular Java event, where an event is a method call, constructor call, field access, etc.

Invocation

An *invocation* is a JBoss AOP class that encapsulates what a joinpoint is at runtime. It could contain information like which method is being called, the arguments of the method, etc.

Advice

An *advice* is a method that is called when a particular joinpoint is executed, such as the behavior that is triggered when a method is called. It could also be thought of as the code that performs the interception. Another analogy is that an advice is an "event handler".

Pointcut

Pointcuts are AOP's expression language. Just as a regular expression matches strings, a pointcut expression matches a particular joinpoint.

Introduction

An *introduction* modifies the type and structure of a Java class. It can be used to force an existing class to implement an interface or to add an annotation to anything.

Aspect

An *aspect* is a plain Java class that encapsulates any number of advices, pointcut definitions, mixins, or any other JBoss AOP construct.

Interceptor

An *interceptor* is an aspect with only one advice, named **invoke**. It is a specific interface that you can

implement if you want your code to be checked by forcing your class to implement an interface. It also will be portable and can be reused in other JBoss environments like EJBs and JMX MBeans.

In AOP, a feature like metrics is called a *crosscutting concern*, as it is a behavior that "cuts" across multiple points in your object models, yet is distinctly different. As a development methodology, AOP recommends that you abstract and encapsulate crosscutting concerns.

For example, let's say you wanted to add code to an application to measure the amount of time it would take to invoke a particular method. In plain Java, the code would look something like the following.

```
public class BankAccountDAO
{
    public void withdraw(double amount)
    {
        long startTime = System.currentTimeMillis();
        try
        {
            // Actual method body...
        }
        finally
        {
            long endTime = System.currentTimeMillis() - startTime;
            System.out.println("withdraw took: " + endTime);
        }
    }
}
```

While this code works, there are a few problems with this approach:

1. It's extremely difficult to turn metrics on and off, as you have to manually add the code in the **try/finally** blocks to each and every method or constructor you want to benchmark.
2. Profiling code should not be combined with your application code. It makes your code more verbose and difficult to read, since the timings must be enclosed within the **try/finally** blocks.
3. If you wanted to expand this functionality to include a method or failure count, or even to register these statistics to a more sophisticated reporting mechanism, you'd have to modify a lot of different files (again).

This approach to metrics is very difficult to maintain, expand, and extend, because it is dispersed throughout your entire code base. In many cases, OOP may not always be the best way to add metrics to a class.

Aspect-oriented programming gives you a way to encapsulate this type of behavior functionality. It allows you to add behavior such as metrics "around" your code. For example, AOP provides you with programmatic control to specify that you want calls to **BankAccountDAO** to go through a metrics aspect before executing the actual body of that code.

9.2. Creating Aspects in JBoss AOP

In short, all AOP frameworks define two things: a way to implement crosscutting concerns, and a programmatic construct — a programming language or a set of tags to specify how you want to apply those snippets of code. Let's take a look at how JBoss AOP, its cross-cutting concerns, and how you can implement a metrics aspect in JBoss Enterprise Web Platform.

The first step in creating a metrics aspect in JBoss AOP is to encapsulate the metrics feature in its own Java class. The following code extracts the **try/finally** block in our first code example's **BankAccountDAO.withdraw()** method into **Metrics**, an implementation of a JBoss AOP Interceptor class.

The following example code demonstrates implementing metrics in a JBoss AOP Interceptor

```

01. public class Metrics implements org.jboss.aop.advice.Interceptor
02. {
03.     public Object invoke(Invocation invocation) throws Throwable
04.     {
05.         long startTime = System.currentTimeMillis();
06.         try
07.         {
08.             return invocation.invokeNext();
09.         }
10.         finally
11.         {
12.             long endTime = System.currentTimeMillis() - startTime;
13.             java.lang.reflect.Method m = ((MethodInvocation)invocation).method;
14.             System.out.println("method " + m.toString() + " time: " + endTime +
"ms");
15.         }
16.     }
17. }

```

Under JBoss AOP, the **Metrics** class wraps **withdraw()**: when calling code invokes **withdraw()**, the AOP framework breaks the method call into its parts and encapsulates those parts into an **Invocation** object. The framework then calls any aspects that sit between the calling code and the actual method body.

When the AOP framework is done dissecting the method call, it calls **Metrics**'s **invoke** method at line 3. Line 8 wraps and delegates to the actual method and uses an enclosing **try/finally** block to perform the timings. Line 13 obtains contextual information about the method call from the **Invocation** object, while line 14 displays the method name and the calculated metrics.

Having the **Metrics** code within its own object allows us to easily expand and capture additional measurements later on. Now that metrics are encapsulated into an aspect, let's see how to apply it.

9.3. Applying Aspects in JBoss AOP

To apply an aspect, you define when to execute the aspect code. Those points in execution are called *pointcuts*. An analogy to a pointcut is a regular expression. Where a regular expression matches strings, a pointcut expression matches events or *points* within your application. For example, a valid pointcut definition would be, "for all calls to the JDBC method **executeQuery()**, call the aspect that verifies SQL syntax."

An entry point could be a field access, or a method or constructor call. An event could be an exception being thrown. Some AOP implementations use languages akin to queries to specify pointcuts. Others use tags. JBoss AOP uses both.

The following listing demonstrates defining a pointcut for the **Metrics** example in JBoss AOP:

```

1. <bind pointcut="public void com.mc.BankAccountDAO->withdraw(double amount)">
2.     <interceptor class="com.mc.Metrics"/>
3. </bind >

4. <bind pointcut="* com.mc.billing.*->*(..)">
5.     <interceptor class="com.mc.Metrics"/>
6. </bind >

```

Lines 1-3 define a pointcut that applies the **metrics** aspect to the specific method **BankAccountDAO.withdraw()**. Lines 4-6 define a general pointcut that applies the **metrics** aspect to all methods in all classes in the **com.mc.billing** package. There is also an optional annotation mapping if you prefer to avoid XML. For more information, see the JBoss AOP reference documentation.

JBoss AOP has a rich set of pointcut expressions that you can use to define various points or events in

your Java application. Once your points are defined, you can apply aspects to them. You can attach your aspects to a specific Java class in your application or you can use more complex compositional pointcuts to specify a wide range of classes within one expression.

With AOP, as this example shows, you can combine all crosscutting behavior into one object and apply it easily and simply, without complicating your code with features unrelated to business logic. Instead, common crosscutting concerns can be maintained and extended in one place.

Note that code within the **BankAccountDAO** class does not detect that it is being profiled. Profiling is part of what aspect-oriented programmers deem orthogonal concerns. In the object-oriented programming code snippet at the beginning of this chapter, profiling was part of the application code. AOP allows you to remove that code. A modern promise of middleware is transparency, and AOP clearly delivers.

Orthogonal behavior can also be included after development. In object-oriented code, monitoring and profiling must be added at development time. With AOP, a developer or an administrator can easily add monitoring and metrics as needed without touching the code. This is a very subtle but significant part of AOP, as this separation allows aspects to be layered on top of or below the code that they cut across. A layered design allows features to be added or removed at will. For instance, perhaps you snap on metrics only when you're doing some benchmarks, but remove it for production. With AOP, this can be done without editing, recompiling, or repackaging the code.

9.4. Packaging AOP Applications

To deploy an AOP application in JBoss Enterprise Web Platform you need to package it. AOP is packaged similarly to SARs (MBeans). You can either deploy an XML file directly in the **deploy/** directory with the signature ***-aop.xml** along with your package (this is how the **base-aop.xml**, included in the **jboss-aop.deployer** file works) or you can include it in the JAR file containing your classes. If you include your XML file in your JAR, it must have the file extension **.aop** and a **jboss-aop.xml** file must be contained in a **META-INF** directory, for instance: **META-INF/jboss-aop.xml**.

In the JBoss Enterprise Web Platform 5, you *must* specify the schema used, otherwise your information will not be parsed correctly. You do this by adding the **xmlns="urn:jboss:aop-beans:1.0"** attribute to the root **aop** element, as shown here:

```
<aop xmlns="urn:jboss:aop-beans:1.0">
</aop>
```

If you want to create anything more than a non-trivial example, using the **.aop** JAR files, you can make any top-level deployment contain an AOP file containing the XML binding configuration. For instance you can have an AOP file in an EAR file, or an AOP file in a WAR file. The bindings specified in the **META-INF/jboss-aop.xml** file contained in the AOP file will affect all the classes in the whole WAR file.

To pick up an AOP file in an EAR file, it must be listed in the **.ear/META-INF/application.xml** as a Java module, as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.2//EN" 'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  <display-name>AOP in JBoss example</display-name>
  <module>
    <java>example.aop</java>
  </module>
  <module>
    <ejb>aopexampleejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>aopexample.war</web-uri>
      <context-root>/aopexample</context-root>
    </web>
  </module>
</application>
```



Important

In the JBoss Enterprise Web Platform 5, the contents of the **.ear** file are deployed in the order they are listed in the **application.xml**. When using loadtime weaving the bindings listed in the **example.aop** file must be deployed before the classes being advised are deployed, so that the bindings exist in the system before (for example) the **ejb** and **servlet** classes are loaded. This is achieved by listing the AOP file at the start of the **application.xml**. Other types of archives are deployed before anything else and so do not require special consideration, such as **.sar** and **.war** files.

9.5. The JBoss AspectManager Service

The **AspectManager** Service can be managed at runtime using the JMX console, which is found at **http://localhost:8080/jmx-console**. It is registered under the ObjectName **jboss.aop:service=AspectManager**. If you want to configure it on startup you need to edit some configuration files.

In JBoss Enterprise Web Platform 5 the **AspectManager** Service is configured using a JBoss Microcontainer bean. The configuration file is **jboss-as-web/server/\$PROFILE/conf/bootstrap/aop.xml**. The **AspectManager** Service is deployed with the following XML:

```

<bean name="AspectManager" class="org.jboss.aop.deployers.AspectManagerJDK5">

<property name="jbossIntegration"><inject bean="AOPJBossIntegration"/></property>

<property name="enableLoadtimeWeaving">false</property>
<!-- only relevant when EnableLoadtimeWeaving is true.
When transformer is on, every loaded class gets transformed.
If AOP can't find the class, then it throws an exception.
Sometimes, classes may not have all the classes they reference.
So, the Suppressing is needed. (For instance, JBoss cache in the default
configuration) -->

<property name="suppressTransformationErrors">true</property>

<property name="prune">true</property>

<property name="include">org.jboss.test., org.jboss.injbossaop.</property>

<property name="exclude">org.jboss.</property>
<!-- This avoids instrumentation of hibernate cglib enhanced proxies

<property name="ignore">*$$EnhancerByCGLIB$$*</property> -->

<property name="optimized">true</property>

<property name="verbose">false</property>
<!-- Available choices for this attribute are:
org.jboss.aop.instrument.ClassicInstrumentor (default)
org.jboss.aop.instrument.GeneratedAdvisorInstrumentor -->

<!-- <property
name="instrumentor">org.jboss.aop.instrument.ClassicInstrumentor</property>-->

<!-- By default the deployment of the aspects contained in
../deployers/jboss-aop-jboss5.deployer/base-aspects.xml
are not deployed. To turn on deployment uncomment this property
<property name="useBaseXml">true</property>-->
</bean>

```

Later we will talk about changing the class of the **AspectManager** Service. To do this, replace the contents of the **class** attribute of the **bean** element.

9.6. Loadtime transformation in the JBoss Enterprise Web Platform Using Sun JDK

The JBoss Enterprise Web Platform has special integration with JDK to do loadtime transformations. This section explains how to use it.

If you want to do load-time transformations with JBoss Enterprise Web Platform 5 and Sun JDK, these are the steps you must take.

- ▶ Set the **enableLoadtimeWeaving** attribute/property to **true**. By default, JBoss Application Server will not do load-time bytecode manipulation of AOP files unless this is set. If **suppressTransformationErrors** is **true**, failed bytecode transformation will only give an error warning. This flag is needed because sometimes a JBoss deployment will not include all of the classes referenced.
- ▶ Copy the **pluggable-instrumentor.jar** from the **lib/** directory of your JBoss AOP distribution to the **bin/** directory of your JBoss Enterprise Web Platform.
- ▶ Next edit **run.sh** or **run.bat** (depending on what OS you're on) and add the following to the **JAVA_OPTS** environment variable:

```
set JAVA_OPTS=%JAVA_OPTS% -Dprogram.name=%PROGNAME% -javaagent:pluggable-
instrumentor.jar
```



Important

The class of the AspectManager Service must be **org.jboss.aop.deployers.AspectManagerJDK5** or **org.jboss.aop.deployment.AspectManagerServiceJDK5** as these are what work with the **-javaagent** option.

9.7. JRockit

JRockit also supports the **-javaagent** switch mentioned in [Section 9.6, “Loadtime transformation in the JBoss Enterprise Web Platform Using Sun JDK”](#). If you wish to use that, then the steps in [Section 9.6, “Loadtime transformation in the JBoss Enterprise Web Platform Using Sun JDK”](#) are sufficient. However, JRockit also comes with its own framework for intercepting when classes are loaded, which might be faster than the **-javaagent** switch. If you want to do load-time transformations using the special JRockit hooks, these are the steps you must take.

- ▶ Set the **enableLoadtimeWeaving** attribute/property to true. By default, JBoss Enterprise Web Platform will not do load-time bytecode manipulation of AOP files unless this is set. If **suppressTransformationErrors** is **true**, failed bytecode transformation will only give an error warning. This flag is needed because sometimes a JBoss deployment will not include all the classes referenced.
- ▶ Copy the **jrockit-pluggable-instrumentor.jar** from the **lib/** directory of your JBoss AOP distribution to the **bin/** directory of your the JBoss Enterprise Web Platform installation.
- ▶ Next edit **run.sh** or **run.bat** (depending on what OS you're on) and add the following to the **JAVA_OPTS** and **JBOSS_CLASSPATH** environment variables:

```
# Setup JBoss specific properties

JAVA_OPTS="$JAVA_OPTS -Dprogram.name=$PROGNAME \
-Xmanagement:classpath=org.jboss.aop.hook.JRockitPluggableClassPreProcessor"

JBOSS_CLASSPATH="$JBOSS_CLASSPATH:jrockit-pluggable-instrumentor.jar"
```

- ▶ Set the class of the **AspectManager** Service to **org.jboss.aop.deployers.AspectManagerJRockit** on JBoss Enterprise Web Platform 5, or **org.jboss.aop.deployment.AspectManagerService** as these are what work with special hooks in JRockit.

9.8. Improving Loadtime Performance in the JBoss Enterprise Web Platform Environment

The same rules apply to the JBoss Enterprise Web Platform for tuning loadtime weaving performance as standalone Java. Switches such as **prune**, **optimized**, **include** and **exclude** are configured through the **jboss-as-web/server/\$PROFILE/conf/bootstrap/aop.xml** file talked about earlier in this chapter.

9.9. Scoping the AOP to the classloader

By default all deployments in JBoss are global to the whole application server. That means that any EAR,

SAR, or JAR (for example), that is put in the deploy directory can see the classes from any other deployed archive. Similarly, AOP bindings are global to the whole virtual machine. This *global* visibility can be turned off per top-level deployment.

9.9.1. Deploying as part of a scoped classloader

The following process may change in future versions of JBoss AOP. If you deploy an AOP file as part of a scoped archive, the bindings (for instance) applied within the `.aop/META-INF/jboss-aop.xml` file will only apply to the classes within the scoped archive and not to anything else in the application server. Another alternative is to deploy `-aop.xml` files as part of a service archive (SAR). Again, if the SAR is scoped, the bindings contained in the `-aop.xml` files will only apply to the contents of the SAR file. It is not currently possible to deploy a standalone `-aop.xml` file and have that attach to a scoped deployment. Standalone `-aop.xml` files will apply to classes in the whole application server.

9.9.2. Attaching to a scoped deployment

If you have an application that uses classloader isolation, as long as you have prepared your classes, you can later attach an AOP file to that deployment. If we have an EAR file scoped using a `jboss-app.xml` file, with the scoped loader repository `jboss.test:service=scoped`:

```
<jboss-app>
  <loader-repository>
    jboss.test:service=scoped
  </loader-repository>
</jboss-app>
```

We can later deploy an AOP file containing aspects and configuration to attach that deployment to the scoped EAR. This is done using the `loader-repository` tag in the AOP file's `META-INF/jboss-aop.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <loader-repository>jboss.test:service=scoped</loader-repository>

  <!-- Aspects and bindings -->
</aop>
```

This has the same effect as deploying the AOP file as part of the EAR as we saw previously, but allows you to hot deploy aspects into your scoped application.

Chapter 10. Transaction Management

10.1. Overview

Transaction support in JBoss Enterprise Web Platform is provided by JBoss Transaction Service, a mature, modular, standards-based, highly configurable transaction manager. By default the server runs with the local-only JTA module of JBossTS installed. This module provides an implementation of the standard JTA API for use by other internal components, such as the EJB container, as well as direct use by application code. It is suitable for coordinating ACID transactions that involve one or more XA Resource managers, such as databases or message queues.

10.2. Configuration Essentials

Configuration of the default JBoss Transaction Service JTA is managed through a combination of the transaction manager's own properties file and the application server's deployment configuration. The configuration file resides at `server/[name]/conf/jbossts-properties.xml`. It contains defaults for the most commonly used properties. Many more are detailed in the accompanying JBossTS documentation and can be added to the configuration file as needed. All also have hard-coded defaults, but the system may not function exactly as expected in the absence of a properties file. The configuration given in this file is supplemented by the microcontainer beans configuration found in the `server/[name]/deploy/transaction-jboss-beans.xml` file. This ties the transaction manager into the wider server configuration, overriding the transaction config file settings with application server specific values where appropriate. In particular, it uses the service binding manager to set port binding information and overrides selected other properties. Configuration properties are read by JBossTS at server initialization and changes made thereafter, either to the properties file, beans file, or programmatically, will not have an effect until server (JVM) restart.

Critical Bean Properties for JTA

transactionTimeout

The default time in seconds before a transaction will be considered stuck and may be rolled back by the transaction manager. This helps to prevent poor code from blocking the system indefinitely.

The default value is **300** seconds. This should be adjusted to suit your environment and workload.

Transaction timeouts are processed asynchronously, which may not be appropriate for your application.

objectStoreDir

Defines the directory in which to log transaction data. This bean property overrides the **jbossts-properties.xml** configuration file value for **com.arjuna.ats.arjuna.objectstore.objectStoreDir**.

This transaction log is required to complete transactions in case of system failure. The storage device used must therefore be highly performant and reliable. In general, local RAID disk is preferred. Remote storage can be used if file locking is correctly implemented. However, requiring network I/O can be a significant performance bottleneck.

The **ObjectStore** usually contains one file of several kilobytes per in-flight transaction. These are distributed over a directory tree for optimal performance. The small file size and rapid creation/deletion of files lends itself well to SSD-based storage devices. If used, RAID controllers should be configured for write through cache, similarly to database storage devices. Writing the transaction log is automatically skipped for transactions that are rolling back or contain only a single resource.

Configuration Properties In `jboss-transaction.xml`

`com.arjuna.common.util.logging.DebugLevel`

This setting determines the internal log threshold for the transaction manager codebase. It is independent of the server's wider log4j logging configuration and represents an additional hurdle that log messages must pass before being printed. The default value is `0x00000000`, that is, no debug logging. **INFO** and **WARN** messages will still be printed by default. This provides optimal performance. The value `0xffffffff` should be used when full debug logging is required. This is very verbose and will result in large log files. Log messages that pass the internal `DebugLevel` check will be passed to the server's logging system for further processing. Thus it may also be necessary to set appropriate configuration for `com.arjuna` code in the `server/$PROFILE/conf/jboss-log4j.xml` file. Note that whilst a value of `0xffffffff` may be left in place permanently and the log4j settings used to turn logging on or off, this is less performant than using the internal `DebugLevel` checking.

`com.arjuna.ats.arjuna.coordinator.commitOnePhase`

This setting determines if the transaction manager will automatically apply the one-phase commit optimization to the transaction completion protocol in cases where only a single resource is registered with the transaction. It is enabled (set to **YES**) by default to provide optimal performance, since no transaction log write is necessary in such cases. Some resource managers may not be compatible with this optimization and it is occasionally necessary to disable it. This can be done by changing the value to **NO**.

`com.arjuna.ats.arjuna.objectstore.transactionSync`

This setting controls the flushing of transaction logs to disk during the transaction termination. It is enabled (set to **ON**) by default, which results in a `FileDescriptor.sync` call for each committing transaction. This is required to provide recovery guarantees and hence ACID properties. If the applications running in the server can tolerate data inconsistency or loss, greater performance may be achieved by disabling this behavior by setting the property value to "OFF". This is not recommended – it is usually preferable to recraft such applications to avoid using the transaction manager entirely.

`com.arjuna.ats.arjuna.xa.nodeIdentifier` and `com.arjuna.ats.jta.xaRecoveryNode`

These properties determine the behavior of the transaction recovery system. Correct configuration is essential to ensure transactions are resolved correctly in the event of a server crash and restart. See the crash recovery section that follows for details.

`com.arjuna.ats.arjuna.coordinator.enableStatistics`

This property enables the gathering of transaction statistics, which may be viewed via methods on the **TransactionManagerService** bean or, more commonly, its corresponding JMX MBean. This option is disabled by default, as the additional locking needed to record statistics accurately may cause a slight performance impact. Thus the statistics getter methods will thus normally return zero values. To enable the option, set its value to **YES** in the properties file.

10.3. Transactional Resources

The transaction manager coordinates the update of state via **XAResource** implementations, which are provided by the various resource managers. In most instances, resource managers will be databases, message queues or third-party JCA resource adapters. The list of JDBC database drivers and servers

certified for use with JBoss Enterprise Web Platform can be found on the redhat.com website. In addition there is a reasonable probability of any driver that complies with the relevant standards functioning correctly. However, interpretation of the XA specification does differ from one vendor to another, as does quality of driver code. For maximum surety in transactional applications, thorough testing is essential, especially with regard to recovery behavior.

Database connection pools configured via the application server's `*-ds.xml` files using `<xa-datasource>` will automatically interact with the transaction manager. Connections obtained by looking up such datasource in JNDI and calling `getConnection` will automatically participate correctly in an ongoing transaction. This is the dominant use case and should be preferred where transactional guarantees for data access are required. For cases where the database cannot support XA transactions, it is also feasible to deploy a connection pool using `<local-xa-datasource>`. Such datasources participate in the managed transaction using the last resource commit optimization (see below) and as such provide more limited transactional guarantees. Applications using this approach should be aware of the limitations and implemented accordingly. Connections obtained from a `<no-tx-datasource>` will not interact with the transaction manager and any work done on such connections must be explicitly committed or rolled back by the application via the JDBC API.

Many databases require additional configuration if they are to be used as XA resource managers. For example, MS SQL Server requires configuration of the DTC service and installation of a server side component of the JDBC drivers. Some versions of Oracle similarly require a server side package to be installed in the database instance. PostgreSQL installations may require an alteration to the number of outstanding transactions they permit the default is normally too low for production usage. MySQL has significant limitations on its XA implementation and is not recommended for use in an XA transaction. If it is used, the InnoDB storage engine must be configured. Please consult your database administrator or database documentation for further product-specific information. In addition, it is important to take any further database configuration steps needed to support XA recovery, see the recovery section below.



Note

The Java EE Connector Architecture container keeps a dedicated physical connection open against the EIS where recovery is performed. Therefore, set the `max-pool-size` to the maximum number of connection possible minus 1.

10.4. Last Resource Commit Optimization (LRCO)

Although the XA transaction protocol is designed to provide ACID properties by using a two-phase commit protocol, it is recognized that this is not possible in all circumstances. In particular, it is occasionally necessary to have a resource manager that is not XA aware participate in the transaction. This is often the case with data stores that can't or won't support distributed transactions. For such circumstances it is possible to employ a technique variously known as the Last Resource Gambit or Last Resource Commit Optimization (LRCO). Using this technique, the one phase resource is processed last in the prepare phase of the transaction, at which time an attempt is made to commit it. If successful, the transaction log is written and the remaining resources go through the phase two commit. If the last resource fails to commit, the transaction is rolled back. Whilst this protocol allows for most transactions to complete normally, certain types of error can cause an inconsistent transaction outcome. Therefore, we recommend using this approach only when no alternative is available. Where a single `<local-tx-datasource>` is used in a transaction, the LRCO will be automatically applied to it. For other cases it is possible to designate a last resource by using a special marker interface. See the JBoss Transactions documentation for details.

It is not transactionally safe (or rather, it is even more unsafe) to use more than a single one-phase resource in the same transaction. For this reason JBoss Transactions treats an attempt to enlist a second such resource as an error and will terminate the transaction. This use case is most commonly found in applications migrating from JBoss Application Server 4.0.x servers, where this usage was not considered an error. Whenever possible the `<local-tx-datasource>` should be changed to `<xa-`

datasource> to resolve the difficulty. Where this is not possible, the transaction manager may be configured to allow multiple last resources, although this is not recommended.

10.5. Transaction Timeout Handling

In order to prevent indefinite locking of resources, the transaction manager will abort in-flight transactions that have not completed after a specified interval. This abort is done by a set of background processes, coordinated by the **TransactionReaper**. This reaper will roll back transactions without interrupting any threads that may be operating within their scope. This prevents instability that can result from interrupting threads executing arbitrary code. Furthermore, it allows for timely abort of transactions where the business logic thread may be executing non-interruptable operations such as network I/O calls. This approach may, however, cause unexpected behavior in code that is not designed to handle multithreaded transactions. Warning or error messages may be printed from Hibernate or other transaction-aware components as a result of the unexpected transaction status change. These should not affect the transaction outcome. The problem can be minimized by appropriate tuning of the transaction timeout.

10.6. Recovery Configuration

JBoss Enterprise Web Platform now includes recovery auto-registration in the JCA. Thus, the `AppServerJDBCXARecovery` property which was used in previous releases is disabled by default, and will be removed entirely from future releases of the Platform.

10.7. Transaction Service FAQ

Q: I turned on debug logging, but nothing is logged.

A: JBossTS sends log statements through two levels of filters.

1. Logs go through JBoss Transaction Service's own logging abstraction layer.
2. Logs go through JBoss Enterprise Application Platform's **log4j** logging system.

A log statement must pass both filters to be printed. A typical mistake is enabling only one or the other of the logging systems.

Q: Why do server logs show `WARN Adding multiple last resources is disallowed.`, and why are my transactions are aborted?

A: You are probably using a `<local-xa-datasource>` and trying to use more than one one-phase aware participant. This is a configuration to be avoided. If you have further concerns, please contact Global Support Services.

Q: My server terminated unexpectedly. It is running again, but my logs are filling with messages like `WARN [com.arjuna.ats.jta.logging.loggerI18N] [com.arjuna.ats.internal.jta.resources.arjunacore.norecoveryxa] Could not find new XAResource to use for recovering non-serializable XAResource.`

A: You may not have configured all resource managers for recovery. Refer to the Recovery chapter of the JBoss Transactions Administration Guide for more information on configuring resource managers for recovery.

Q: My transactions take a long time and sometimes strange things happen. The server log contains `WARN [arjLoggerI18N] [BasicAction_58] - Abort of action id ... invoked while multiple threads active within it.`

A: Transactions which exceed their timeout may be rolled back. This is done by a background thread, which can confuse some application code that may be expecting an interrupt.

If you have questions besides the ones addressed above, please consult the other JBoss Transactions guides, or contact Global Support Services.

Chapter 11. Use Alternative Databases with JBoss Enterprise Platform

11.1. How to Use Alternative Databases

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 Enterprise Platform to use alternative databases. We cover the procedures for all officially supported databases on the JBoss Enterprise Platform. For a complete list of certified databases, refer to

<http://www.jboss.com/products/platforms/application/supportedconfigurations/>.

Please note that in this chapter, we explain how to use alternative databases to support all services in JBoss Enterprise Platform. This includes all the system level services such as EJB and JMS. For individual applications (e.g., WAR or EAR) deployed in JBoss Enterprise Platform, you can still use any backend database by setting up the appropriate data source connection.

Installing the external database is out of the scope of this document. Use the tools provided by your database vendor to set up an empty database. You will need the database name, connection URL, username, and password, in order to create the datasources the Platform will use to connect to the database.

11.2. Install JDBC Drivers

To use the selected external database, you must also install the JDBC driver for your database. The JDBC driver is a JAR file, which must be placed into the **JBOSS_HOME/server/PROFILE/lib** directory. Replace **PROFILE** with the server profile you are using.

This file is loaded when JBoss Enterprise Platform starts up, so if you have Platform running, you will need to shut down and restart. Review the list below for a suitable JDBC driver. For a full list of certified JBoss Enterprise Platform database drivers, refer to <http://www.jboss.com/products/platforms/application/supportedconfigurations/#JEAP5-0>. If the links fail to work, please file a JIRA against this documentation, but be aware that Red Hat does not control these external links. Contact your database vendor for the most current version of the driver for your database.

JBDC Driver Download Locations

MySQL

Download from <http://www.mysql.com/products/connector/>.

PostgreSQL

Download from <http://jdbc.postgresql.org/>.

Oracle

Download from http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html.

IBM

Download from <http://www-306.ibm.com/software/data/db2/java/>.

Sybase

Download from the Sybase jConnect product page
<http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect>.



Sybase jConnect JDBC Driver 7

When using Sybase database with this driver, the **MaxParams** attribute cannot be set higher than **481** due to a limitation in the driver's **PreparedStatement** class.

Microsoft

Download from the MSDN web site <http://msdn.microsoft.com/data/jdbc/>.

11.2.1. 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 to the Sybase manuals for more options.

Additionally, text and image values stored in the database can be very large. When a select list includes both text and image values, the length limit of the data returned is determined by the **@@textsize** global variable. The default setting for this variable depends on the software used to access Adaptive Server. For the JDBC driver, the default value is 32 kilobytes.

11.2.1.1. Enable JAVA services

To use any Java service (for example; JMS, CMP, timers) configured with Sybase, Java must be enabled on Sybase Adaptive Server. To do this use:

```
sp_configure "enable java",1
```

Refer to the sybase manuals for more information.

If Java is not enabled for Sybase Adaptive Server, the following error message may be echoed in the console.

```
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
```

11.2.1.2. 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 to the **installjava** manual in Sybase for more options.

11.2.1.3. Installing Java Classes

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 requires the Java classes to be compiled with the same.

11.2.2. Configuring JDBC DataSources

Datasources correspond to the simplified JCA Datasource configuration specifications.

Datasources need to reside in the **`$JBOSS_HOME/server/$PROFILE/deploy`** directory, alongside other deployable applications and resources. The files use a standard naming scheme of **`DBNAME-ds.xml`**.

Example datasources for all certified databases are located in the **`$JBOSS_HOME/docs/examples/jca`** directory. Edit the datasource that corresponds to your database, and copy it to the **`deploy/`** directory before restarting the application server.

See [Chapter 12, Datasource Configuration](#) for information on configuring datasources. As a minimum, you will need to change the **`connection-url`**, **`user-name`**, and **`password`** to correspond to your database of choice.

11.3. Common Database-Related Tasks

11.3.1. Security and Pooling

Unless the **`ResourceAdapter`** has **`<reauthentication-support>`**, using multiple security identities will create subpools for each identity.



Note

The min and max pool size are per subpool, so be careful with these parameters if you have lots of identities.

11.3.2. Change Database for the JMS Services

The JMS service in the JBoss Enterprise Platform uses relational databases to persist its messages. For improved performance, we should change the JMS service to take advantage of the external database. To do that, we need to replace the file

`$JBOSS_HOME/server/$PROFILE/deploy/messaging/$DATABASE-persistence-service.xml` with the **`$DATABASE-persistence-service.xml`** filename depending on your external database.

- MySQL: **`mysql-persistence-service.xml`**
- PostgreSQL: **`postgresql-persistence-service.xml`**
- Oracle: **`oracle-persistence-service.xml`**
- DB2: **`db2-persistence-service.xml`**
- Sybase: **`sybase-persistence-service.xml`**
- MS SQL Server: **`mssql-persistence-service.xml`**

11.3.3. Support Foreign Keys in CMP Services

Next, we need to go change the **`$JBOSS_HOME/server/$PROFILE/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 Platform. This file configures the database connection settings for the EJB2 CMP beans deployed in the JBoss Enterprise Platform.

```
<fk-constraint>true</fk-constraint>
```

11.3.4. 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 Enterprise Platform. Hibernate has a dialect auto-detection mechanism that works for most databases including the dialects for databases referenced in this appendix which are listed below. If a specific dialect is needed for alternative databases, you can configure the database dialect in the **\$JBOSS_HOME/server/\$PROFILE/deployers/ejb3.deployer/META-INF/jpa-deployers-jboss-beans.xml** file. To configure this file you need to uncomment the set of tags related to the map entry **hibernate.dialect** and change the values to the following based on the database you setup.

- ▶ Oracle 10g: **org.hibernate.dialect.Oracle10gDialect**
- ▶ Oracle 11g: **org.hibernate.dialect.Oracle10gDialect**
- ▶ Microsoft SQL Server 2005: **org.hibernate.dialect.SQLServerDialect**
- ▶ Microsoft SQL Server 2008: **org.hibernate.dialect.SQLServerDialect**
- ▶ PostgreSQL 8.2.3: **org.hibernate.dialect.PostgreSQLDialect**
- ▶ PostgreSQL 8.3.7: **org.hibernate.dialect.PostgreSQLDialect**
- ▶ MySQL 5.0: **org.hibernate.dialect.MySQL5InnoDBDialect**
- ▶ MySQL 5.1: **org.hibernate.dialect.MySQL5InnoDBDialect**
- ▶ DB2 9.1: **org.hibernate.dialect.DB2Dialect**
- ▶ Sybase ASE 15: **org.hibernate.dialect.SybaseASE15Dialect**

11.3.5. Change Other JBoss Enterprise Platform 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.

11.3.5.1. 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 would change **MySqlDS** to **DefaultDS** and so on. You will need to remove the **\$JBOSS_HOME/server/\$PROFILE/deploy/hsqldb-ds.xml** file after you are done to avoid duplicated **DefaultDS** definition.

In the **messaging/\$DATABASE-persistence-service.xml** file, you should also change the datasource name in the **depends** tag for the **PersistenceManagers** MBean to **DefaultDS**. For instance, for **mysql-persistence-service.xml** file, we change the **MySqlDS** to **DefaultDS**.

```
<mbean
    code="org.jboss.messaging.core.jmx.JDBCPersistenceManagerService"
    name="jboss.messaging:service=PersistenceManager"
    xmbean-dd="xmdesc/JDBCPersistenceManager-xmbean.xml">

    <depends>jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
```

11.3.5.2. 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 Enterprise Platform 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 (for example, 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_HOME/server/\$PROFILE/conf/login-config.xml**: This file is used in Java EE container managed security services.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/conf/standardjbosscmp-jdbc.xml**: This file configures the CMP beans in the EJB container.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/deploy/ejb2-timer-service.xml**: This file configures the EJB timer services.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/deploy/juddi-service.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/deploy/juddi-service.sar/juddi.war/WEB-INF/jboss-web.xml**: This file configures the UUDI service.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/deploy/juddi-service.sar/juddi.war/WEB-INF/juddi.properties**: This file configures the UUDI service.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- ▶ **\$JBOSS_HOME/server/\$PROFILE/deploy/messaging/messaging-jboss-beans.xml** and **\$JBOSS_HOME/server/\$PROFILE/deploy/messaging/persistence-service.xml**: Those files configure the JMS persistence service as we discussed earlier.

11.3.6. A Special Note About Oracle Databases

In our setup discussed in this chapter, we rely on the JBoss Enterprise Platform 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 Enterprise Platform instance.

The Oracle database creates tables of the form **schemaname.tablename**. The **TIMERS** and **HILOSEQUENCES** tables needed by JBoss Enterprise Platform would not be created on a schema if the table already existed on a different schema. To work around this issue, you need to edit the **\$JBOSS_HOME/server/\$PROFILE/deploy/ejb2-timer-service.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_HOME/server/\$PROFILE/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</depends>

    <!-- table name -->
    <attribute name="TableName">HILOSEQUENCES</attribute>
```



Regression in Oracle JDBC driver 11.1.0.7.0

Oracle JDBC driver version 11.1.0.7.0 causes the JBoss Messaging Test Suite to fail with a **SQLException** ("Bigger type length than Maximum") on Oracle 11g R1. This is caused by a regression in Oracle JDBC driver 11.1.0.7.0. We recommend Oracle JDBC driver version 11.2.0.1.0 for use with Oracle 11g R1, Oracle 11g R2, Oracle RAC 11g R1 and Oracle RAC 11g R2.

Chapter 12. Datasource Configuration



You must change your database

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.

12.1. 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.

12.2. Datasource Parameters

Common Datasource Parameters

`<mbean>`

A standard JBoss MBean deployment.

`<depends>`

The **ObjectName** of an MBean service this **ConnectionFactory** or **DataSource** deployment depends upon.

`<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 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 12.1. Using <share-prepared-statements>

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

```
Connection c = dataSource.getConnection(); // auto-commit == false
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. 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 seperate thread (ConnectionValidator). <background-validation-minutes> depends on this value also being set to **TRUE**. Defaults to **FALSE**.

<background-validation-millis>

Background connection validation reduces the overall load on the RDBMS system when validating a connection. Setting this parameter means that JBoss will attempt to validate the current connections in the pool as a separate thread (**ConnectionValidator**). This parameter's value defines the interval, in milliseconds, for which the **ConnectionValidator** will run. (This value should not be the same as your <idle-timeout-minutes value.)

<idle-timeout-minutes>

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

<track-connection-by-tx>

Whether the connection should be locked to the transaction, instead of returning it to the pool at the end of the transaction. In previous releases, this was **true** for local connection factories and **false** for XA connection factories. The default is now **true** for both local and XA connection factories, and the element has been deprecated.

<interleaving>

Enables interleaving for XA connection factories.

<background-validation-minutes>

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



Note

You should set this to a smallervalue 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 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 12.2. 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 12.3. 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>
```

`<xa-resource-timeout>`

The number of seconds passed to `XAResource.setTransactionTimeout()` when not zero.

<isSameRM-override-value>

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

<no-tx-separate-pools>

Pool transactional and non-transactinal 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.

Security Parameters

<application-managed-security>

Uses the username and password passed on the `getConnection` or `createConnection` request by the application.

<security-domain>

Uses the identified login module configured in `conf/login-module.xml`.

<security-domain-and-application>

Uses the identified login module configured in `conf/login-module.xml` and other connection request information supplied by the application, for example JMS Queues and Topics.

Parameters for XA Recovery in the JCA Layer

<recover-user-name>

The user with credentials to perform a recovery operation.

<recover-password>

Password of the user with credentials to perform a recovery operation.

<recover-security-domain>

Security domain for recovery.

<no-recover>

Excludes a datasource from recovery.

The fields in [Parameters for XA Recovery in the JCA Layer](#) should have a fall back value of their non-recover counterparts: `<user-name>`, `<password>` and `<security-domain>`.

12.3. Datasource Examples

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

12.3.1. Generic Datasource Example

Example 12.4. 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>

```

12.3.2. Configuring a DataSource for Remote Usage

JBoss EAP supports accessing a DataSource from a remote client. See [Example 12.5, “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 12.5. 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.

12.3.3. Configuring a Datasource to Use Login Modules

Procedure 12.1. 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=OracleDS
        JAAS</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 13. Pooling

13.1. Strategy

[JBossJCA](#) uses a **ManagedConnectionPool** to perform the pooling. The **ManagedConnectionPool** is made up of subpools depending upon the strategy chosen and other pooling parameters.

XML	MBean	Internal Name	Description
	ByNothing	OnePool	A single pool of equivalent connections
<application-managed-security/>	ByApplication	PoolByCRI	Use the connection properties from allocateConnection()
<security-domain/>	ByContainer	PoolBySubject	A pool per Subject, for example, preconfigured or EJB/Web login subjects
<security-domain-and-application/>	ByContainerAndApplication	PoolBySubjectAndCri	A per Subject and connection property combination



Note

Despite the XML names, these properties do not only relate to security.

For **<security-domain-and-application/>** the Subject always overrides any user/password from `createConnection(user, password)` in the CRI:

```
(
  ConnectionRequestInfo
)
```

13.2. Sticky Transactions

Transaction stickiness is enabled by default in JBoss Enterprise Web Platform. The **<track-connection-by-tx/>** flag, which forced a pool connection to be reused throughout a transaction, is deprecated. This is the only supported behavior for local transactions. XA users can explicitly enable interleaving with the **<interleaving/>** element.

13.3. Workaround for Oracle

Oracle does not like XA connections getting used both inside and outside a JTA transaction. To work around the problem you can create separate sub-pools for the different contexts using **<no-tx-separate-pools/>**.

13.4. Pool Access

The pool is designed for concurrent usage.

Up to **<max-pool-size/>** threads can be inside the pool at the same time (or using connections from a pool).

Once this limit is reached, threads wait for a period defined by `<blocking-timeout-seconds/>` to use the pool before throwing a **NoManagedConnectionsAvailable** exception.

The `<allocation-retry/>` and `<allocation-retry-wait-millis/>` elements let the pool retry a connection attempt before throwing the exception.

13.5. Pool Filling

The number of connections in the pool is controlled by the pool sizes.

`<min-pool-size/>`

New connections are created when the number of connections falls below the number specified.

`<max-pool-size/>`

The maximum number of connections.

`<prefill/>`

Supported only by **OnePool** or **ByNothing** pooling strategies.

Filling the thread pool is done by a separate **PoolFiller** thread to avoid blocking application threads.



Pool Filling to Minimum Pool Size

Connection pools are filled to their min-pool-size only on their first usage.

13.6. Idle Connections

You can configure connections to close when they are idle with the `<idle-timeout-minutes/>`.

Idle checking is done on a separate **IdleRemover** thread on a LRU (least recently used) basis. The check is done twice per **idle-timeout-minutes** period for connections that have not been used for **idle-timeout-minutes**.

The pool itself operates on an MRU (most recently used) basis. This allows the excess connections to be easily identified.

Should closing idle connections cause the pool to fall below the **min-pool-size**, new connections are created.



Note

If you have long running transactions and you use interleaving (that is, do not enable **track-connection-by-tx**), make sure the idle timeout is greater than the transaction timeout. When interleaving, the connection is returned to the pool for others to use. If it remains unused, it still becomes a candidate for removal before the transaction is committed.

13.7. Dead connections

The JDBC protocol does not provide a natural **connectionErrorOccured()** event when a connection is broken. There are a number of plugins available to support dead/broken connection

checking.

13.7.1. Valid connection checking

The simplest option is to run an SQL statement like the following before handing the connection to the application.

```
<check-valid-connection-sql>select 1 from dual</check-valid-connection-sql>
```

If this fails, another connection is selected until there are no more connections at which point new connections are constructed.

The potentially more performant check is to use vendor specific features, such as Oracle's or MySQL's `pingDatabase()`, with the `<valid-connection-checker-class-name/>` element.

13.7.2. Errors during SQL queries

You can check whether a connection broke during a query by looking for **FATAL** errors in the `SQLException`'s error messages. These messages can be vendor-specific, for example:

```
<exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</exception-sorter-class-name>
```

FATAL errors will close the connection.

13.7.3. Changing/Closing/Flushing the pool

Use JMX to change the attributes on the connection pool, like so:

```
jboss.jca:service=JBossManagedConnectionPool,name=<jndi-name>
```

Then use `flush()` to reset the pools. `flush()` closes all idle connections immediately. Any connections in use are closed after the application is finished with them. New connections are then created.

Chapter 14. Frequently Asked Questions

14.1. I have problems with Oracle XA

Check the following:

- Is **pad=true** for the **XidFactory** in **conf/jboss-service.xml**?
- Is **<track-connection-by-tx/>** set in your **oracle-xa-ds.xml**? (This may not be required. The element is deprecated, and enabled by default.)
- Is **<isSameRM-override-value>>false</isSameRM-override-value>** set in your **oracle-xa-ds.xml**?
- Is **<no-tx-separate-pools/>** set in your **oracle-xa-ds.xml**?
- Does **jbosscomp-jdbc.xml** specify the version of Oracle you are using?
- Does the Oracle server you connect to support XA?

You can configure Oracle database to support XA resources. This enables you to use JDBC 2.0-compliant Oracle driver. To XA-initialize Oracle database, complete the following steps:

Ensure that Oracle JServer is installed with your database. If it is not installed, add it using the **Oracle Database Configuration Assistant**.

1. Launch the **Oracle Database Configuration Assistant**. You can do this by typing **dbca** at the command prompt. In a Windows environment, you can launch it from the **Start** menu, under **Configuration Assistants**.
2. Select "Change an Existing DB".
3. Select the database to which you wish to add Oracle JServer. Then, click **Next**.
4. Select **Oracle JServer**, and then click **Finish**.

If your previous database settings are not suitable or insufficient for the Oracle JServer installation, you will be prompted to enter additional parameters. The database configuration file, **init.ora**, is located in **\oracle\admin\\${DATABASE_NAME}\pfile**.

Execute **initxa.sql** over your database. By default, this script is located in **\oracle\ora81\javavm\install**. If errors occur during execution, you must execute the SQL statements in the script manually.

Use DBA Studio to create a package and package body named **JAVA_XA** in **SYS** schema, and an identically named **JAVA_XA** in **PUBLIC** schema.

More detailed instructions can be found at http://www.ibm.com/developerworks/websphere/library/techarticles/0407_woolf/0407_woolf.html?ca=dnp-327#oracle_exception, *Configuring and using XA distributed transactions in WebSphere Studio - OracleXAException*.

Part III. Clustering Guide

Chapter 15. Introduction and Quick Start

Clustering allows you to run an application on several parallel servers (also known as 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 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 Enterprise Web Platform comes with clustering support out of the box, as part of the **production** configuration. The **production** configuration includes support for the following:

- A scalable, fault-tolerant JNDI implementation (HA-JNDI).
- Web tier clustering, including:
 - High availability for web session state via state replication.
 - Ability to integrate with hardware and software load balancers, including special integration with `mod_jk` and other JK-based software load balancers.
 - Single Sign-on support across a cluster.
- A distributed cache for JPA/Hibernate entities.
- Deploying a service or application on multiple nodes in the cluster but having it active on only one (but at least one) node is called a *HA Singleton*.
- Keeping deployed content in sync on all nodes in the cluster via the **Farm** service.

This part of the *Administration and Configuration Guide* aims to provide you with an in depth understanding of how to use JBoss Enterprise Web Platform's clustering features. In this first part of the guide, the goal is to provide some basic "Quick Start" steps to encourage you to start experimenting with JBoss Enterprise Web Platform Clustering, and then to provide some background information that will allow you to understand how JBoss Enterprise Web Platform Clustering works. The next part of the guide then explains in detail how to use these features to cluster your JEE services. Finally, we provide some more details about advanced configuration of JGroups and JBoss Cache, the core technologies that underlie JBoss Enterprise Web Platform Clustering.

15.1. Quick Start Guide

The goal of this section is to give you the minimum information needed to let you get started experimenting with JBoss Enterprise Web Platform Clustering. Most of the areas touched on in this section are covered in much greater detail later in this guide.

15.1.1. Initial Preparation

Preparing a set of servers to act as a JBoss Enterprise Web Platform cluster involves a few simple steps:

- **Install JBoss Enterprise Web Platform on all your servers.** In its simplest form, this is just a matter of unzipping the JBoss download onto the filesystem on each server. For more information, see the *Installation Guide*.

If you want to run multiple JBoss Enterprise Web Platform instances on a single server, you can either install the full JBoss distribution onto multiple locations on your filesystem, or you can simply make copies of the **production** configuration. For example, assuming the root of the JBoss distribution was unzipped to `/var/jboss`, you would:

```
$ cd /var/jboss/server
$ cp -r production node1
$ cp -r production node2
```

- **For each node, determine the address to bind sockets to.** When you start JBoss, whether clustered or not, you need to tell JBoss on what address its sockets should listen for traffic. (The

default is **localhost** which is secure but isn't very useful, particularly in a cluster.) So, you need to decide what those addresses will be.

- **Ensure multicast is working.** By default JBoss Enterprise Web Platform uses UDP multicast for most intra-cluster communications. Make sure each server's networking configuration supports multicast and that multicast support is enabled for any switches or routers between your servers. If you are planning to run more than one node on a server, make sure the server's routing table includes a multicast route. See the JGroups documentation at <http://www.jgroups.org> for more on this general area, including information on how to use JGroups' diagnostic tools to confirm that multicast is working.



Note

JBoss Enterprise Web Platform clustering does not require the use of UDP multicast; the Enterprise Web Platform can also be reconfigured to use TCP unicast for intra-cluster communication.

Beyond the above required steps, the following two optional steps are recommended to help ensure that your cluster is properly isolated from other JBoss Enterprise Web Platform clusters that may be running on your network:

- **Pick a unique name for your cluster.** The default name for a JBoss Enterprise Web Platform cluster is **DefaultPartition**. Devise a different name for each cluster in your environment, for example, **QAPartition** or **BobsDevPartition**. The use of "Partition" is not required; it's just a semi-convention. As a small aid to performance try to keep the name short, as it gets included in every message sent around the cluster. We'll cover how to use the name you pick in the next section.
- **Pick a unique multicast address for your cluster.** By default JBoss Enterprise Web Platform uses UDP multicast for most intra-cluster communication. Pick a different multicast address for each cluster you run. Generally a good multicast address is of the form **239.255.x.y**. See <http://www.29west.com/docs/THPM/multicast-address-assignment.html> for a good discussion on multicast address assignment. We'll cover how to use the address you pick in the next section.

See [Section 23.6.2, "Isolating JGroups Channels"](#) for more on isolating clusters.

15.1.2. Launching a JBoss Enterprise Web Platform Cluster

The simplest way to start a JBoss server cluster is to start several JBoss instances on the same local network, using the **-c production** command line option for each instance. Those server instances will detect each other and automatically form a cluster.

Let's look at a few different scenarios for doing this. In each scenario we'll be creating a two node cluster, where the **ServerPeerID** for the first node is **1** and for the second node is **2**. We've decided to call our cluster **DocsPartition** and to use **239.255.100.100** as our multicast address. These scenarios are meant to be illustrative; the use of a two node cluster shouldn't be taken to mean that is the best size for a cluster; it's just that's the simplest way to do the examples.

► Scenario 1: Nodes on Separate Machines

This is the most common production scenario. For this example, assume that:

- The machines are named **node1** and **node2**.
- **node1** has an IP address of **192.168.0.101**, and **node2** has an address of **192.168.0.102**.
- The **ServerPeerID** is **1** for **node1**, and **2** for **node2**.
- On each machine, JBoss is installed in **/var/jboss**.

On node1, to launch JBoss:

```
$ cd /var/jboss/bin
$ ./run.sh -c production -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1
```

On node2, it's the same except for a different **-b** value and ServerPeerID:

```
$ cd /var/jboss/bin
$ ./run.sh -c production -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.102
```

The **-c** switch says to use the **production** config, which includes clustering support. The **-g** switch sets the cluster name. The **-u** switch sets the multicast address that will be used for intra-cluster communication. The **-b** switch sets the address on which sockets will be bound.

► Scenario 2: Two Nodes on a Single, Multihomed, Server

Running multiple nodes on the same machine is a common scenario in a development environment, and is also used in production in combination with Scenario 1. (Running *all* the nodes in a production cluster on a single machine is generally not recommended, since the machine itself becomes a single point of failure.) In this version of the scenario, the machine is multihomed, that is, it has more than one IP address. This allows the binding of each JBoss instance to a different address, preventing port conflicts when the nodes open sockets.

Assume the single machine has the **192.168.0.101** and **192.168.0.102** addresses assigned, and that the two JBoss instances use the same addresses and ServerPeerIDs as in Scenario 1. The difference from Scenario 1 is that we need to be sure each Enterprise Web Platform instance has its own work area. So, instead of using the **production** config, we are going to use the **node1** and **node2** configs we copied from **production** in the previous section.

To launch the first instance, open a console window and:

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101
```

For the second instance, it's the same except for different **-b** and **-c** values and a different ServerPeerID:

```
$ cd /var/jboss/bin
$ ./run.sh -c node2 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.102
```

► Scenario 3: Two Nodes on a Single, Non-Multihomed, Server

This is similar to Scenario 2, but here the machine only has one IP address available. Two processes can't bind sockets to the same address and port, so we'll have to tell JBoss to use different ports for the two instances. This can be done by configuring the **ServiceBindingManager** service by setting the **jboss.service.binding.set** system property.

To launch the first instance, open a console window and:

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.service.binding.set=ports-default
```

For the second instance:

```
$ cd /var/jboss/bin
$ ./run.sh -c node2 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.service.binding.set=ports-01
```

This tells the **ServiceBindingManager** on the first node to use the standard set of ports (for

example, JNDI on port 1099). The second node uses the **ports-01** binding set, which by default for each port has an offset of 100 from the standard port number (for example, JNDI on port 1199). See the **conf/bindingservice.beans/META-INF/bindings-jboss-beans.xml** file for the full **ServiceBindingManager** configuration.

Note that this setup is not advised for production use, due to the increased management complexity that comes with using different ports. But it is a fairly common scenario in development environments where developers want to use clustering but cannot multihome their workstations.



Note

Including **-Djboss.service.binding.set=ports-default** on the command line for **node1** isn't technically necessary, since **ports-default** is the default value. But using a consistent set of command line arguments across all servers is helpful to people less familiar with all the details.

That's it; that's all it takes to get a cluster of JBoss Enterprise Web Platform servers up and running.

15.1.3. Web Application Clustering Quick Start

JBoss Enterprise Web Platform supports clustered web sessions, where a backup copy of each user's **HttpSession** state is stored on one or more nodes in the cluster. In case the primary node handling the session fails or is shut down, any other node in the cluster can handle subsequent requests for the session by accessing the backup copy. Web tier clustering is discussed in detail in [Chapter 21, HTTP Services](#).

There are two aspects to setting up web tier clustering:

- ▶ **Configuring an External Load Balancer.** Web applications require an external load balancer to balance HTTP requests across the cluster of JBoss Enterprise Web Platform instances (see [Section 16.2.2, “External Load Balancer Architecture”](#) for more on why that is). JBoss Enterprise Web Platform itself doesn't act as an HTTP load balancer. So, you will need to set up a hardware or software load balancer. There are many possible load balancer choices, so how to configure one is really beyond the scope of a Quick Start. But see [Section 21.1, “Configuring load balancing using Apache and mod_jk”](#) for details on how to set up the popular mod_jk software load balancer.
- ▶ **Configuring Your Web Application for Clustering.** This aspect involves telling JBoss you want clustering behavior for a particular web app, and it couldn't be simpler. Just add an empty **distributable** element to your application's **web.xml** file:

```
<?xml version="1.0"?>
<web-app 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
    http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
  version="2.5">

  <distributable/>

</web-app>
```

Simply doing that is enough to get the default JBoss Enterprise Web Platform web session clustering behavior, which is appropriate for most applications. See [Section 21.2, “Configuring HTTP session state replication”](#) for more advanced configuration options.

Chapter 16. Clustering Concepts

In the next section, we discuss basic concepts behind JBoss' clustering services. It is helpful that you understand these concepts before reading the rest of the *Clustering Guide*.

16.1. Cluster Definition

A cluster is a set of nodes that communicate with each other and work toward a common goal. In a JBoss Enterprise Web Platform cluster (also known as a *partition*), a node is an JBoss Enterprise Web Platform 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 production** on two Enterprise Web Platform instances on the same network is enough for them to form a cluster — each Enterprise Web Platform starts one or more **Channels** with the same default configuration, so that 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.

On the same Enterprise Web Platform instance, different services can create their own **Channel**. In a standard startup of the Enterprise Web Platform 5 **production** configuration, a core general purpose clustering service known as **HAPartition** creates two channels. If you deploy clustered web applications or a clustered JPA/Hibernate entity cache, additional channels will be created. The channels the Enterprise Web Platform connects can be divided into two broad categories: a general purpose channel used by the **HAPartition** service, and channels created by JBoss Cache for special purpose caching and cluster wide state replication.

So, if you go to two Enterprise Web Platform 5 instances and execute **run -c production**, the channels will discover each other and you'll have a conceptual **cluster**. It is easy to think of this as a two node cluster, but it's important to understand that you really have multiple channels, and hence multiple two node clusters.

On the same network, you may have different sets of servers whose services wish to cluster. [Figure 16.1, “Clusters and server nodes”](#) shows an example network of JBoss server instances divided into three sets, with the third set only having one node. This sort of topology can be set up simply by configuring the Enterprise Web Platform 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 channel configurations and names match while they differ from any other channels on the same network. The Enterprise Web Platform tries to make this as easy as possible, such that servers that are meant to cluster only need to have the same values passed on the command line to the **-g** (partition name) and **-u** (multicast address) startup switches. Different values should be chosen for each set of servers. [Section 23.1, “Configuring a JGroups Channel's Protocol Stack”](#) and [Section 23.6.2, “Isolating JGroups Channels”](#) cover in detail how to configure the Enterprise Web Platform such that desired peers find each other and unwanted peers do not.

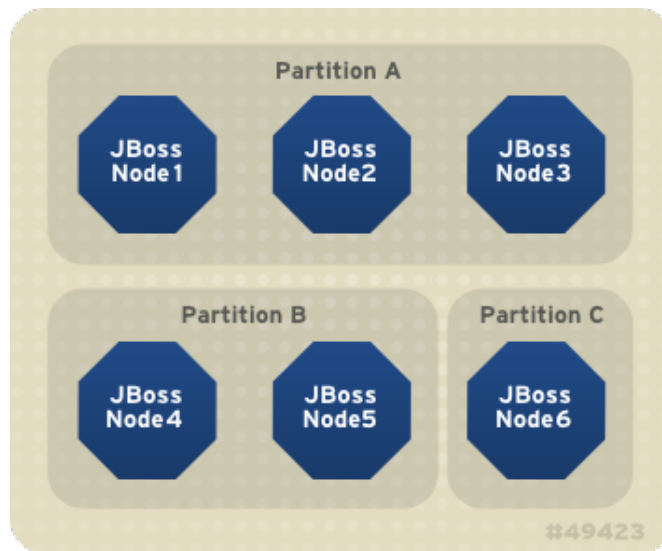


Figure 16.1. Clusters and server nodes

16.2. Service Architectures

The clustering topography defined by the JGroups configuration on each node is of great importance to system administrators. But for most application developers, the greater concern is probably the cluster architecture from a client application's point of view. Two basic clustering architectures are used with JBoss Enterprise Web Platform: client-side interceptors (also known as smart proxies or *stubs*) and external load balancers. Which architecture your application will use will depend on what type of client you have.

16.2.1. Client-side interceptor architecture

Most remote services provided by the JBoss Enterprise Web Platform, including JNDI and RMI, require the client to obtain (that is, to look up and download) a remote proxy object. The proxy object is generated by the server and it implements the business interface of the service. The client then makes local method calls against the proxy object. The proxy automatically routes the call across the network where it is invoked against service objects managed in the server. The proxy object figures out how to find the appropriate server node, marshal call parameters, unmarshal call results, and return the result to the caller client. In a clustered environment, the server-generated proxy object includes an interceptor that understands how to route calls to multiple nodes in the cluster.

The proxy's clustering logic maintains up-to-date knowledge about the cluster. For instance, it knows 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 is not available. As part of handling each service request, if the cluster topology has changed the server node updates the proxy with the latest changes in the cluster. For instance, if a node drops out of the cluster, each proxy is updated with the new topology the next time it connects to any active node in the cluster. All the manipulations done by the proxy's clustering logic are transparent to the client application. The client-side interceptor clustering architecture is illustrated in [Figure 16.2, "The client-side interceptor \(proxy\) architecture for clustering"](#).

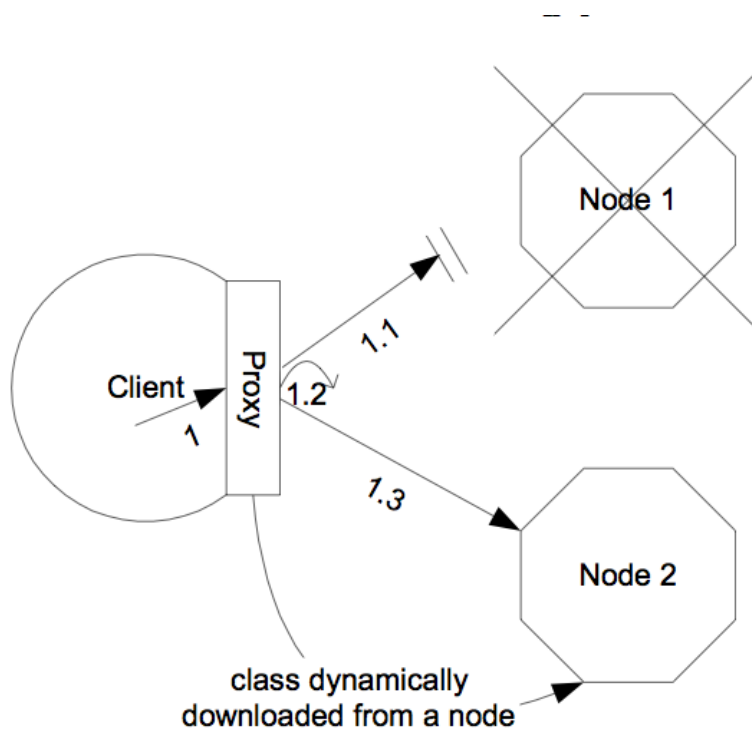


Figure 16.2. The client-side interceptor (proxy) architecture for clustering

16.2.2. External Load Balancer Architecture

The HTTP-based JBoss services do not require the client to download anything. The client (for example, a web browser) sends in requests and receives responses directly over the wire using 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 how to contact the load balancer; it has no knowledge of the JBoss Enterprise Web Platform 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 Enterprise Web Platform instances. It can be implemented either in software or hardware. There are many vendors of hardware load balancers; the **mod_jk** 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 16.3, “The external load balancer architecture for clustering”](#).

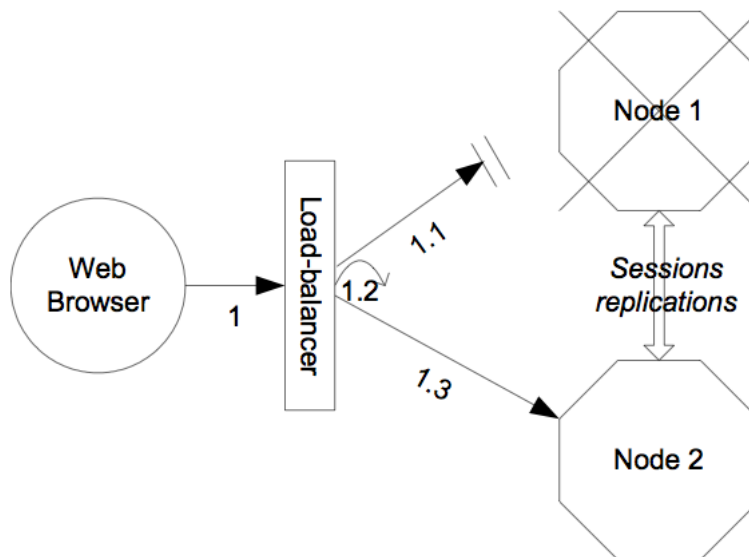


Figure 16.3. The external load balancer architecture for clustering

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.

16.3. Load Balancing Policies

Both the JBoss client-side interceptor (stub) and load balancer use load balancing policies to determine which server node should receive a new request. In this section, let's go over the load balancing policies available in JBoss Enterprise Web Platform.

16.3.1. Client-side interceptor architecture

In JBoss Enterprise Web Platform 5, 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. Each policy has two implementation classes, one meant for use by legacy services that use the legacy detached invoker architecture, and the other meant for services that use AOP-based invocations.

Round-Robin

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. Implemented by **org.jboss.ha.framework.interfaces.RoundRobin** (legacy) and **org.jboss.ha.client.loadbalance.RoundRobin** (AOP).

Random-Robin

For each call the target node is randomly selected from the list. Implemented by **org.jboss.ha.framework.interfaces.RandomRobin** (legacy) and **org.jboss.ha.client.loadbalance.RandomRobin** (AOP).

First Available

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 proxy elects its own

target node independently of the other proxies, so if a particular client downloads two proxies for the same target service, each proxy 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. Implemented by **org.jboss.ha.framework.interfaces.FirstAvailable** (legacy) and **org.jboss.ha.client.loadbalance.aop.FirstAvailable** (AOP).

First Available Identical All Proxies

Has the same behavior as the "First Available" policy but the elected target node is shared by all proxies in the same client-side VM that are associated with the same target service. So if a particular client downloads two proxies for the same target service, each proxy will use the same target. Implemented by **org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies** (legacy) and **org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies** (AOP).

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.

16.3.2. External load balancer architecture

New in JBoss Enterprise Web Platform 5 are a set of **TransactionSticky** load balance policies. These extend the standard policies above to add behavior such that all invocations that occur within the scope of a transaction are routed to the same node (if that node still exists). These are based on the legacy detached invoker architecture, so they are not available for AOP-based services.

Transaction-Sticky Round-Robin

Transaction-sticky variant of Round-Robin. Implemented by **org.jboss.ha.framework.interfaces.TransactionStickyRoundRobin**.

Transaction-Sticky Random-Robin

Transaction-sticky variant of Random-Robin. Implemented by **org.jboss.ha.framework.interfaces.TransactionStickyRandomRobin**.

Transaction-Sticky First Available

Transaction-sticky variant of First Available. Implemented by **org.jboss.ha.framework.interfaces.TransactionStickyFirstAvailable**.

Each of the above is an implementation of a simple interface; users are free to write their own implementations if they need some special behavior. In later sections we'll see how to configure the load balance policies used by different services.

Chapter 17. Clustering Building Blocks

The clustering features in JBoss Enterprise Web Platform are built on top of lower level libraries that provide much of the core functionality.

```
JGroups
|-- JBoss Cache
|   |-- HTTP Session
|   |-- Hibernate
|
|-- HAPartition
|   |-- HA-JNDI
|   |-- HA Singleton
|   |-- Client Proxies
```

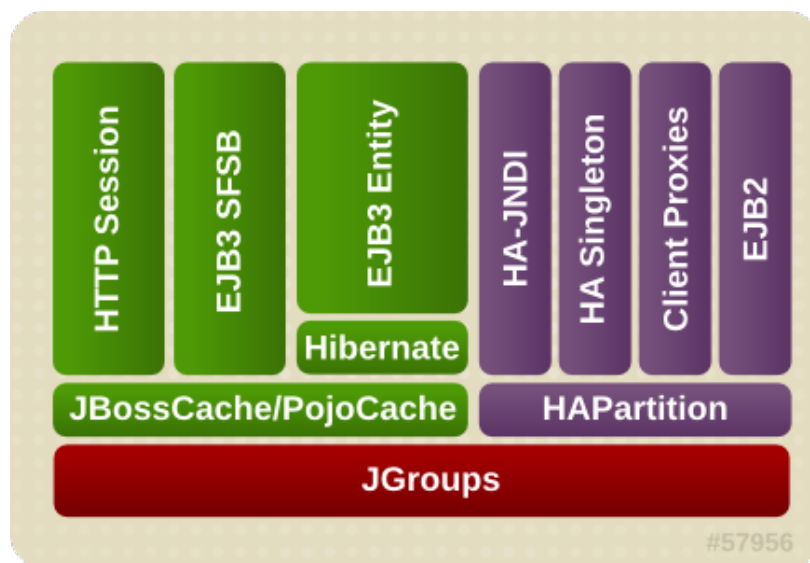


Figure 17.1. The JBoss Enterprise Web Platform clustering architecture

JGroups is a toolkit for reliable point-to-point and point-to-multipoint communication. JGroups is used for all clustering-related communications between nodes in a JBoss Enterprise Web Platform cluster.

JBoss Cache is a highly flexible clustered transactional caching library. Many Enterprise Web Platform clustering services need to cache some state in memory while ensuring that a backup of that state is available on another node, and the data cached on each node in the cluster is consistent. JBoss Cache handles these concerns for most JBoss Enterprise Web Platform clustered services. JBoss Cache uses JGroups to handle its group communication requirements. **POJO Cache** is an extension of the core JBoss Cache that JBoss Enterprise Web Platform uses to support fine-grained replication of clustered web session state. See [Section 17.1.3, “Distributed Caching with JBoss Cache”](#) for more on how JBoss Enterprise Web Platform uses JBoss Cache and POJO Cache.

HAPartition is an adapter on top of a JGroups channel that allows multiple services to use the channel. HAPartition also supports a distributed registry of which HAPartition-based services are running on which cluster members. It provides notifications to interested listeners when the cluster membership changes or the clustered service registry changes. See [Section 17.1.4, “The HAPartition Service”](#) for more details on HAPartition.

The other higher level clustering services make use of JBoss Cache or HAPartition, or, in the case of HA-JNDI, both.

17.1. Group Communication with JGroups

JGroups provides the underlying group communication support for JBoss Enterprise Web Platform

clusters. Services deployed on JBoss Enterprise Web Platform which need group communication with their peers will obtain a JGroups **Channel** and use it to communicate. The **Channel** handles such tasks as managing which nodes are members of the group, detecting node failures, ensuring lossless, first-in-first-out delivery of messages to all group members, and providing flow control to ensure fast message senders cannot overwhelm slow message receivers.

The characteristics of a JGroups **Channel** are determined by the set of *protocols* that compose it. Each protocol handles a single aspect of the overall group communication task; for example the **UDP** protocol handles the details of sending and receiving UDP datagrams. A **Channel** that uses the **UDP** protocol is capable of communicating with UDP unicast and multicast; alternatively one that uses the **TCP** protocol uses TCP unicast for all messages. JGroups supports a wide variety of different protocols (see [Section 23.1, “Configuring a JGroups Channel's Protocol Stack”](#) for details), but the Enterprise Web Platform ships with a default set of channel configurations that should meet most needs.

By default, all JGroups channels of the Enterprise Application Platform use the UDP multicast (an exception to this is a JBoss Messaging channel, which is TCP-based). To change the default multicast type for a server, in `$JBOSS_HOME/bin/` create `run.conf`. Open the file and add the following: `JAVA_OPTS="$JAVA_OPTS -Djboss.default.jgroups.stack=<METHOD>”`.

17.1.1. The Channel Factory Service

It is important to note that the JGroups channels needed by clustering services (for example, a channel used by a distributed **HttpSession** cache) are not configured in detail as part of the consuming service's configuration, and are not directly instantiated by the consuming service. Instead, a new **ChannelFactory** service is used as a registry for named channel configurations and as a factory for **Channel** instances. A service that needs a channel requests the channel from the **ChannelFactory**, passing in the name of the desired configuration.

The **ChannelFactory** service is deployed in the `server/production/deploy/cluster/jgroups-channelfactory.sar`. On startup the ChannelFactory service parses the `server/production/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml` file, which includes various standard JGroups configurations identified by name (for example, UDP or TCP). Services that require a channel access the channel factory and request a channel with a particular named configuration.



Note

If several services request a channel with the same configuration name from the **ChannelFactory**, they are not handed a reference to the same underlying **Channel**. Each receives its own **Channel**, but the channels will have an identical configuration. A logical question is how those channels avoid forming a group with each other if each, for example, is using the same multicast address and port. The answer is that when a consuming service connects its **Channel**, it passes a unique-to-that-service `cluster_name` argument to the `Channel.connect(String cluster_name)` method. The channel uses that `cluster_name` as one of the factors that determine whether a particular message received over the network is intended for that particular channel.

17.1.1.1. Standard Protocol Stack Configurations

The standard protocol stack configurations that ship with Enterprise Web Platform 5 are described below. Note that not all of these are actually used; many are included as a convenience to users who may wish to alter the default server configuration profile. The configuration used in a pristine Enterprise Web Platform 5 **production** configuration is **udp**.

You can add a new stack configuration by adding a new **stack** element to the `server/all/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-`

channelfactory-stacks.xml file. You can alter the behavior of an existing configuration by editing this file. Before doing this though, see the other standard configurations shipped with the Enterprise Web Platform to determine whether they will meet your requirements.



Important

Before you edit a configuration, you should understand which services use that configuration. It is important that any changes you make are appropriate for all affected services. If the change is not appropriate for a particular service, it may be better to create a new configuration and alter some services to use the new configuration.

udp

UDP multicast based stack meant to be shared between different channels. Message bundling is disabled, as it can add latency to synchronous group remote procedure calls (RPCs). Services that only make asynchronous RPCs (for example, JBoss Cache configured for **REPL_ASYNC**) and do so in high volume may be able to improve performance by configuring their cache to use the **udp-async** stack below. Services that only make synchronous RPCs (for example, JBoss Cache configured for **REPL_SYNC** or **INVALIDATION_SYNC**) may be able to improve performance by using the **udp-sync** stack below, which does not include flow control.

udp-async

Same as the default **udp** stack above, except message bundling is enabled in the transport protocol (**enable_bundling=true**). Useful for services that make high-volume asynchronous RPCs (for example, high volume JBoss Cache instances configured for **REPL_ASYNC**) where message bundling may improve performance.

udp-sync

UDP multicast based stack, without flow control and without message bundling. This can be used instead of **udp** if synchronous calls are used and the message volume (rate and size) is not very large. Do not use this configuration if you send messages at a high sustained rate, or you might run out of memory.

tcp

TCP based stack, with flow control and message bundling. TCP stacks are usually used when IP multicasting cannot be used in a network (for example, when routers discard multicast).

tcp-sync

TCP based stack, without flow control and without message bundling. TCP stacks are usually used when IP multicasting cannot be used in a network (for example, when routers discard multicast). This configuration should be used instead of **tcp** above when synchronous calls are used and the message volume (rate and size) is not very large. Do not use this configuration if you send messages at a high sustained rate, or you might run out of memory.

17.1.2. The JGroups Shared Transport

As the number of JGroups-based clustering services has risen over the years, the need to share the resources (particularly sockets and threads) used by these channels became a glaring problem. A pristine Enterprise Web Platform 5 **production** configuration will connect four JGroups channels during startup, and a total of seven or eight will be connected if distributable web applications, clustered EJB3 SFSBs and a clustered JPA or Hibernate second level cache are all used. This many channels can

consume a lot of resources, and can be a real configuration nightmare if the network environment requires configuration to ensure cluster isolation.

Beginning with Enterprise Web Platform 5, JGroups supports sharing of transport protocol instances between channels. A JGroups channel is composed of a stack of individual protocols, each of which is responsible for one aspect of the channel's behavior. A transport protocol is a protocol that is responsible for actually sending messages on the network and receiving them from the network. The resources that are most desirable for sharing (sockets and thread pools) are managed by the transport protocol, so sharing a transport protocol between channels efficiently accomplishes JGroups resource sharing.

To configure a transport protocol for sharing, simply add a `singleton_name="someName"` attribute to the protocol's configuration. All channels whose transport protocol configuration uses the same `singleton_name` value will share their transport. All other protocols in the stack will not be shared.

[Figure 17.2, "Services using a Shared Transport"](#) illustrates four services running in a JVM, each with its own channel. Three of the services are sharing a transport; the fourth is using its own transport.

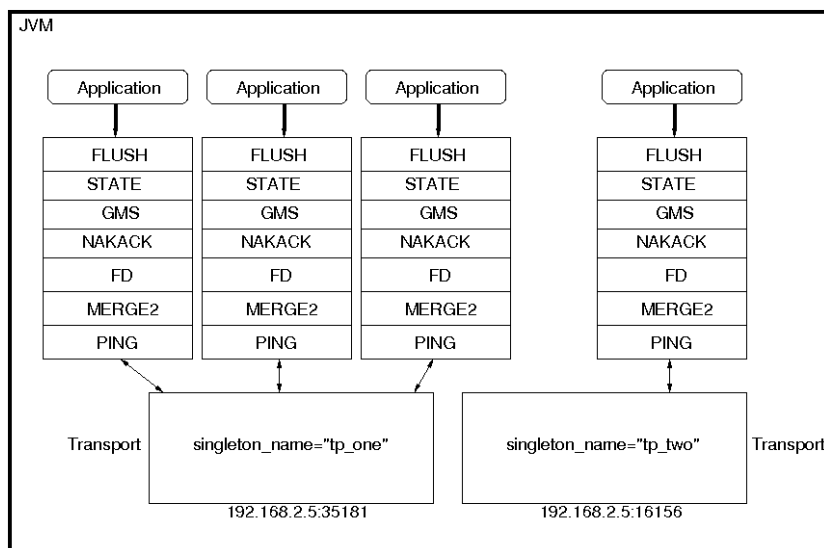


Figure 17.2. Services using a Shared Transport

The protocol stack configurations used by the Enterprise Web Platform 5 **ChannelFactory** all have a `singleton_name` configured. In fact, if you add a stack to the **ChannelFactory** that doesn't include a `singleton_name`, before creating any channels for that stack, the **ChannelFactory** will synthetically create a `singleton_name` by concatenating the stack name to the string `unnamed_`, for example, `unnamed_customStack`.

17.1.3. Distributed Caching with JBoss Cache

JBoss Cache is a fully featured distributed cache framework that can be used in any application server environment or standalone. JBoss Cache provides the underlying distributed caching support used by many of the standard clustered services in a JBoss Enterprise Web Platform cluster, including:

- replication of clustered web application sessions
- clustered caching of JPA and Hibernate entities
- clustered Single Sign-On
- the HA-JNDI replicated tree
- DistributedStateService

Users can also create their own JBoss Cache and POJO Cache instances for custom use by their applications, see [Chapter 24, JBoss Cache Configuration and Deployment](#) for more on this.

17.1.3.1. The JBoss Enterprise Web Platform CacheManager Service

Many of the standard clustered services in JBoss Enterprise Web Platform use JBoss Cache to maintain consistent state across the cluster. Different services (for example, web session clustering or second level caching of JPA/Hibernate entities) use different JBoss Cache instances, with each cache configured to meet the needs of the service that uses it.

In JBoss Enterprise Web Platform 5, the scattered cache deployments use the **CacheManager** service, deployed via the `$JBOSS_HOME/server/$PROFILE/deploy/cluster/jboss-cache-manager.sar`. The **CacheManager** is a factory and registry for JBoss Cache instances. It is configured with a set of named JBoss Cache configurations. Services that require a cache request it by name from the cache manager. The cache manager creates the cache if it does not exist already, and returns it. The cache manager keeps a reference to each cache it has created, so all services that request the same cache configuration name will share the same cache. When a service is done with the cache, it releases it to the cache manager. The cache manager keeps track of how many services are using each cache, and will stop and destroy the cache when all services have released it.

17.1.3.1.1. Standard Cache Configurations

The following standard JBoss Cache configurations ship with JBoss Enterprise Web Platform 5. You can add others to suit your needs, or edit these configurations to adjust cache behavior. Additions or changes are done by editing the `deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml` file (see [Section 24.2.1, “Deployment Via the CacheManager Service”](#) for details). Note however that these configurations are specifically optimized for their intended use, and except as specifically noted in the documentation chapters for each service in this guide, it is not advisable to change them.

standard-session-cache

Standard cache used for web sessions.

field-granularity-session-cache

Standard cache used for FIELD granularity web sessions.

ha-partition

Used by web tier Clustered Single Sign-On, HA-JNDI, Distributed State.

mvcc-entity

A configuration appropriate for JPA/Hibernate entity/collection caching that uses JBoss Cache's MVCC locking (see notes below).

optimistic-entity

A configuration appropriate for JPA/Hibernate entity/collection caching that uses JBoss Cache's optimistic locking (see notes below).

pessimistic-entity

A configuration appropriate for JPA/Hibernate entity/collection caching that uses JBoss Cache's pessimistic locking (see notes below).

mvcc-entity-repeatable

Same as **mvcc-entity** but uses JBoss Cache's **REPEATABLE_READ** isolation level instead of **READ_COMMITTED** (see notes below).

pessimistic-entity-repeatable

Same as **pessimistic-entity** but uses JBoss Cache's **REPEATABLE_READ** isolation level instead of **READ_COMMITTED** (see notes below).

local-query

A configuration appropriate for JPA/Hibernate query result caching. Does not replicate query results. *Do not* store the timestamp data Hibernate uses to verify validity of query results in this cache.

replicated-query

A configuration appropriate for JPA/Hibernate query result caching. Replicates query results. **DO NOT** store the timestamp data Hibernate uses to verify validity of query result in this cache.

timestamps-cache

A configuration appropriate for the timestamp data cached as part of JPA/Hibernate query result caching. A replicated timestamp cache is required if query result caching is used, even if the query results themselves use a non-replicating cache like **local-query**.

mvcc-shared

A configuration appropriate for a cache that's shared for JPA/Hibernate entity, collection, query result and timestamp caching. Not an advised configuration, since it requires cache mode **REPL_SYNC**, which is the least efficient mode. Also requires a full state transfer at startup, which can be expensive. Retained for backwards compatibility. Uses JBoss Cache's MVCC locking.

optimistic-shared

A configuration appropriate for a cache that's shared for JPA/Hibernate entity, collection, query result and timestamp caching. Not an advised configuration, since it requires cache mode **REPL_SYNC**, which is the least efficient mode. Also requires a full state transfer at startup, which can be expensive. Maintained for backwards compatibility reasons, as a shared cache was the only option in JBoss 4. Uses JBoss Cache's optimistic locking.

pessimistic-shared

A configuration appropriate for a cache that's shared for JPA/Hibernate entity, collection, query result and timestamp caching. Not an advised configuration, since it requires cache mode **REPL_SYNC**, which is the least efficient mode. Also requires a full state transfer at startup, which can be expensive. Maintained for backwards compatibility reasons, as a shared cache was the only option in JBoss 4. Uses JBoss Cache's pessimistic locking.

mvcc-shared-repeatable

Same as **mvcc-shared** but uses JBoss Cache's **REPEATABLE_READ** isolation level instead of **READ_COMMITTED** (see notes below).

pessimistic-shared-repeatable

Same as **pessimistic-shared** but uses JBoss Cache's **REPEATABLE_READ** isolation level instead of **READ_COMMITTED** (see notes below).

**Note**

For more on JBoss Cache's locking schemes, see [Section 24.1.4, "Concurrent Access"](#))

**Note**

For JPA/Hibernate second level caching, **REPEATABLE_READ** is only useful if the application evicts or clears entities from the Hibernate Entity Manager Session and then expects to repeatably re-read them in the same transaction. Otherwise, the session's internal cache provides a repeatable-read semantic.

17.1.3.1.2. Cache Configuration Aliases

The **CacheManager** also supports aliasing of caches; that is, allowing caches registered under one name to be looked up under a different name. Aliasing is useful for sharing caches between services whose configuration may specify different cache configuration names.

Aliases can be configured by editing the **CacheManager** bean in the **jboss-cache-manager-jboss-beans.xml** file. The following redacted configuration shows the standard aliases in Enterprise Web Platform 5:

```
<<<<<<
<bean name="CacheManager" class="org.jboss.ha.cachemanager.CacheManager">

    . . .

    <!-- Aliases for cache names. Allows caches to be shared across
         services that may expect different cache configuration names. -->
    <property name="configAliases">
        <map keyClass="java.lang.String" valueClass="java.lang.String">
            <!-- Use the HAPartition cache for ClusteredSSO caching -->
            <entry>
                <key>clustered-sso</key>
                <value>ha-partition</value>
            </entry>
        </map>
    </property>

    . . .

</bean>
```

17.1.4. The HAPartition Service

HAPartition is a general purpose service used for a variety of tasks in Enterprise Web Platform clustering. At its core, it is an abstraction built on top of a JGroups **Channel** that provides support for making and receiving RPC invocations from one or more cluster members. **HAPartition** allows services that use it to share a single **Channel** and multiplex RPC invocations over it, eliminating the configuration complexity and runtime overhead of having each service create its own **Channel**.

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, farming, HA-JNDI and HA singletons. Custom services can also make use of **HAPartition**.

The following snippet shows the **HAPartition** service definition packaged with the standard JBoss Enterprise Web Platform distribution. This configuration can be found in the

server/\$PROFILE/deploy/cluster/hapartition-jboss-beans.xml file.

```
<bean name="HAPartitionCacheHandler"
class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
  <property name="cacheManager"><inject bean="CacheManager"/></property>
  <property name="cacheConfigName">ha-partition</property>
</bean>
<bean name="HAPartition" class="org.jboss.ha.framework.server.ClusterPartition">
  <depends>jboss:service=Naming</depends>
  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX

(name="jboss:service=HAPartition,partition=${jboss.partition.name:DefaultPartition}
", exposedInterface=org.jboss.ha.framework.server.ClusterPartitionMBean.class,
registerDirectly=true)</annotation>

<!-- ClusterPartition requires a Cache for state management -->

<property name="cacheHandler"><inject bean="HAPartitionCacheHandler"/></property>

<!-- Name of the partition being built -->

<property
name="partitionName">${jboss.partition.name:DefaultPartition}</property>

<!-- The address used to determine the node name -->

<property name="nodeAddress">${jboss.bind.address}</property>

<!-- Max time (in ms) to wait for state transfer to complete. Increase for large
states -->

<property name="stateTransferTimeout">30000</property>

<!-- Max time (in ms) to wait for RPC calls to complete. -->

<property name="methodCallTimeout">60000</property>

<!-- Optionally provide a thread source to allow async connect of our channel -->

<property name="threadPool"><inject
bean="jboss:system:service=ThreadPool"/></property>
<property name="distributedStateImpl">
  <bean name="DistributedState"
class="org.jboss.ha.framework.server.DistributedStateImpl">
  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX

(name="jboss:service=DistributedState,partitionName=${jboss.partition.name:DefaultP
artition}",
exposedInterface=org.jboss.ha.framework.server.DistributedStateImplMBean.class,
registerDirectly=true)</annotation>
  <property name="cacheHandler"><inject
bean="HAPartitionCacheHandler"/></property>
</bean>
</property>
</bean>
```

Much of the above is generic; below we'll touch on the key points relevant to end users. There are two beans defined above, the **HAPartitionCacheHandler** and the **HAPartition** itself.

The **HAPartition** bean itself exposes the following configuration properties:

partitionName

An optional attribute to specify the name of the cluster. Its default value is

DefaultPartition. Use the **-g** (or **--partition**) command line switch to set this value at server startup.



The `partitionName` property on the `MCBean:ServerConfig Profile Service Component`

If you use the `partitionName` property on the `MCBean:ServerConfig Profile Service` component, the system returns a null value for the property. Use the `PartionName` from the `MCBean:HAPartition` managed component to obtain the correct value.

nodeAddress

This attribute is not used and can be ignored.

stateTransferTimeout

Specifies the timeout in milliseconds for initial application state transfer. State transfer refers to the process of obtaining a serialized copy of initial application state from other already-running cluster members at service startup. Its default value is **30000**.

methodCallTimeout

Specifies the timeout in milliseconds for obtaining responses to group remote procedure calls from other cluster members. Its default value is **60000**.

The **HAPartitionCacheHandler** is a small utility service that helps the **HAPartition** integrate with JBoss Cache (see [Section 17.1.3.1, “The JBoss Enterprise Web Platform CacheManager Service”](#)).

HAPartition exposes a child service called **DistributedState** that uses JBoss Cache; the **HAPartitionCacheHandler** helps ensure consistent configuration between the JGroups **Channel** used by **DistributedState**'s cache and the one used directly by **HAPartition**.

cacheConfigName

The name of the JBoss Cache configuration to use for the **HAPartition**-related cache. Indirectly, this also specifies the name of the JGroups protocol stack configuration **HAPartition** should use. See [Section 24.1.5, “JGroups Integration”](#) for more on how the JGroups protocol stack is configured.

In order for nodes to form a cluster, they must have the exact same **partitionName** and the **HAPartitionCacheHandler**'s **cacheConfigName** must specify an identical JBoss Cache configuration. Changes in either element on some but not all nodes would prevent proper clustering behavior.

You can view the current cluster information by pointing your browser to the JMX console of any JBoss instance in the cluster (that is, **http://hostname:8080/jmx-console/**) and then clicking on the **jboss:service=HAPartition,partition=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.

17.1.4.1. DistributedReplicantManager Service

The **DistributedReplicantManager** (DRM) service is a component of the **HAPartition** service made available to **HAPartition** users via the

HAPartition.getDistributedReplicantManager() method. Generally speaking, JBoss Enterprise Web Platform users will not directly make use of the DRM; we discuss it here as an aid to those who want a deeper understanding of how Enterprise Web Platform clustering internals work.

The DRM is a distributed registry that allows **HAPartition** users to register objects under a given key, making the objects registered under that key by various cluster members available to callers. The DRM also provides a notification mechanism so that interested listeners can be notified when the contents of the registry changes.

There are two main uses for the DRM in JBoss Enterprise Web Platform:

Clustered Smart Proxies

Here the keys are the names of the various services that need a clustered smart proxy (see [Section 16.2.1, “Client-side interceptor architecture”](#)). The value object each node stores in the DRM is known as a *target*. This is used by the smart proxy's transport layer to contact the node (for example, an RMI stub, an HTTP URL or a JBoss Remoting **InvokerLocator**). The factory that builds clustered smart proxies accesses the DRM to get the set of targets that should be injected into the proxy to allow it to communicate with all the nodes in a cluster.

HASingleton

Here the keys are the names of the various services that need to function as High Availability Singletons. The value object each node stores in the DRM is simply a string that acts as a token to indicate that the node has the service deployed, and thus is a candidate to become the "master" node for the **HASingleton** service.

In both cases, the key under which objects are registered identifies a particular clustered service. It is useful to understand that every node in a cluster need not register an object under every key. Only services that are deployed on a particular node will register something under that service's key, and services don't have to be deployed homogeneously across the cluster. The DRM is thus useful as a mechanism for understanding a service's "topology" around the cluster — which nodes have the service deployed.

17.1.4.2. Custom Use of HAPartition

Custom services can also use make use of **HAPartition** to handle interactions with the cluster.

Generally the easiest way to do this is to extend the

org.jboss.ha.framework.server.HAServiceImpl base class, or the

org.jboss.ha.jmx.HAServiceMBeanSupport class if JMX registration and notification support are desired.

Chapter 18. Clustered JNDI Services

The Java Naming and Directory Interface (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 a HA-JNDI naming Context is connected to the HA-JNDI service on a particular JBoss Enterprise Web Platform instance, and that service fails or is shut down, the HA-JNDI client can transparently fail over to another Enterprise Web Platform instance.
- Load balancing of naming operations. A 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. A 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 JNDI so the client can look up 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.

18.1. How it works

The JBoss client-side HA-JNDI naming Context is based on the client-side interceptor architecture (see [Chapter 15, Introduction and Quick Start](#)). 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 [Section 18.2, “Client configuration”](#). 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 each node is able to find objects bound into the local JNDI context tree, and is also able to make a cluster-wide RPC to find objects bound in the local tree on any other node. An application can bind its objects to either tree, although in practice most objects are bound into the local JNDI context 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. So, homes 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 service 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 service 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. A lookup performed 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 under the same name (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

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 is higher that a lookup will hit a HA-JNDI server that does not own this binding 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.



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.

18.2. Client configuration

Configuring a client to use HA-JNDI is a matter of ensuring the correct set of naming environment properties are available when a new **InitialContext** is created. How this is done varies depending on whether the client is running inside JBoss Enterprise Web Platform itself or is in another VM.

18.2.1. For clients running inside the Enterprise Web Platform

If you want to access HA-JNDI from inside the Enterprise Web Platform, you must explicitly configure your **InitialContext** by passing in JNDI properties to the constructor. 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");
// HA-JNDI is listening on the address passed to JBoss via -b
String bindAddress = System.getProperty("jboss.bind.address", "localhost");
p.put(Context.PROVIDER_URL, bindAddress + ":1100"); // HA-JNDI address and port.
return new InitialContext(p);
```

The `Context.PROVIDER_URL` property points to the HA-JNDI service configured in the **deploy/cluster/ha-jndi-jboss-beans.xml** file (see [Section 18.3, “JBoss configuration”](#)). By default this service listens on the interface named via the **jboss.bind.address** system property, which itself is set to whatever value you assign to the **-b** command line option when you start JBoss Enterprise Web Platform (or **localhost** if not specified). The above code shows an example of accessing this property.

However, this does not work in all cases, especially when running several JBoss Enterprise Web Platform instances on the same machine and bound to the same IP address, but configured to use different ports. A safer method is to not specify the `Context.PROVIDER_URL` but instead allow the **InitialContext** to statically find the in-VM HA-JNDI by specifying the **jnp.partitionName** 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");
// HA-JNDI is registered under the partition name passed to JBoss via -g
String partitionName = System.getProperty("jboss.partition.name",
    "DefaultPartition");
p.put("jnp.partitionName", partitionName);
return new InitialContext(p);
```

This example uses the **jboss.partition.name** system property to identify the partition with which the HA-JNDI service works. This system property is set to whatever value you assign to the **-g** command line option when you start JBoss Enterprise Web Platform (or **DefaultPartition** if not specified).

Do not attempt to simplify things by placing a **jndi.properties** file in your deployment or by editing the Enterprise Web Platform's **conf/jndi.properties** file. Doing either will almost certainly break things for your application and quite possibly across the 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.

18.2.1.1. Accessing HA-JNDI Resources from WARs -- Environment Naming Context

If your HA-JNDI client is a servlet, the least intrusive way to configure the lookup of resources is to bind the resources to the environment naming context of the bean or webapp performing the lookup. The binding can then be configured to use HA-JNDI instead of a local mapping.



Why do this programmatically and not just put this in a **jndi.properties** file?

The JBoss Enterprise Web Platform's internal naming environment is controlled by the **conf/jndi.properties** file, which should not be edited.

No other **jndi.properties** file should be deployed inside the Enterprise Web Platform. If another **jndi.properties** file is deployed, it may be found on the classpath when it is not intended for use and disrupt the internal operation of the server.

18.2.1.2. How can I tell if things are being bound into HA-JNDI that shouldn't be?

Go into the the jmx-console and execute the **list** operation on the **jboss:service=JNDIView** mbean. Towards the bottom of the results, the contents of the "HA-JNDI Namespace" are listed. Typically this will be empty; if any of your own deployments are shown there and you didn't explicitly bind them there, there's probably an improper **jndi.properties** file on the classpath.

18.2.2. For clients running outside the Enterprise Web Platform

The JNDI client needs to be aware of the HA-JNDI cluster. You can pass a list of JNDI servers (that is, 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 18.3, "JBoss configuration"](#) for how to configure the servers and ports).

```
java.naming.provider.url=server1:1100, server2:1100, server3:1100, server4:1100
```

When initializing, 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; once the HA-JNDI stub is downloaded, the stub will include information on all the available servers. A good practice is to include a set of servers such that you are certain that at least one of those in the list will be available.

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 [Section 18.3, "JBoss configuration"](#) for 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

A list of IP addresses and port numbers for HA-JNDI provider nodes in the cluster. The client tries each provider and uses the first one that responds.

jnp.disableDiscovery

When **true**, disables the automatic discovery feature. The default value is **false**.

jnp.partitionName

In an environment running multiple HA-JNDI services bound to distinct clusters or partitions, this property ensures that your client only accepts automatic discovery responses from servers in the desired partition. If **jnp.disableDiscovery** is set to **true**, this property is not used. By default, this property is not set, and automatic discovery selects the first HA-JNDI server that responds, regardless of the cluster partition name.

jnp.discoveryTimeout

The time in milliseconds that the context will wait for a response to its automatic discovery packet. The default value is **5000**.

jnp.discoveryGroup

Determines which multicast group address is used for automatic discovery. The default value is **230.0.0.4**. This value must match the value of **AutoDiscoveryAddress** configured on the server-side HA-JNDI service. By default this service listens on the address specified with the **-u** startup switch, so if **-u** is used on the server side, as recommended, **jnp.discoveryGroup** must be configured on the client side.

jnp.discoveryPort

Determines the multicast port used for automatic discovery. The default value is **1102**. This value must match the value of **AutoDiscoveryPort** on the server-side HA-JNDI service.

jnp.discoveryTTL

Specifies the TTL (time to live) for automatic discovery IP multicast packets. This value represents the number of network hops a multicast packet can make before networking equipment should drop the packet.

18.3. JBoss configuration

The **hajndi-jboss-beans.xml** file in the **JBOSS_HOME/server/production/deploy/cluster** directory includes the following bean to enable HA-JNDI services:

```

<bean name="HAJNDI" class="org.jboss.ha.jndi.HANamingService">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HAJNDI",

exposedInterface=org.jboss.ha.jndi.HANamingServiceMBean.class)</annotation>

    <!-- The partition used for group RPCs to find locally bound objects on
other nodes -->
    <property name="HAPartition"><inject bean="HAPartition"/></property>

    <!-- Handler for the replicated tree -->
    <property name="distributedTreeManager">
        <bean class="org.jboss.ha.jndi.impl.jbc.JBossCacheDistributedTreeManager">
            <property name="cacheHandler"><inject
bean="HAPartitionCacheHandler"/></property>
        </bean>
    </property>

    <property name="localNamingInstance">
        <inject bean="jboss:service=NamingBeanImpl" property="namingInstance"/>
    </property>

    <!-- The thread pool used to control the bootstrap and auto discovery lookups
-->
    <property name="lookupPool"><inject
bean="jboss.system:service=ThreadPool"/></property>

    <!-- Bind address of bootstrap endpoint -->
    <property name="bindAddress">${jboss.bind.address}</property>
    <!-- Port on which the HA-JNDI stub is made available -->
    <property name="port">
        <!-- Get the port from the ServiceBindingManager -->
        <value-factory bean="ServiceBindingManager" method="getIntBinding">
            <parameter>jboss:service=HAJNDI</parameter>
            <parameter>Port</parameter>
        </value-factory>
    </property>

    <!-- Bind address of the HA-JNDI RMI endpoint -->
    <property name="rmiBindAddress">${jboss.bind.address}</property>

    <!-- RmiPort to be used by the HA-JNDI service once bound. 0 = ephemeral. -
->
    <property name="rmiPort">
        <!-- Get the port from the ServiceBindingManager -->
        <value-factory bean="ServiceBindingManager" method="getIntBinding">
            <parameter>jboss:service=HAJNDI</parameter>
            <parameter>RmiPort</parameter>
        </value-factory>
    </property>

    <!-- Accept backlog of the bootstrap socket -->
    <property name="backlog">50</property>

    <!-- A flag to disable the auto discovery via multicast -->
    <property name="discoveryDisabled">false</property>
    <!-- Set the auto-discovery bootstrap multicast bind address. If not
specified and a BindAddress is specified, the BindAddress will be used. -->
    <property name="autoDiscoveryBindAddress">${jboss.bind.address}</property>
    <!-- Multicast Address and group port used for auto-discovery -->
    <property
name="autoDiscoveryAddress">${jboss.partition.udpGroup:230.0.0.4}</property>
    <property name="autoDiscoveryGroup">1102</property>

```

```

<!-- The TTL (time-to-live) for autodiscovery IP multicast packets -->
<property name="autoDiscoveryTTL">16</property>

<!-- The load balancing policy for HA-JNDI -->
<property
name="loadBalancePolicy">org.jboss.ha.framework.interfaces.RoundRobin</property>

<!-- Client socket factory to be used for client-server
      RMI invocations during JNDI queries
<property name="clientSocketFactory">custom</property>
-->
<!-- Server socket factory to be used for client-server
      RMI invocations during JNDI queries
<property name="serverSocketFactory">custom</property>
-->
</bean>

```

You can see that this bean has a number of other services injected into different properties:

HAPartition

The core clustering service used to manage HA-JNDI's clustered proxies and to make the group RPCs that find locally-bound objects on other nodes. See [Section 17.1.4, "The HAPartition Service"](#) for more details.

distributedTreeManager

A handler for the replicated tree. The standard handler uses JBoss Cache to manage the replicated tree. The cache instance is retrieved by the injected **HAPartitionCacheHandler** bean. See [Section 17.1.4, "The HAPartition Service"](#) for more details.

localNamingInstance

A reference to the local JNDI service.

lookupPool

The thread pool used to provide threads to handle the bootstrap and automatic discovery lookups.

Besides the above dependency injected services, the available configuration attributes for the HA-JNDI bean are as follows:

bindAddress

The address to which the HA-JNDI server binds to listen for naming proxy requests from JNP clients. The default value is the value of the **jboss.bind.address** system property, if set with the **-b** switch at server startup.

port

Specifies the port to which HA-JNDI binds to listen for naming proxy download requests from JNP clients. The default value is obtained from the **ServiceBindingManager** bean configured in **conf/bootstrap/bindings.xml**. The default value is **100**.

backlog

Specifies the maximum queue length for incoming connection indications for the TCP server socket on which the service listens for naming proxy download requests from JNP clients. The default value is **50**.

rmiBindAddress

Specifies the address to which the HA-JNDI server binds to listen for RMI requests from naming proxies. The default value is the value of the **jboss.bind.address** system property, if set with the **-b** switch on server startup. If the property is not set, the default value is **localhost**.

rmiPort

Specifies the port to which the server binds to communicate with the downloaded stub. The value is obtained from the **ServiceBindingManager** bean configured in **conf/bootstrap/bindings.xml**. The default value is **1101**. If no value is set, the operating system automatically assigns a port.

discoveryDisabled

A Boolean flag that disables configuration of the automatic discovery multicast listener. The default value is **false**.

autoDiscoveryAddress

Specifies the multicast address on which to listen for JNDI automatic discovery. The default value is the value of the **jboss.partition.udpGroup** system property, if set with the **-u** switch on server startup; otherwise, the default is **230.0.0.4**.

autoDiscoveryGroup

Specifies the port to listen on for multicast JNDI automatic discovery packets. The default value is **1102**.

autoDiscoveryBindAddress

Sets the interface on which HA-JNDI listens for auto-discovery request packets. If this attribute is not specified and a **bindAddress** is specified, the **bindAddress** will be used.

autoDiscoveryTTL

Specifies the time-to-live (TTL) for automatic discovery IP multicast packets. This value represents the number of network hops that a multicast packet is allowed to make before networking equipment drops the packet.

loadBalancePolicy

Specifies the class name of the **LoadBalancePolicy** implementation to be included in the client proxy.

clientSocketFactory

An optional attribute that specifies the fully-qualified class name of the **java.rmi.server.RMIServerSocketFactory** used to create client sockets. The default value is **null**.

serverSocketFactory

An optional attribute that specifies the fully qualified class name of the **java.rmi.server.RMIServerSocketFactory** used to create server sockets. The default value is **null**.

18.3.1. Adding a Second HA-JNDI Service

It is possible to start several HA-JNDI services that use different **HAPartitions**. This can be used, for example, if a node is part of many logical clusters. In this case, make sure that you set a different port or IP address for each service. For instance, if you wanted to hook up HA-JNDI to the example cluster you set up and change the binding port, the bean descriptor would look as follows (properties that do not vary from the standard deployments are omitted):

```
<-- Cache Handler for secondary HAPartition -->
<bean name="SecondaryHAPartitionCacheHandler"
      class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
  <property name="cacheManager"><inject bean="CacheManager"/></property>
  <property name="cacheConfigName">secondary-ha-partition</property>
</bean>

<-- The secondary HAPartition -->
<bean name="SecondaryHAPartition"
      class="org.jboss.ha.framework.server.ClusterPartition">

  <depends>jboss:service=Naming</depends>

  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HAPartition,partition=SecondaryPartition",
     exposedInterface=org.jboss.ha.framework.server.ClusterPartitionMBean.class,
     registerDirectly=true)</annotation>

  <property name="cacheHandler"><inject
bean="SecondaryHAPartitionCacheHandler"/></property>

  <property name="partitionName">SecondaryPartition</property>

  ....
</bean>

<bean name="MySpecialPartitionHAJNDI"
      class="org.jboss.ha.jndi.HANamingService">

  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HAJNDI,partitionName=SecondaryPartition",
     exposedInterface=org.jboss.ha.jndi.HANamingServiceMBean.class)</annotation>

  <property name="HAPartition"><inject
bean="SecondaryHAPartition"/></property>

  <property name="distributedTreeManager">
    <bean class="org.jboss.ha.jndi.impl.jbc.JBossCacheDistributedTreeManager">
      <property name="cacheHandler"><inject
bean="SecondaryHAPartitionPartitionCacheHandler"/></property>
    </bean>
  </property>

  <property name="port">56789</property>

  <property name="rmiPort">56790</property>

  <property name="autoDiscoveryGroup">56791</property>

  .....
</bean>
```

Chapter 19. Clustered Session EJBs



Remote EJB connectivity is Not Supported

JBoss Enterprise Web Platform does not support remote EJB connectivity. Further, other EJB support will be removed in versions of JBoss Enterprise Web Platform based on JBoss Application Server 6. If you require this functionality, use JBoss Enterprise Application Platform.

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 the same as the client for the non-clustered version of the session bean, except for some minor changes. No code change or re-compilation is needed on the client side. This chapter shows you how to configure clustered session beans in EJB 3.0 and EJB 2.x server applications respectively.

19.1. Stateless Session Bean in EJB 3.0

Clustering stateless session beans is the simplest case since no state is involved. Calls can be load balanced to any participating node (that is, any node that has this specific bean deployed) of the cluster.

To cluster a stateless session bean in EJB 3.0, simply annotate the bean class with the **@Clustered** annotation. This annotation contains optional parameters for overriding both the load balance policy and partition to use.

```
public @interface Clustered
{
    String partition() default "${jboss.partition.name:DefaultPartition}";
    String loadBalancePolicy() default "LoadBalancePolicy";
}
```

partition

Specifies the name of the cluster the bean participates in. While the **@Clustered** annotation lets you override the default partition, **DefaultPartition**, for an individual bean, you can override this for all beans using the **jboss.partition.name** system property.

loadBalancePolicy

Defines the name of a class implementing **org.jboss.ha.client.loadbalance.LoadBalancePolicy**, indicating how the bean stub should balance calls made on the nodes of the cluster. The default value, **LoadBalancePolicy** is a special token indicating the default policy for the session bean type. For stateless session beans, the default policy is **org.jboss.ha.client.loadbalance.RoundRobin**. You can override the default value using your own implementation, or choose one from the list of available policies:

org.jboss.ha.client.loadbalance.RoundRobin

Starting with a random target, always favors the next available target in the list, ensuring maximum load balancing always occurs.

org.jboss.ha.client.loadbalance.RandomRobin

Randomly selects its target without any consideration to previously selected targets.

org.jboss.ha.client.loadbalance.aop.FirstAvailable

Once a target is chosen, always favors that same target; i.e. no further load balancing occurs. Useful in cases where "sticky session" behavior is desired, e.g. stateful

session beans.

org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies

Similar to **FirstAvailable**, except that the favored target is shared across all proxies.

The following 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
    }
}
```

Rather than using the **@Clustered** annotation, you can also enable clustering for a session bean 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>
```



Note

The **<clustered>true</clustered>** element is an alias for the **<container-name>Clustered Stateless SessionBean</container-name>** element in the **conf/standardjboss.xml** file.

In the bean configuration, only the **<clustered>** element is necessary to indicate that the bean needs to support clustering features. The default values for the optional **<cluster-config>** elements match those of the corresponding properties from the **@Clustered** annotation.

19.2. Stateful Session Beans in EJB 3.0

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 is replicated and synchronized across the cluster each time the state of a bean changes.

19.2.1. The EJB application configuration

To cluster stateful session beans in EJB 3.0, you need to tag the bean implementation class with the **@Clustered** annotation, just as we did with the EJB 3.0 stateless session bean earlier. In contrast to stateless session beans, stateful session bean method invocations are load balanced using **org.jboss.ha.client.loadbalance.aop.FirstAvailable** policy, by default. Using this policy, methods invocations will stick to a randomly chosen node.

The **@org.jboss.ejb3.annotation.CacheConfig** annotation can also be applied to the bean to override the default caching behavior. The definition of the **@CacheConfig** annotation is:

```
public @interface CacheConfig
{
    String name() default "";
    int maxSize() default 10000;
    long idleTimeoutSeconds() default 300;
    boolean replicationIsPassivation() default true;
    long removalTimeoutSeconds() default 0;
}
```

name

Specifies the name of a cache configuration registered with the **CacheManager** service discussed in [Section 19.2.3, "CacheManager service configuration"](#). By default, the **sf-sb-cache** configuration will be used.

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 maximum period of time a bean can go unused before the cache should passivate it (regardless of whether **maxSize** beans are cached).

removalTimeoutSeconds

Specifies the maximum 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 this is set to **true**, since replication involves serializing the bean, and preparing for and recovering from serialization is a common reason for implementing the callback methods.

The following 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 alternatively be omitted and the clustering configuration instead applied to **jboss.xml** like so:

```

<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>NonAnnotationStateful</ejb-name>
      <clustered>true</clustered>
      <cache-config>
        <cache-max-size>5000</cache-max-size>
        <remove-timeout-seconds>18000</remove-timeout-seconds>
      </cache-config>
    </session>
  </enterprise-beans>
</jboss>

```

19.2.2. Optimize state replication

As the replication process is a costly operation, you can optimise this behaviour by optionally implementing the **org.jboss.ejb3.cache.Optimized** interface in your bean class:

```

public interface Optimized
{
    boolean isModified();
}

```

Before replicating your bean, the container will check whether your bean implements the **Optimized** interface. If this is the case, the container calls the **isModified()** method and will only replicate 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 replication would not occur.

19.2.3. CacheManager service configuration

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The **CacheManager** service, described in [Section 17.1.3.1, "The JBoss Enterprise Web Platform CacheManager Service"](#) is both a factory and registry of JBoss Cache instances. By default, stateful session beans use the **sfsb-cache** configuration from the **CacheManager**, defined as follows:

```

<bean name="StandardSFSBCacheConfig" class="org.jboss.cache.config.Configuration">

  <!-- No transaction manager lookup -->

  <!-- Name of cluster. Needs to be the same for all members -->
  <property name="clusterName">${jboss.partition.name:DefaultPartition}-
SFSBCache</property>
  <!--
    Use a UDP (multicast) based stack. Need JGroups flow control (FC)
    because we are using asynchronous replication.
  -->
  <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
  <property name="fetchInMemoryState">true</property>

  <property name="nodeLockingScheme">PESSIMISTIC</property>
  <property name="isolationLevel">REPEATABLE_READ</property>
  <property name="useLockStriping">false</property>
  <property name="cacheMode">REPL_ASYNC</property>

  <!--
    Number of milliseconds to wait until all responses for a
    synchronous call have been received. Make this longer
    than lockAcquisitionTimeout.
  -->
  <property name="syncReplTimeout">17500</property>
  <!-- Max number of milliseconds to wait for a lock acquisition -->
  <property name="lockAcquisitionTimeout">15000</property>
  <!-- The max amount of time (in milliseconds) we wait until the
  state (ie. the contents of the cache) are retrieved from
  existing members at startup. -->
  <property name="stateRetrievalTimeout">60000</property>

  <!--
    SFSBs use region-based marshalling to provide for partial state
    transfer during deployment/undeployment.
  -->
  <property name="useRegionBasedMarshalling">false</property>
  <!-- Must match the value of "useRegionBasedMarshalling" -->
  <property name="inactiveOnStartup">false</property>

  <!-- Disable asynchronous RPC marshalling/sending -->
  <property name="serializationExecutorPoolSize">0</property>
  <!-- We have no asynchronous notification listeners -->
  <property name="listenerAsyncPoolSize">0</property>

  <property name="exposeManagementStatistics">true</property>

  <property name="buddyReplicationConfig">
    <bean class="org.jboss.cache.config.BuddyReplicationConfig">

      <!-- Just set to true to turn on buddy replication -->
      <property name="enabled">false</property>

      <!--
        A way to specify a preferred replication group. We try
        and pick a buddy who shares the same pool name (falling
        back to other buddies if not available).
      -->
      <property name="buddyPoolName">default</property>

      <property name="buddyCommunicationTimeout">17500</property>

      <!-- Do not change these -->
      <property name="autoDataGravitation">false</property>
    </bean>
  </property>

```

```

<property name="dataGravitationRemoveOnFind">true</property>
<property name="dataGravitationSearchBackupTrees">true</property>

<property name="buddyLocatorConfig">
  <bean
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
    <!-- The number of backup nodes we maintain -->
    <property name="numBuddies">1</property>
    <!-- Means that each node will *try* to select a buddy on
         a different physical host. If not able to do so
         though, it will fall back to colocated nodes. -->
    <property name="ignoreColocatedBuddies">true</property>
  </bean>
</property>
</bean>
</property>
<property name="cacheLoaderConfig">
  <bean class="org.jboss.cache.config.CacheLoaderConfig">
    <!-- Do not change these -->
    <property name="passivation">true</property>
    <property name="shared">false</property>

    <property name="individualCacheLoaderConfigs">
      <list>
        <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
          <!-- Where passivated sessions are stored -->
          <property name="location">${jboss.server.data.dir}${/}sfsb</property>
          <!-- Do not change these -->
          <property name="async">false</property>
          <property name="fetchPersistentState">true</property>
          <property name="purgeOnStartup">true</property>
          <property name="ignoreModifications">false</property>
          <property name="checkCharacterPortability">false</property>
        </bean>
      </list>
    </property>
  </bean>
</property>

<!-- EJBs use JBoss Cache eviction -->
<property name="evictionConfig">
  <bean class="org.jboss.cache.config.EvictionConfig">
    <property name="wakeupInterval">5000</property>
    <!-- Overall default -->
    <property name="defaultEvictionRegionConfig">
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.NullEvictionAlgorithmConfig"/>
        </property>
      </bean>
    </property>
    <!-- EJB3 integration code will programatically create other regions as
        beans are deployed -->
  </bean>
</property>
</bean>

```

Eviction

The default SFSB 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.

CacheLoader

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 **CacheLoaderConfig**, be sure that you do not use a shared store, for example, 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 other's data.

Buddy Replication

Using buddy replication, state is replicated to a configurable number of backup servers in the cluster (also known as *buddies*), rather than to all servers in the cluster. To enable buddy replication, adjust the following properties in the **buddyReplicationConfig** property bean:

- Set **enabled** to **true**.
- Use the **buddyPoolName** to form logical subgroups of nodes within the cluster. If possible, buddies will be chosen from nodes in the same buddy pool.
- Adjust the **buddyLocatorConfig.numBuddies** property to reflect the number of backup nodes to which each node should replicate its state.

19.3. Stateless Session Bean in EJB 2.x



EJB 2.x is not supported

EJB 2.x is included, but not supported, in JBoss Enterprise Web Platform, and may not be available in future versions.

To make an EJB 2.x 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>
```

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.

bean-load-balance-policy

Indicates the class to be used by the bean stub to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a **RoundRobin** fashion.

19.4. Stateful Session Bean in EJB 2.x



EJB 2.x is not supported

EJB 2.x is included, but not supported, in JBoss Enterprise Web Platform, and may not be available in future versions.

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 Enterprise Web Platform uses the **HASessionStateService** bean 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 **HASessionStateService** bean configuration.

19.4.1. 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 **HASessionStateService** 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 "stick" to the first node in the list.

19.4.2. 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.

19.4.3. The **HASessionStateService** configuration

The **HASessionStateService** bean is defined in the **\$PROFILE/deploy/cluster/ha-legacy-jboss-beans.xml** file.

```
<bean name="HASessionStateService"
      class="org.jboss.ha.hasessionstate.server.HASessionStateService">

  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HASessionState",
    exposedInterface=org.jboss.ha.hasessionstate.server.
    HASessionStateServiceMBean.class,
    registerDirectly=true)</annotation>

  <!-- Partition used for group RPCs -->
  <property name="HAPartition"><inject bean="HAPartition"/></property>

  <!-- JNDI name under which the service is bound -->
  <property name="jndiName">/HASessionState/Default</property>
  <!-- Max delay before cleaning unreclaimed state.
       Defaults to 30*60*1000 => 30 minutes -->
  <property name="beanCleaningDelay">0</property>

</bean>
```

The configuration attributes in the **HASessionStateService** bean are listed below.

HAPartition

A required attribute to inject the **HAPartition** service that HA-JNDI uses for intra-cluster communication.

jndiName

An optional attribute to specify the JNDI name under which this **HASessionStateService** bean is bound. The default value is **/HAPartition/Default**.

beanCleaningDelay

An optional attribute to specify the number of milliseconds after which the **HASessionStateService** 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, which is why **HASessionStateService** must perform this cleanup. The default value is **30*60*1000** milliseconds (that is, 30 minutes).

19.4.4. Handling Cluster Restart

We have covered the HA smart client architecture in [Section 16.2.1, “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/HA-JNDI when the nodes are restarted.

RetryInterceptor 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>
          <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>
        </home>
      </client-interceptors>
    </proxy-factory-config>
  </invoker-proxy-binding>
</jboss>
```

19.4.5. 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. 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 VM 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, set the **java.naming.factory.initial=org.jboss.naming.NamingContextFactory** property in your **jndi.properties**. The advantage of this approach is use of **org.jboss.naming.NamingContextFactory** is simply a configuration option in your **jndi.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 [Chapter 18, Clustered JNDI Services](#) for more on multicast discovery of HA-JNDI.

19.4.6. 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 does not make the **RetryInterceptor** part of its default client interceptor stacks for clustered EJBs.

The **org.jboss.proxy.ejb.SingleRetryInterceptor** was introduced only recently. 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. The **SingleRetryInterceptor** is now 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.

Chapter 20. Clustered Entity EJBs



Remote EJB connectivity is Not Supported

JBoss Enterprise Web Platform does not support remote EJB connectivity. Further, other EJB support will be removed in versions of JBoss Enterprise Web Platform based on JBoss Application Server 6. If you require this functionality, use JBoss Enterprise Application Platform.

In a JBoss Enterprise Web Platform cluster, 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.

20.1. Entity Bean in EJB 3.0

In EJB 3.0, 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.

20.1.1. 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 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.

As well as a region for caching entities, the second-level cache also contains regions for caching collections, queries, and timestamps. The Hibernate setup used for the JBoss EJB 3.0 implementation uses JBoss Cache as its underlying second-level cache implementation.

Configuration of a the second-level cache is done via your EJB3 deployment's **persistence.xml**, like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/persistence"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="tempdb" transaction-type="JTA">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.use_second_level_cache" value="true"/>
      <property name="hibernate.cache.use_query_cache" value="true"/>
      <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
      <!-- region factory specific properties -->
      <property name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
      <property name="hibernate.cache.region.jbc2.cfg.entity" value="mvcc-
entity"/>
      <property name="hibernate.cache.region.jbc2.cfg.collection" value="mvcc-
entity"/>
    </properties>
  </persistence-unit>
</persistence>
```

hibernate.cache.use_second_level_cache

Enables second-level caching of entities and collections.

hibernate.cache.use_query_cache

Enables second-level caching of queries.

hibernate.cache.region.factory_class

Defines the **RegionFactory** implementation that dictates region-specific caching behavior. Hibernate ships with 2 types of JBoss Cache-based second-level caches: shared and multiplexed.

A shared region factory uses the same Cache for all cache regions - much like the legacy CacheProvider implementation in older Hibernate versions.

Hibernate ships with 2 shared region factory implementations:

org.hibernate.cache.jbc2.SharedJBossCacheRegionFactory

Uses a single JBoss Cache configuration, from a newly instantiated CacheManager, for all cache regions.

Table 20.1. Additional properties for SharedJBossCacheRegionFactory

Property	Default	Description
hibernate.cache.region.jbc2.cfg.shared	treecache.xml	The classpath or filesystem resource containing the JBoss Cache configuration settings.
hibernate.cache.region.jbc2.cfg.jgroups.stacks	org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	The classpath or filesystem resource containing the JGroups protocol stack configurations.

org.hibernate.cache.jbc2.JndiSharedJBossCacheRegionFactory

Uses a single JBoss Cache configuration, from an existing CacheManager bound to JNDI, for all cache regions.

Table 20.2. Additional properties for JndiSharedJBossCacheRegionFactory

Property	Default	Description
hibernate.cache.region.jbc2.cfg.shared	<i>Required</i>	JNDI name to which the shared Cache instance is bound.

A multiplexed region factory uses separate Cache instances, using optimized configurations for each cache region.

Table 20.3. Common properties for multiplexed region factory implementations

Property	Default	Description
hibernate.cache.region.jbc2.cfg.entity	optimistic-entity	The JBoss Cache configuration used for the entity cache region. Alternative configurations: mvcc-entity, pessimistic-entity, mvcc-entity-repeatable, optimistic-entity-repeatable, pessimistic-entity-repeatable
hibernate.cache.region.jbc2.cfg.collection	optimistic-entity	The JBoss Cache configuration used for the collection cache region. The collection cache region typically uses the same configuration as the entity cache region.
hibernate.cache.region.jbc2.cfg.query	local-query	The JBoss Cache configuration used for the query cache region. By default, cached query results are not replicated. Alternative configurations: replicated-query
hibernate.cache.region.jbc2.cfg.ts	timestamps-cache	The JBoss Cache configuration used for the timestamp cache region. If query caching is used, the corresponding timestamp cache must be replicating, even if the query cache is non-replicating. The timestamp cache region must never share the same cache as the query cache.

Hibernate ships with 2 shared region factory implementations:

org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory

Uses separate JBoss Cache configurations, from a newly instantiated CacheManager, per cache region.

Table 20.4. Additional properties for MultiplexedJBossCacheRegionFactory

Property	Default	Description
hibernate.cache.region.jbc2.configs	org/hibernate/cache/jbc2/builder/jbc2-configs.xml	The classpath or filesystem resource containing the JBoss Cache configuration settings.
hibernate.cache.region.jbc2.cfg.jgroups.stacks	org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	The classpath or filesystem resource containing the JGroups protocol stack configurations.

org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory

Uses separate JBoss Cache configurations, from a JNDI-bound CacheManager, see [Section 17.1.3.1, “The JBoss Enterprise Web Platform CacheManager Service”](#), per cache region.

Table 20.5. Additional properties for JndiMultiplexedJBossCacheRegionFactory

Property	Default	Description
hibernate.cache.region.jbc2.cachefactory	<i>Required</i>	JNDI name to which the CacheManager instance is bound.

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.

20.1.2. Configure the entity beans for cache

Next we need to configure which entities to cache. 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 appropriate JBoss Cache configuration file, e.g. **jboss-cache-manager-jboss-beans.xml**. For instance, you can specify the size of the cache. If there are too many objects in the cache, the cache can evict the oldest 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 would be

/myprefix/com/mycompany/entities/Account.

```

<bean name="..." class="org.jboss.cache.config.Configuration">
    ...
    <property name="evictionConfig">
        <bean class="org.jboss.cache.config.EvictionConfig">
            <property name="wakeupInterval">5000</property>
            <!-- Overall default -->
            <property name="defaultEvictionRegionConfig">
                <bean class="org.jboss.cache.config.EvictionRegionConfig">
                    <property name="regionName">/</property>
                    <property name="evictionAlgorithmConfig">
                        <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                            <!-- Evict LRU node once we have more than this number of nodes -->
                            <property name="maxNodes">10000</property>
                            <!-- And, evict any node that hasn't been accessed in this many
seconds -->
                            <property name="timeToLiveSeconds">1000</property>
                            <!-- Don't evict a node that's been accessed within this many
seconds.
length. -->
                            Set this to a value greater than your max expected transaction
                            <property name="minTimeToLiveSeconds">120</property>
                        </bean>
                    </property>
                </bean>
            </property>
            <property name="evictionRegionConfigs">
                <list>
                    <bean class="org.jboss.cache.config.EvictionRegionConfig">
                        <property
name="regionName">/myprefix/com/mycompany/entities/Account</property>
                        <property name="evictionAlgorithmConfig">
                            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                                <property name="maxNodes">10000</property>
                                <property name="timeToLiveSeconds">5000</property>
                                <property name="minTimeToLiveSeconds">120</property>
                            </bean>
                        </property>
                    </bean>
                    ...
                </list>
            </property>
        </bean>
    </property>
</bean>

```

If you do not specify a cache region for an entity bean class, all instances of this class will be cached using the **defaultEvictionRegionConfig** 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:

```

<bean name="..." class="org.jboss.cache.config.Configuration">
  ... ..
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeupInterval">5000</property>
      <!-- Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">5000</property>
              <property name="timeToLiveSeconds">1000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
      </property>
    </bean>
  </property>
  <property name="evictionRegionConfigs">
    <list>
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/myprefix/Account</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
            <property name="maxNodes">10000</property>
            <property name="timeToLiveSeconds">5000</property>
            <property name="minTimeToLiveSeconds">120</property>
          </bean>
        </property>
      </bean>
      ... ..
    </list>
  </property>
</bean>
</property>
</bean>

```

20.1.3. Query result caching

The EJB3 Query API also provides means for you to save the results (i.e., collections of primary keys of entity beans, or collections of scalar values) of specified queries in the second-level cache. 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:

```

<bean name="..." class="org.jboss.cache.config.Configuration">
  ... ..
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeupInterval">5000</property>
      <!-- Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">5000</property>
              <property name="timeToLiveSeconds">1000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
      </property>
    </bean>
  </property>
  <property name="evictionRegionConfigs">
    <list>
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/myprefix/Account</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
            <property name="maxNodes">10000</property>
            <property name="timeToLiveSeconds">5000</property>
            <property name="minTimeToLiveSeconds">120</property>
          </bean>
        </property>
      </bean>
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property
name="regionName">/myprefix/org/hibernate/cache/StandardQueryCache</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
            <property name="maxNodes">100</property>
            <property name="timeToLiveSeconds">600</property>
            <property name="minTimeToLiveSeconds">120</property>
          </bean>
        </property>
      </bean>
    </list>
  </property>
</bean>
</property>
</bean>

```

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:

```

<bean name="..." class="org.jboss.cache.config.Configuration">
  ... ..
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeupInterval">5000</property>
      <!-- Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">5000</property>
              <property name="timeToLiveSeconds">1000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
      </property>
    </bean>
  </property>
  <property name="evictionRegionConfigs">
    <list>
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/myprefix/Account</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
            <property name="maxNodes">10000</property>
            <property name="timeToLiveSeconds">5000</property>
            <property name="minTimeToLiveSeconds">120</property>
          </bean>
        </property>
      </bean>
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/myprefix/Queries</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
            <property name="maxNodes">100</property>
            <property name="timeToLiveSeconds">600</property>
            <property name="minTimeToLiveSeconds">120</property>
          </bean>
        </property>
      </bean>
      ... ..
    </list>
  </property>
</bean>
</property>
</bean>

```

20.2. Entity Bean in EJB 2.x



EJB 2.x is not supported

EJB 2.x is included, but not supported, in JBoss Enterprise Web Platform, and may not be available in future versions.

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.

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.

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.



Note

If you are using Bean Managed Persistence (BMP), you are going to have to implement synchronization on your own.

Chapter 21. HTTP Services

HTTP session replication is used to replicate the state associated with web client sessions to other nodes in a cluster. Thus, in the event one of your nodes crashes, another node in the cluster will be able to recover. Two distinct functions must be performed:

- Session state replication
- Load-balancing HTTP Requests

State replication is directly handled by JBoss. When you run JBoss in the **production** configuration, session state replication is enabled by default. Just configure your web application as **<distributable>** in its **web.xml** (see [Section 21.2, “Configuring HTTP session state replication”](#)), 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 or session affinity: 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 request for a session 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 or on the client. In other situations, losing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

21.1. 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.

21.1.1. Download the software

First of all, make sure that you have Apache installed. You can download Apache directly from the Apache web site at <http://httpd.apache.org/>. Its installation is straightforward and requires no specific configuration. We advise you to use the latest stable version of Apache, which is version 2.2.x. We will assume, 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 the use of mod_jk 1.2.x, as both earlier versions of mod_jk, and mod_jk2, are deprecated, unsupported and no further development is 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/**.

21.1.2. 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/uriworkmap.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 downloaded and placed in **\$APACHE_HOME/modules/**. You must indicate the same name, including 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 serve static contents (or PHP contents) directly and only use the load balancer for Java applications. If you only use **mod_jk** as a load balancer, you can also forward all URLs (that is, **/***) 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 code 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 made 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.

21.1.3. Configure worker nodes in mod_jk

Next, you need to configure the **mod_jk** file **conf/workers.properties** file. 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 node setup, the file might 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
```

The above file configures **mod_jk** to perform weighted round-robin load balancing with sticky sessions between two servlet containers (JBoss Enterprise Web Platform instances) **node1** and **node2** listening on port **8009**.

In the **workers.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 (that is, the number of concurrent requests it will forward to the Servlet container). Make sure this number is not greater than the number of threads configured on the AJP13 connector of the Servlet container. See <http://tomcat.apache.org/connectors-doc/reference/workers.html> for comments on **cachesize** for Apache 1.3.x.

The last part of the **conf/workers.properties** file defines the load balancer 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**; that is, 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.

21.1.4. Configuring JBoss to work with mod_jk

Finally, we must configure the JBoss Enterprise Web Platform instances on all clustered nodes so that they can expect requests forwarded from the mod_jk load balancer.

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/\$PROFILE/deploy/jbossweb.sar/server.xml** file. Locate the **<Engine>** element and add an attribute **jvmRoute**:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
</Engine>
```

You also need to ensure that the AJP connector in **server.xml** is enabled (that is, uncommented). It is enabled by default.

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector protocol="AJP/1.3" port="8009" address="{jboss.bind.address}"
    redirectPort="8443" />
```

At this point, you have a fully working Apache with 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 Application Server, refer to the JBoss wiki page at <http://www.jboss.org/community/wiki/UsingModjk12WithJBoss>.

21.1.5. Configuring the NSAPI connector on Solaris

This section shows you how to configure the NSAPI connector to use a JBoss Enterprise Platform as a worker node for a Sun Java System Web Server (SJWS) master node.

**Note**

Sun Java System Web Server has recently been renamed to the Oracle iPlanet Web Server.

In this section, all of the server instances are on the same machine. To use different machines for each instance, use the **-b** switch to bind your instance of JBoss Enterprise Platform to a public IP address. Remember to edit the **workers.properties** file on the SJWS machine to reflect these changes in IP address.

21.1.5.1. Prerequisites

This section assumes that:

- Your worker node(s) are already installed with a JBoss Enterprise Platform 5.1 or later. The Native components are optional. Refer to the *Installation Guide* for assistance with this prerequisite.
- Your master node is already installed with any of the following technology combinations, and the appropriate Native binary for its operating system and architecture. Refer to the *Installation Guide* for assistance with this prerequisite.
 - Solaris 9 x86 with Sun Java System Web Server 6.1 SP12
 - Solaris 9 SPARC 64 with Sun Java System Web Server 6.1 SP12
 - Solaris 10 x86 with Sun Java System Web Server 7.0 U8
 - Solaris 10 SPARC 64 with Sun Java System Web Server 7.0 U8

21.1.5.2. Configure JBoss Enterprise Platform as a Worker Node

This section shows you how to safely configure your JBoss Enterprise Platform instance as a worker node for use with Sun SJWS.

Procedure 21.1. Configure a JBoss Enterprise Platform instance as a worker node

1. Create a server profile for each worker node

Make a copy of the server profile that you wish to configure as a worker node. (This procedure uses the **default** server profile.)

```
[user@workstation jboss-eap-5.1]$ cd jboss-as/server
[user@workstation server]$ cp -r default/ default-01
[user@workstation server]$ cp -r default/ default-02
```

2. Give each instance a unique name

Edit the following line in the **deploy/jbossweb.sar/server.xml** file of each new worker instance:

```
<Engine name="jboss.web" defaultHost="localhost">
```

Add a unique **jvmRoute** value, as shown. This value is the identifier for this node in the cluster.

For the **default-01** server profile:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="worker01">
```

For the **default-02** server profile:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="worker02">
```

3. Enable session handling

Edit the following line in the **deployers/jbossweb.deployer/META-INF/war-deployers-jboss-beans.xml** file of each worker node:

```
<property name="useJK">false</property>
```

This property controls whether special session handling is used to coordinate with mod_jk and other connector variants. Set this property to **true** in both worker nodes:

```
<property name="useJK">true</property>
```

4. Start your worker nodes

Start each worker node in a separate command line interface. Ensure that each node is bound to a different IP address with the **-b** switch.

```
[user@workstation jboss-eap-5.1]$ ./jboss-as/bin/run.sh -b 127.0.0.1 -c
default-01
```

```
[user@workstation jboss-eap-5.1]$ ./jboss-as/bin/run.sh -b 127.0.0.100 -c
default-02
```

21.1.5.3. Configure Sun Java System Web Server for Clustering

The procedures in the following sections assume that the contents of the Native zip appropriate for your operating system and architecture have been extracted to **/tmp/connectors/jboss-ep-native-5.1/**. This path is referred to as **NATIVE** in the procedures that follow. These procedures also assume that the **/tmp/connectors** directory is used to store logs, properties files and NSAPI locks.

These procedures also assume that your installation of Sun Java System Web Server is in one of the following locations, depending on your version of Solaris:

- for Solaris 9 x86 or SPARC 64: **/opt/SUNWwbsrv61/**
- for Solaris 10 x86 or SPARC 64: **/opt/SUNWwbsrv70/**

This path is referred to as **SJWS** in the procedures that follow.

Procedure 21.2. Initial clustering configuration

1. Disable servlet mappings

Under *Built In Servlet Mappings* in the **SJWS/PROFILE/config/default-web.xml** file, disable the mappings for the following servlets, as shown in the code sample:

- default
- invoker
- jsp

```

<!-- ===== Built In Servlet Mappings ===== -
->

<!-- The servlet mappings for the built in servlets defined above. -->

<!-- The mapping for the default servlet -->
<!--servlet-mapping>
  <servlet-name>default</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping-->

<!-- The mapping for the invoker servlet -->
<!--servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/servlet/*</url-pattern>
</servlet-mapping-->

<!-- The mapping for the JSP servlet -->
<!--servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping-->

```

2. Load the required modules and properties

Append the following lines to the ***SJWS/PROFILE/config/magnus.conf*** file:

```

Init fn="load-modules" funcs="jk_init,jk_service"
shlib="NATIVE/lib/nsapi_redirector.so" shlib_flags="(global|now)"
Init fn="jk_init" worker_file="/tmp/connectors/workers.properties"
log_level="debug" log_file="/tmp/connectors/nsapi.log"
shm_file="/tmp/connectors/jk_shm"

```

These lines define the location of the **nsapi_redirector.so** module used by the **jk_init** and **jk_service** functions, and the location of the **workers.properties** file, which defines the worker nodes and their attributes.



Note

The **lib** directory in the **NATIVE/lib/nsapi_redirector.so** path applies only to 32-bit machines. On 64-bit machines, this directory is called **lib64**.

21.1.5.3.1. Configure a basic cluster with NSAPI

Use the following procedure to configure a basic cluster, where requests for particular paths are forwarded to particular worker nodes. In [Procedure 21.3, "Configure a basic cluster with NSAPI"](#), worker02 serves the **/nc** path, while worker01 serves **/status** and all other paths defined in the first part of the **obj.conf** file.

Procedure 21.3. Configure a basic cluster with NSAPI

1. Define the paths to serve via NSAPI

Edit the ***SJWS/PROFILE/config/obj.conf*** file. Define paths that should be served via NSAPI at the end of the **default** Object definition, as shown:

```
<Object name="default">
  [...]
  NameTrans fn="assign-name" from="/status" name="jknsapi"
  NameTrans fn="assign-name" from="/images(|/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/css(|/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/nc(|/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/jmx-console(|/*)" name="jknsapi"
</Object>
```

You can map the path of any application deployed on your JBoss Enterprise Platform instance in this **obj.conf** file. In the example code, the **/nc** path is mapped to an application deployed under the name **nc**.

2. Define the worker that serves each path

Edit the **\$JWS/PROFILE/config/obj.conf** file and add the following **jknsapi** Object definition after the **default** Object definition.

```
<Object name="jknsapi">
  ObjectType fn="force-type" type="text/plain
  Service fn="jk_service" worker="worker01" path="/status"
  Service fn="jk_service" worker="worker02" path="/nc(/*)"
  Service fn="jk_service" worker="worker01"
</Object>
```

This **jknsapi** Object defines the worker nodes used to serve each path that was assigned to **name="jknsapi"** in the **default** Object.

In the example code, the third Service definition does not specify a **path** value, so the worker node defined (**worker01**) serves all of the paths assigned to **jknsapi** by default. In this case, the first Service definition in the example code, which assigns the **/status** path to **worker01**, is superfluous.

3. Define the workers and their attributes

Create a **workers.properties** file in the location you defined in [Step 2](#). Define the list of worker nodes and each worker node's properties in this file, like so:

```
# An entry that lists all the workers defined
worker.list=worker01, worker02

# Entries that define the host and port associated with these workers
worker.worker01.host=127.0.0.1
worker.worker01.port=8009
worker.worker01.type=ajp13

worker.worker02.host=127.0.0.100
worker.worker02.port=8009
worker.worker02.type=ajp13
```

21.1.5.3.2. Configure a Load-balanced Cluster with NSAPI

Procedure 21.4. Configure a load-balancing cluster with NSAPI

1. Define the paths to serve via NSAPI

Edit the **\$JWS/PROFILE/config/obj.conf** file. Define paths that should be served via NSAPI at the end of the **default** Object definition, as shown:

```
<Object name="default">
  [...]
  NameTrans fn="assign-name" from="/status" name="jknsapi"
  NameTrans fn="assign-name" from="/images(/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/css(/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/nc(/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/jmx-console(/*)" name="jknsapi"
  NameTrans fn="assign-name" from="/jkmanager/*" name="jknsapi"
</Object>
```

You can map the path of any application deployed on your JBoss Enterprise Platform instance in this **obj.conf** file. In the example code, the **/nc** path is mapped to an application deployed under the name **nc**.

2. Define the worker that serves each path

Edit the **\$JWS/PROFILE/config/obj.conf** file and add the following **jknsapi** Object definition after the **default** Object definition.

```
<Object name="jknsapi">
  ObjectType fn="force-type" type="text/plain"
  Service fn="jk_service" worker="status" path="/jkmanager(/*)"
  Service fn="jk_service" worker="router"
</Object>
```

This **jknsapi** Object defines the worker nodes used to serve each path that was assigned to **name="jknsapi"** in the **default** Object.

3. Define the workers and their attributes

Create a **workers.properties** file in the location you defined in [Step 2](#). Define the list of worker nodes and each worker node's properties in this file, like so:

```
# The advanced router LB worker
worker.list=router,status

# Define a worker using ajp13
worker.worker01.port=8009
worker.worker01.host=127.0.0.1
worker.worker01.type=ajp13
worker.worker01.ping_mode=A
worker.worker01.socket_timeout=10
worker.worker01.lbfactor=3

# Define another worker using ajp13
worker.worker02.port=8009
worker.worker02.host=127.0.0.100
worker.worker02.type=ajp13
worker.worker02.ping_mode=A
worker.worker02.socket_timeout=10
worker.worker02.lbfactor=1

# Define the LB worker
worker.router.type=lb
worker.router.balance_workers=worker01,worker02

# Define the status worker
worker.status.type=status
```

21.1.5.3.3. Restart Sun Java System Web Server

Once your Sun Java System Web Server instance is configured, restart it so that your changes take effect.

For Sun Java System Web Server 6.1:

```
SJWS/PROFILE/stop  
SJWS/PROFILE/start
```

For Sun Java System Web Server 7.0:

```
SJWS/PROFILE/bin/stopserv  
SJWS/PROFILE/bin/startserv
```

21.2. Configuring HTTP session state replication

The preceding discussion has been focused on using `mod_jk` as a load balancer. The following information about clustering HTTP services in JBoss Enterprise Web Platform applies regardless of the load balancer used.

In [Section 21.1.3, “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 more reliable solution is to replicate session data across the nodes in the cluster. This way, if a server node fails or is shut down, the load balancer can fail over the next client request to any server node and obtain the same session state.

21.2.1. Enabling session replication in your application

To enable replication of your web application you must tag the application 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. However, the **replication-config** element only needs to be set if one or more of the default values described below is unacceptable. Here is an example:

```
<!DOCTYPE jboss-web PUBLIC
-//JBoss//DTD Web Application 5.0//EN
http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd>

<jboss-web>

  <replication-config>
    <cache-name>custom-session-cache</cache-name>
    <replication-trigger>SET</replication-trigger>
    <replication-granularity>ATTRIBUTE</replication-granularity>
    <replication-field-batch-mode>true</replication-field-batch-mode>
    <use-jk>false</use-jk>
    <max-unreplicated-interval>30</max-unreplicated-interval>
    <snapshot-mode>INSTANT</snapshot-mode>
    <snapshot-interval>1000</snapshot-interval>
    <session-notification-
policy>com.example.CustomSessionNotificationPolicy</session-notification-policy>
  </replication-config>

</jboss-web>
```

All of the above configuration elements are optional and can be omitted if the default value is acceptable. A couple are commonly used; the rest are very infrequently changed from the defaults. We'll cover the commonly used ones first.

The **replication-trigger** element determines when the container should consider that session data must be replicated across the cluster. The rationale for this setting is that after a mutable object stored as a session attribute is accessed from the session, in the absence of a **setAttribute** call the container has no clear way to know if the object (and hence the session state) has been modified and needs to be replicated. This element has three valid values:

SET_AND_GET

Always replicates session data, even if its content has not been modified but simply accessed. This previously ensured that every request triggered a timestamp. Since setting **max_unreplicated_interval** to 0 accomplishes the same thing at much lower cost, using **SET_AND_GET** is not sensible with Enterprise Web Platform 5.

SET_AND_NON_PRIMITIVE_GET

Only replicates if an object of a non-primitive type has been accessed (that is, the object is not of a well-known immutable JDK type, such as **Integer**, **Long**, **String**, etc.) This is the default value.

SET

Assumes that the developer will explicitly call **setAttribute** on the session if the data needs to be replicated. This setting prevents unnecessary replication and can have major performance benefits, but requires very good coding practices to ensure that **setAttribute** is always called whenever a mutable object stored in the session is modified.

In all cases, calling **setAttribute** marks the session as needing replication.

The **replication-granularity** element determines the granularity of what gets replicated if the container determines session replication is needed. The supported values are:

SESSION

Indicates that the entire session attribute map should be replicated when any attribute is considered modified. Replication occurs at request end. This option replicates the most data

and thus incurs the highest replication cost, but since all attributes values are always replicated together it ensures that any references between attribute values will not be broken when the session is deserialized. For this reason it is the default setting.

ATTRIBUTE

Indicates that only attributes that the session considers to be potentially modified are replicated. Replication occurs at request end. For sessions carrying large amounts of data, parts of which are infrequently updated, this option can significantly increase replication performance. However, it is not suitable for applications that store objects in different attributes that share references with each other (for example, a **Person** object in the **husband** attribute sharing a reference to an **Address** object with another **Person** in the **wife** attribute). This is because if the attributes are separately replicated, when the session is deserialized on remote nodes the shared references will be broken.

FIELD

Useful if the classes stored in the session have been bytecode enhanced for use by POJO Cache. If they have been, the session management layer will detect field level changes within objects stored to the session, and will replicate only those changes. This is the most performant setting. 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 other elements under the **replication-config** element are much less frequently used.

cacheName

Indicates the name of the JBoss Cache configuration that should be used for storing distributable sessions and replicating them around the cluster. This element lets web applications that require different caching characteristics specify the use of separate, differently configured, JBoss Cache instances. The default value is **standard-session-cache** if the **replication-granularity** is not **FIELD**, and **field-granularity-session-cache** if it is. See [Section 21.2.3, “Configuring the JBoss Cache instance used for session state replication”](#) for more details on JBoss Cache configuration for web tier clustering.

replication-field-batch-mode

Indicates whether all replication messages associated with a request will be batched into one message. This is applicable only if **replication-granularity** is **FIELD**. If **replication-field-batch-mode** is set to **true**, fine-grained changes made to objects stored in the session attribute map will replicate only when the HTTP request is finished; otherwise they replicate as they occur. Setting this to **false** is not advised. The default value is **true**.

useJK

Indicates whether the container should assume that a JK-based software load balancer (for example, **mod_jk**, **mod_proxy**, **mod_cluster**) is being used for load balancing for this web application. If set to **true**, the container will examine the session ID associated with every request and replace the **jvmRoute** portion of the session ID if it detects a failover.

The default value is **null** (that is, unspecified). In this case the session manager will use the presence or absence of a **jvmRoute** configuration on its enclosing JBoss Web **Engine** (see [Section 21.1.4, “Configuring JBoss to work with mod_jk”](#)) to determine whether JK is used. **useJK** need only be set to **false** for web applications whose URL cannot be handled by the JK load balancer.

max-unreplicated-interval

Configures the maximum interval between requests, in seconds, after which a request will trigger replication of the session's timestamp regardless of whether the request has otherwise made the session dirty. Such replication ensures that other nodes in the cluster are aware of the most recent value for the session's timestamp and won't incorrectly expire an unreplicated session upon failover. It also results in correct values for

HttpSession.getLastAccessedTime() calls following failover.

A value of **0** means the timestamp will be replicated whenever the session is accessed. A value of **-1** means the timestamp will be replicated only if some other activity during the request (for example, modifying an attribute) has resulted in other replication work involving the session. A positive value greater than the **HttpSession.getMaxInactiveInterval()** value will be treated as a probable misconfiguration and converted to **0**; that is, metadata will be replicated on every request. The default value is **60**.

snapshot-mode

Configures when sessions are replicated to the other nodes. Possible values are **INSTANT** (the default) and **INTERVAL**.

The typical value, **INSTANT**, replicates changes to the other nodes at the end of requests, using the request processing thread to perform the replication. In this case, the **snapshot-interval** property is ignored.

With **INTERVAL** mode, a background task is created that runs every **snapshot-interval** milliseconds, checking for modified sessions and replicating them.

Note that this property has no effect if **replication-granularity** is set to **FIELD**. If it is **FIELD**, **INSTANT** mode will be used.

snapshot-interval

Defines how often (in milliseconds) the background task that replicates modified sessions should be started for this web application. Only meaningful if **snapshot-mode** is set to **INTERVAL**.

session-notification-policy

Specifies the fully qualified class name of the implementation of the **ClusteredSessionNotificationPolicy** interface that should be used to govern whether servlet specification notifications should be emitted to any registered **HttpSessionListener**, **HttpSessionAttributeListener** or **HttpSessionBindingListener**.

Sensible event notifications for a non-clustered environment may not remain sensible in a clustered environment. (See <https://jira.jboss.org/jira/browse/JBAS-5778> for an example of why a notification may not be desired.) Configuring an appropriate **ClusteredSessionNotificationPolicy** gives the application author fine-grained control over what notifications are issued.

If no value is explicitly set, the default behavior is **IgnoreUndeployLegacyClusteredSessionNotificationPolicy**, which implements the same behavior except during undeployment, during which no **HttpSessionListener** and **HttpSessionAttributeListener** notifications are sent.

21.2.2. HttpSession Passivation and Activation

Passivation is the process of controlling memory usage by removing relatively unused sessions from memory while storing them in persistent storage. If a passivated session is requested by a client, it can be "activated" back into memory and removed from the persistent store. JBoss Enterprise Web Platform 5 supports passivation of `HttpSessions` from web applications whose `web.xml` includes the **distributable** tag (that is, clustered web applications).

Passivation occurs at three points during the lifecycle of a web application:

- When the container requests the creation of a new session. If the number of currently active sessions exceeds a configurable limit, an attempt is made to passivate sessions to make room in memory.
- Periodically (by default every ten seconds) as the JBoss Web background task thread runs.
- When the web application is deployed and a backup copy of sessions active on other servers is acquired by the newly deploying web application's session manager.

A session will be passivated if one of the following holds true:

- The session has not been in use for longer than a configurable maximum idle time.
- The number of active sessions exceeds a configurable maximum and the session has not been in use for longer than a configurable minimum idle time.

In both cases, sessions are passivated on a Least Recently Used (LRU) basis.

21.2.2.1. Configuring HttpSession Passivation

Session passivation behavior is configured via the `jboss-web.xml` deployment descriptor in your web application's **WEB-INF** directory.

```
<!DOCTYPE jboss-web PUBLIC
-//JBoss//DTD Web Application 5.0//EN
http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd>

<jboss-web>

  <max-active-sessions>20</max-active-sessions>
  <passivation-config>
    <use-session-passivation>true</use-session-passivation>
    <passivation-min-idle-time>60</passivation-min-idle-time>
    <passivation-max-idle-time>600</passivation-max-idle-time>
  </passivation-config>

</jboss-web>
```

max-active-session

Determines the maximum number of active sessions allowed. If the number of sessions managed by the session manager exceeds this value and passivation is enabled, the excess will be passivated based on the configured **passivation-min-idle-time**. If after passivation is completed (or if passivation is disabled), the number of active sessions still exceeds this limit, attempts to create new sessions will be rejected. If set to **-1** (the default), there is no limit.

use-session-passivation

Determines whether session passivation will be enabled for the web application. Default is **false**.

passivation-min-idle-time

Determines the minimum time (in seconds) that a session must have been inactive before the

container will consider passivating it in order to reduce the active session count to obey the value defined by **max-active-sessions**. A value of **-1** (the default) disables passivating sessions before **passivation-max-idle-time**. Neither a value of **-1** nor a high value are recommended if **max-active-sessions** is set.

passivation-max-idle-time

Determines the maximum time (in seconds) that a session can be inactive before the container should attempt to passivate it to save memory. Passivation of such sessions will take place regardless of whether the active session count exceeds **max-active-sessions**. Should be less than the **web.xml session-timeout** setting. A value of **-1** (the default) disables passivation based on maximum inactivity.

The total number of sessions in memory includes sessions replicated from other cluster nodes that are not being accessed on this node. Take this into account when setting **max-active-sessions**. The number of sessions replicated from other nodes will also depend on whether *buddy replication* is enabled.

Say, for example, that you have an eight node cluster, and each node handles requests from 100 users. With *total replication*, each node would store 800 sessions in memory. With *buddy replication* enabled, and the default **numBuddies** setting (**1**), each node will store 200 sessions in memory.

21.2.3. Configuring the JBoss Cache instance used for session state replication

The container for a distributable web application makes use of JBoss Cache to provide HTTP session replication services around the cluster. The container integrates with the **CacheManager** service to obtain a reference to a JBoss Cache instance (see [Section 17.1.3.1, “The JBoss Enterprise Web Platform CacheManager Service”](#)).

The name of the JBoss Cache configuration to use is controlled by the **cacheName** element in the application's **jboss-web.xml** (see [Section 21.2.1, “Enabling session replication in your application”](#)). In most cases, though, this does not need to be set as the default values of **standard-session-cache** and **field-granularity-session-cache** (for applications configured for **FIELD** granularity) are appropriate.

The JBoss Cache configurations in the **CacheManager** service expose a number of options. See [Chapter 24, JBoss Cache Configuration and Deployment](#) and the JBoss Cache documentation for a more complete discussion. The **standard-session-cache** and **field-granularity-session-cache** configurations are already optimized for the web session replication use case, and most of the settings should not be altered. Administrators may be interested in altering the following settings:

cacheMode

The default is **REPL_ASYNC**, which specifies that a session replication message sent to the cluster does not wait for responses from other cluster nodes confirming that the message has been received and processed. The alternative mode, **REPL_SYNC**, offers a greater degree of confirmation that session state has been received, but reduces performance significantly. See [Section 24.1.2, “Cache Mode”](#) for further details.

enabled (in buddyReplicationConfig)

Set to **true** to enable buddy replication. See [Section 24.1.8, “Buddy Replication”](#). Default is **false**.

numBuddies (in buddyReplicationConfig)

Set to a value greater than the default (**1**) to increase the number of backup nodes onto which sessions are replicated. Only relevant if buddy replication is enabled. See [Section 24.1.8,](#)

[“Buddy Replication”](#).

buddyPoolName (in buddyReplicationConfig)

A way to specify a preferred replication group when buddy replication is enabled. JBoss Cache tries to pick a buddy who shares the same pool name (falling back to other buddies if not available). Only relevant if buddy replication is enabled. See [Section 24.1.8, “Buddy Replication”](#).

multiplexerStack

Name of the JGroups protocol stack the cache should use. See [Section 17.1.1, “The Channel Factory Service”](#).

clusterName

Specifies the name JGroups will use for this cache's channel. Only change this if you create a new cache configuration, in which case this property should have a different value from all other cache configurations.

If you wish to use a completely new JBoss Cache configuration rather than editing one of the existing ones, please see [Section 24.2.1, “Deployment Via the CacheManager Service”](#).

21.3. Using FIELD-level replication



This feature is deprecated

This feature is deprecated as of JBoss Enterprise Web Platform 5.1, and will be removed in a future release of JBoss Enterprise Web Platform. Customers are recommended to migrate away from this feature in existing implementations, and not use it in new implementations.

FIELD-level replication only replicates modified data fields inside objects stored in the session. It can reduce the data traffic between clustered nodes, and hence improve the performance of the whole cluster. To use FIELD-level replication, you must first prepare (that is, 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.

First, you need to identify the classes that you need to prepare. You can identify these classes by using annotations, like so:

```
@org.jboss.cache.pojo.annotation.Replicable
public class Address
{
    ...
}
```

If you annotate a class with **@Replicable**, then all of its subclasses will be automatically annotated as well. Similarly, you can annotate an interface with **@Replicable** and all of its implementing classes will be annotated. For example:

```
@org.jboss.cache.aop.InstanceOfAopMarker
public class Person
{
    ...
}

public class Student extends Person
{
    ...
}
```

There is no need to annotate **Student**. POJO Cache will recognize it as `@Replicable` because it is a sub-class of **Person**.

Once you have annotated your classes, you will need to perform a pre-processing step to bytecode enhance your classes for use by POJO Cache. 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 of 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]
```

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 of how to build, deploy, and validate a FIELD-level replicated web application from this page: <http://www.jboss.org/community/wiki/httpsessionfieldlevelexample>. The example bundles the pre- and post-compile tools so you do not need to download JBoss AOP separately.

Finally, let's see an example on how to use FIELD-level replication on those data classes. First, we see some servlet code that reads some data from the request parameters, creates a couple of objects and stores them in the session:

```
Person husband = new Person(getHusbandName(request), getHusbandAge(request));
Person wife = new Person(getWifeName(request), getWifeAge(request));
Address addr = new Address();
addr.setPostalCode(getPostalCode(request));

husband.setAddress(addr);
wife.setAddress(addr); // husband and wife share the same address!

session.setAttribute("husband", husband); // that's it.
session.setAttribute("wife", wife); // that's it.
```

Later, a different servlet could update the family's postal code:

```
Person wife = (Person)session.getAttribute("wife");
wife.getAddress().setPostalCode(getPostalCode(request)); // this
will update and replicate the postal code
```

Notice that in 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.

Besides plain objects, you can also use regular Java collections of those objects as session attributes. POJO Cache automatically figures out how to handle those collections and replicate field changes in their member objects.

21.4. Using Clustered Single Sign-on (SSO)

JBoss supports clustered single sign-on, allowing a user to authenticate to one web application and to be recognized on all web applications that are deployed on the same virtual host, whether or not they are deployed on that same machine or on another node in the cluster. Authentication replication is handled by JBoss Cache. Clustered single sign-on support is a JBoss-specific extension of the non-clustered `org.apache.catalina.authenticator.SingleSignOn` valve that is a standard part of Tomcat and JBoss Web. Both the non-clustered and clustered versions allow users to sign on to any one of the web apps associated with a virtual host and have their identity recognized by all other web applications on the same virtual host. The clustered version brings the added benefits of enabling SSO failover and allowing a load balancer to direct requests for different web applications to different servers, while maintaining the SSO.

21.4.1. Configuration

To enable clustered single sign-on, you must add the `ClusteredSingleSignOn` valve to the appropriate `Host` elements of the `JBOSS_HOME/server/production/deploy/jbossweb.sar/server.xml` file. The valve element is already included in the standard file; you just need to uncomment it. The valve configuration is shown here:

```
<Valve className="org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn" />
```

The element supports the following attributes:

className

Required. Sets the Java class name of the valve implementation to use. This must be set to `org.jboss.web.tomcat.service.sso.ClusteredSingleSign`.

cacheConfig

Specifies the name of the cache configuration (see [Section 17.1.3.1, "The JBoss Enterprise Web Platform CacheManager Service"](#)) to use for the clustered SSO cache. Default is `clustered-ss`.

treeCacheName

Deprecated; use `cacheConfig`. Specifies a JMX ObjectName of the JBoss Cache MBean to use for the clustered SSO cache. If no cache can be located from the CacheManager service using the value of `cacheConfig`, an attempt to locate an mbean registered in JMX under this ObjectName will be made. Default value is `jboss.cache:service=TomcatClusteringCache`.

cookieDomain

Sets the host domain to be used for SSO cookies. See [Section 21.4.4, "Configuring the Cookie Domain"](#) for more. Default is `"/"`.

maxEmptyLife

The maximum number of seconds an SSO with no active sessions will be usable by a request. The clustered SSO valve tracks what cluster nodes are managing sessions related to an SSO. A positive value for this attribute allows proper handling of shutdown of a node that is the only

one that had handled any of the sessions associated with an SSO. The shutdown invalidates the local copy of the sessions, eliminating all sessions from the SSO. If **maxEmptyLife** were zero, the SSO would terminate along with the local session copies. But, backup copies of the sessions (if they are from clustered webapps) are available on other cluster nodes. Allowing the SSO to live beyond the life of its managed sessions gives the user time to make another request which can fail over to a different cluster node, where it activates the the backup copy of the session. Default is **1800** (30 minutes).

processExpiresInterval

The minimum number of seconds between efforts by the valve to find and invalidate SSO's that have exceeded their **maxEmptyLife**. Does not imply effort will be spent on such cleanup every **processExpiresInterval**, just that it won't occur more frequently than that. Default is **60**.

requireReauthentication

A flag to determine whether each request needs to be reauthenticated to the security *Realm*. If **true**, this valve uses cached security credentials (username and password) to reauthenticate to the JBoss Web security *Realm* each request associated with an SSO session. If **false**, the valve can itself authenticate requests based on the presence of a valid SSO cookie, without rechecking with the *Realm*. Setting to **true** can allow web applications with different **security-domain** configurations to share an SSO. Default is **false**.

21.4.2. SSO Behavior

The user will not be challenged as long as they access only unprotected resources in any of the web applications on the virtual host.

Upon access to a protected resource in any web application, the user will be challenged to authenticate, using the login method defined for the web application.

Once authenticated, the roles associated with this user will be utilized for access control decisions across all of the associated web applications, without challenging the user to authenticate themselves to each application individually.

If the web application invalidates a session (by invoking the **javax.servlet.http.HttpSession.invalidate()** method), the user's sessions in all web applications will be invalidated.

A session timeout does not invalidate the SSO if other sessions are still valid.

21.4.3. Limitations

There are a number of known limitations to this Tomcat valve-based SSO implementation:

- It is only useful within a cluster of JBoss servers; SSO does not propagate to other resources.
- It requires the use of container managed authentication (via the **login-config** element in **web.xml**)
- Requires cookies. SSO is maintained via a cookie and URL rewriting is not supported.
- Unless **requireReauthentication** is set to **true**, all web applications configured for the same SSO valve must share the same JBoss Web **Realm** and JBoss Security **security-domain**. This means:
 - In **server.xml** you can nest the **Realm** element inside the **Host** element (or the surrounding **Engine** element), but not inside a **context.xml** packaged with one of the involved web applications.
 - The **security-domain** configured in **jboss-web.xml** or **jboss-app.xml** must be consistent for all of the web applications.

- Even if you set **requireReauthentication** to **true** and use a different **security-domain** (or, less likely, a different **Realm**) for different web applications, the varying security integrations must all accept the same credentials (e.g. username and password).

21.4.4. Configuring the Cookie Domain

As noted above the SSO valve supports a **cookieDomain** configuration attribute. This attribute allows configuration of the SSO cookie's domain (that is, the set of hosts to which the browser will present the cookie). By default the domain is `"/"`, meaning the browser will only present the cookie to the host that issued it. The **cookieDomain** attribute allows the cookie to be scoped to a wider domain.

For example, suppose we have a case where two apps, with URLs **http://app1.xyz.com** and **http://app2.xyz.com**, that wish to share an SSO context. These applications could be running on different servers in a cluster or the virtual host with which they are associated could have multiple aliases. This can be supported with the following configuration:

```
<Valve className="org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn"
        cookieDomain="xyz.com" />
```

Chapter 22. Clustered Deployment Options

22.1. Clustered Singleton Services

A clustered singleton service (also known as a 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.

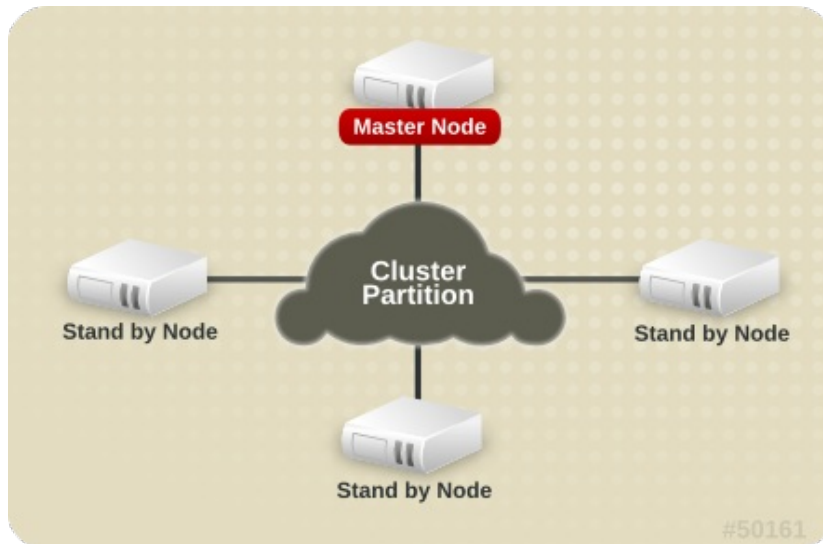


Figure 22.1. Topology before the Master Node fails

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.

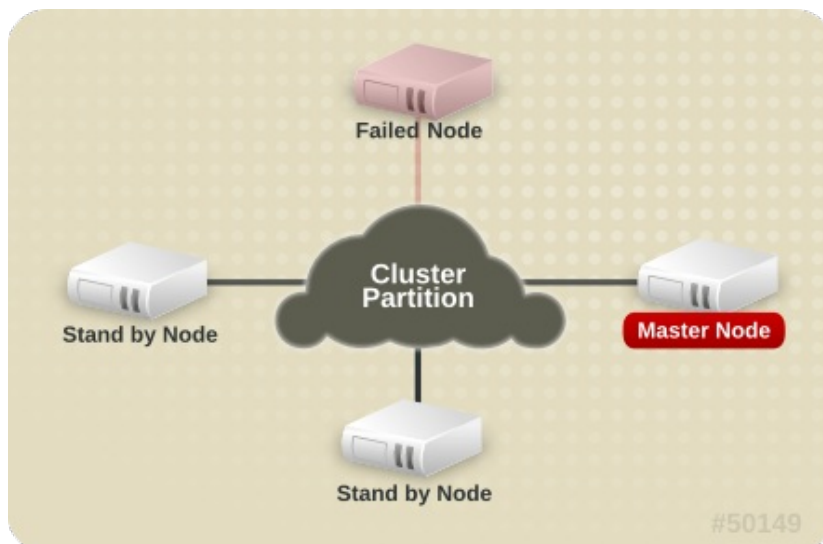


Figure 22.2. Topology after the Master Node fails

22.1.1. HASingleton Deployment Options

The JBoss Enterprise Web Platform 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.

22.1.1.1. 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/production/deploy-hasingleton** directory instead of in **deploy**. The **deploy-hasingleton** directory does not lie under **deploy** or **farm** directories, so its contents are not automatically deployed when an Enterprise Web Platform instance starts. Instead, deploying the contents of this directory is the responsibility of a special service, the **HASingletonDeployer** bean (which itself is deployed via the **deploy/deploy-hasingleton-jboss-beans.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 whichever 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 the complexity of your service's deployment, and the extent of startup activity in which it engages, this could take a while, during which time the service is not being provided.

22.1.1.2. POJO deployments using HASingletonController

If your service is a POJO (that is, 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 a POJO that we want to make an **HASingleton**. The only thing special about it is it needs to expose a public method that can be called when it should begin providing service, and another that can be called when it should stop providing service:

```
public interface HASingletonExampleMBean
{
    boolean isMasterNode();
}
```

```

public class HASingletonExample implements HASingletonExampleMBean
{
    private boolean isMasterNode = false;

    public boolean isMasterNode()
    {
        return isMasterNode;
    }

    public void startSingleton()
    {
        isMasterNode = true;
    }

    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-beans.xml**:

```

<deployment xmlns="urn:jboss:bean-deployer:2.0">
  <!-- This bean is an example of a clustered singleton -->
  <bean name="HASingletonExample"
class="org.jboss.ha.examples.HASingletonExample">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
      (name="jboss:service=HASingletonExample",
exposedInterface=org.jboss.ha.examples.HASingletonExampleMBean.class)</annotation>
  </bean>

  <bean name="ExampleHASingletonController"
class="org.jboss.ha.singleton.HASingletonController">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
      (name="jboss:service=ExampleHASingletonController",
exposedInterface=org.jboss.ha.singleton.HASingletonControllerMBean.class,
registerDirectly=true)</annotation>
    <property name="HAPartition"><inject bean="HAPartition"/></property>
    <property name="target"><inject bean="HASingletonExample"/></property>
    <property name="targetStartMethod">startSingleton</property>
    <property name="targetStopMethod">stopSingleton</property>
  </bean>
</deployment>

```

This creates a clustered singleton service.

The primary advantage of this approach over **deploy-ha-singleton** is 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.

Although not demonstrated in the example above, the **HASingletonController** can support an optional argument for either or both of the target start and stop methods. These are specified using the **targetStartMethodArgument** and **TargetStopMethodArgument** properties, respectively.

Currently, only string values are supported.

22.1.1.3. HASingleton deployments using a Barrier

Services deployed normally inside **deploy** or **farm** that should be started or stopped whenever the content of **deploy-hasingleton** is deployed or undeployed, (that is, whenever the current node becomes the master), need only specify a dependency on the Barrier service:

```
<depends>jboss.ha:service=HASingletonDeployer,type=Barrier</depends>
```

The way it works is that a **BarrierController** is deployed along with the **HASingletonDeployer** 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 bean 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 also destroyed.

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 (that is, any **create()** method invoked) on all nodes, but only started on the master node. This is different with the **deploy-hasingleton** approach, which will only deploy (instantiate and 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 starting and stopping of dependent services, but not their destruction, which happens only when the **BarrierController** is itself destroyed and undeployed. Thus using the **Barrier** to control services that need to be *destroyed* as part of their normal *undeploy* operation will not have the desired effect.

22.1.2. 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 identify whether it is now the *master node*. How is this done?

For each member of the cluster, the **HAPartition** service 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 receives 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.

Say we have a four node cluster with nodes **A**, **B**, **C**, and **D**. The current view can be expressed as **{A, B, C, D}**.

Now, imagine that a singleton service (that is, an **HASingletonController**) named **Foo** is deployed on all nodes on the cluster except **B**. The **HAPartition** service maintains a registry of services deployed across the cluster, 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}**.

Whenever the cluster topology of the **Foo** service changes, the **HAPartition** service involves a callback on **Foo**, notifying it of the new topology. So when **Foo** started on node **D**, the **Foo** service running on **A, C** and **D** all received callbacks informing them that the new view for **Foo** was **{A, C, D}**. This callback gives each node enough information to decide independently whether it is now the master node. The **Foo** service on each node uses the **HAPartition**'s **HASingletonElectionPolicy** to determine whether it is the master, as explained in [Section 22.1.2.1, “HA singleton election policy”](#).

If **A** fails or shuts down, **Foo** on **C** and **D** would receive 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 receive a callback with a new view for **Foo** of **{C, D, A}**. **C** would remain the master — there is no reason that **A** in particular should be reassigned the master role simply because it previously held that role.

22.1.2.1. HA singleton election policy

The **HASingletonElectionPolicy** object is responsible for electing a master node from a list of available nodes, on behalf of an **HASingleton**, following a change in cluster topology.

```
public interface HASingletonElectionPolicy
{
    ClusterNode elect(List<ClusterNode> nodes);
}
```

JBoss Enterprise Web Platform ships with two election policies:

HASingletonElectionPolicySimple

This policy selects a master node based relative age. The desired age is configured via the **position** property, which corresponds to the index in the list of available nodes. **position = 0**, the default, refers to the oldest node; **position = 1**, refers to the second oldest, etc. **position** can also be negative to indicate youth. It is therefore useful to imagine the list of available nodes as a circular linked list. **position = -1**, refers to the youngest node; **position = -2**, refers to the second youngest, etc.

```
<bean class="org.jboss.ha.singleton.HASingletonElectionPolicySimple">
  <property name="position">-1</property>
</bean>
```

PreferredMasterElectionPolicy

This policy extends **HASingletonElectionPolicySimple**, allowing the configuration of a preferred node. The **preferredMaster** property, specified as **host:port** or **address:port**, identifies a specific node that should become master, if available. If the preferred node is not available, the election policy will behave as described above.

```
<bean class="org.jboss.ha.singleton.PreferredMasterElectionPolicy">
  <property name="preferredMaster">server1:12345</property>
</bean>
```

22.2. Farming Deployment

The easiest way to deploy an application into the cluster is to use the farming service. Using the farming service, you can deploy an application (EAR, WAR, or SAR; either an archive file or in exploded form) to the **production/farm/** directory of any cluster member and the application will be automatically duplicated across all nodes in the same cluster. If a 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 a running clustered server node's **farm/** directory, the application will be undeployed locally and then removed from all other clustered server nodes' **farm/** directories (triggering undeployment).

Farming is enabled by default in the **production** configuration in JBoss Enterprise Web Platform and thus requires no manual setup. The required **farm-deployment-jboss-beans.xml** and **timestamps-jboss-beans.xml** configuration files are located in the **deploy/cluster** directory. If you want to enable farming in a custom configuration, simply copy these files to the corresponding JBoss **deploy** directory: **\$JBoss_HOME/server/\$CUSTOM_CONFIG/deploy/cluster**. Make sure that your custom configuration has clustering enabled.

While there is little need to customize the farming service, it can be customized via the **FarmProfileRepositoryClusteringHandler** bean, whose properties and default values are listed below:

```
<bean name="FarmProfileRepositoryClusteringHandler"
      class="org.jboss.profileservice.cluster.repository.
      DefaultRepositoryClusteringHandler">

  <property name="partition"><inject bean="HAPartition"/></property>
  <property name="profileDomain">default</property>
  <property name="profileServer">default</property>
  <property name="profileName">farm</property>
  <property name="immutable">false</property>
  <property name="lockTimeout">60000</property><!-- 1 minute -->
  <property name="methodCallTimeout">60000</property><!-- 1 minute -->
  <property name="synchronizationPolicy"><inject
  bean="FarmProfileSynchronizationPolicy"/></property>
</bean>
```

partition

Required to inject the **HAPartition** service that the farm service uses for intra-cluster communication.

profileDomain, profileServer, profileName

Used to identify the profile for which this handler is intended.

immutable

Indicates whether this handler allows a node to push content changes to the cluster. A value of **true** is equivalent to setting **synchronizationPolicy** to **org.jboss.system.server.profileservice.repository.clustered.sync.ImmutableSynchronizationPolicy**.

lockTimeout

Defines the number of milliseconds to wait for cluster-wide lock acquisition.

methodCallTimeout

Defines the number of milliseconds to wait for invocations on remote cluster nodes.

synchronizationPolicy

Determines how to handle content addition, reincarnation, updates and removals from nodes attempting to join the cluster or from cluster merges. The policy is consulted on the *authoritative* node (the master node for the service on the cluster). *Reincarnation* describes a situation where a newly started node may contain an application starting node in its **farm** directory that was previously removed by the farming service, but may still exist on the starting node if it was

not running when the removal took place.

The default synchronization policy is defined as follows:

```
<bean name="FarmProfileSynchronizationPolicy"
      class="org.jboss.profileservice.cluster.repository.
      DefaultSynchronizationPolicy">
  <property name="allowJoinAdditions"><null/></property>
  <property name="allowJoinReincarnations"><null/></property>
  <property name="allowJoinUpdates"><null/></property>
  <property name="allowJoinRemovals"><null/></property>
  <property name="allowMergeAdditions"><null/></property>
  <property name="allowMergeReincarnations"><null/></property>
  <property name="allowMergeUpdates"><null/></property>
  <property name="allowMergeRemovals"><null/></property>
  <property name="developerMode">false</property>
  <property name="removalTrackingTime">2592000000</property><!-- 30 days -
  ->
  <property name="timestampService"><inject
  bean="TimestampDiscrepancyService"/></property>
</bean>
```

- **allow[Join|Merge][Additions|Reincarnations|Updates|Removals]** define fixed responses to requests to allow additions, reincarnations, updates, or removals from joined or merged nodes.
- **developerMode** enables a lenient synchronization policy that allows all changes. Enabling developer mode is equivalent to setting each of the above properties to **true** and is intended for development environments.
- **removalTrackingTime** defines the number of milliseconds for which this policy should remember removed items, for use in detecting reincarnations.
- **timestampService** estimates and tracks discrepancies in system clocks for current and past members of the cluster. Default implementation is defined in **timestamps-jboss-beans.xml**.

Chapter 23. JGroups Services

JGroups provides the underlying group communication support for JBoss Enterprise Web Platform clusters. The interaction of clustered services with JGroups was covered in [Section 17.1, “Group Communication with JGroups”](#). This chapter focuses on the details of this interaction, with particular attention to configuration details and troubleshooting tips.

This chapter is not intended as complete JGroups documentation. If you want to know more about JGroups, you can consult:

- The JGroups project documentation at <http://jgroups.org/ug.html>
- The JGroups wiki pages at jboss.org, rooted at <https://www.jboss.org/community/wiki/JGroups>

The first section of this chapter covers the many JGroups configuration options in detail. JBoss Enterprise Web Platform ships with a set of default JGroups configurations. Most applications will work with the default configurations out of the box. You will only need to edit these configurations when you deploy an application with special network or performance requirements.

23.1. Configuring a JGroups Channel's Protocol Stack

The JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster. Communication occurs over a communication channel. The channel built up from a stack of network communication *protocols*, each of which is responsible for adding a particular capability to the overall behavior of the channel. Key capabilities provided by various protocols include transport, cluster discovery, message ordering, lossless message delivery, detection of failed peers, and cluster membership management services.

[Figure 23.1, “Protocol stack in JGroups”](#) shows a conceptual cluster with each member's channel composed of a stack of JGroups protocols.

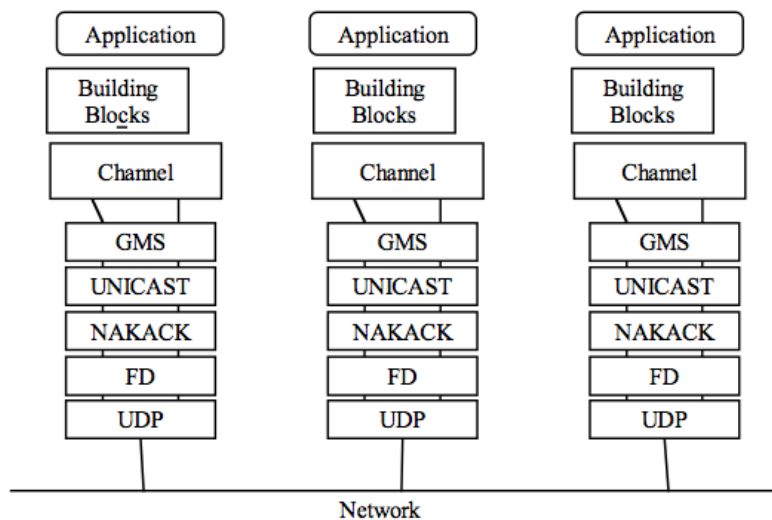


Figure 23.1. Protocol stack in JGroups

This section of the chapter covers some of the most commonly used protocols, according to the type of behaviour they add to the channel. We discuss a few key configuration attributes exposed by each protocol, but since these attributes should be altered only by experts, this chapter focuses on familiarizing users with the purpose of various protocols.

The JGroups configurations used in JBoss Enterprise Web Platform appear as nested elements in the **\$JBoss_HOME/server/production/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** file. This file is parsed by the **ChannelFactory**

service, which uses the contents to provide correctly configured channels to the clustered services that require them. See [Section 17.1.1, “The Channel Factory Service”](#) for more on the **ChannelFactory** service.

The following is an example protocol stack configuration from **jgroups-channelfactory-stacks.xml**:

```

<stack name="udp-async"
    description="Same as the default 'udp' stack above, except message
bundling
    is enabled in the transport protocol
(enable_bundling=true).
    Useful for services that make high-volume asynchronous
    RPCs (e.g. high volume JBoss Cache instances configured
    for REPL_ASYNC) where message bundling may improve
performance.">
    <config>
        <UDP
            singleton_name="udp-async"
            mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
            mcast_addr="${jboss.partition.udpGroup:228.11.11.11}"
            tos="8"
            ucast_rcv_buf_size="20000000"
            ucast_send_buf_size="640000"
            mcast_rcv_buf_size="25000000"
            mcast_send_buf_size="640000"
            loopback="true"
            discard_incompatible_packets="true"
            enable_bundling="true"
            max_bundle_size="64000"
            max_bundle_timeout="30"
            ip_ttl="${jgroups.udp.ip_ttl:2}"
            thread_naming_pattern="cl"
            timer.num_threads="12"
            enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"
            diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
            diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

            thread_pool.enabled="true"
            thread_pool.min_threads="8"
            thread_pool.max_threads="200"
            thread_pool.keep_alive_time="5000"
            thread_pool.queue_enabled="true"
            thread_pool.queue_max_size="1000"
            thread_pool.rejection_policy="discard"

            oob_thread_pool.enabled="true"
            oob_thread_pool.min_threads="8"
            oob_thread_pool.max_threads="200"
            oob_thread_pool.keep_alive_time="1000"
            oob_thread_pool.queue_enabled="false"
            oob_thread_pool.rejection_policy="discard"/>
        <PING timeout="2000" num_initial_members="3"/>
        <MERGE2 max_interval="100000" min_interval="20000"/>
        <FD_SOCK/>
        <FD timeout="6000" max_tries="5" shun="true"/>
        <VERIFY_SUSPECT timeout="1500"/>
        <BARRIER/>
        <pbcast.NAKACK use_mcast_xmit="true" gc_lag="0"
            retransmit_timeout="300,600,1200,2400,4800"
            discard_delivered_msgs="true"/>
        <UNICAST timeout="300,600,1200,2400,3600"/>
        <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
            max_bytes="400000"/>
        <VIEW_SYNC avg_send_interval="10000"/>
        <pbcast.GMS print_local_addr="true" join_timeout="3000"
            shun="true"
            view_bundling="true"
            view_ack_collection_timeout="5000"
            resume_task_timeout="7500"/>
        <FC max_credits="2000000" min_threshold="0.10"

```

```

        ignore_synchronous_response="true"/>
        <FRAG2 frag_size="60000"/>
        <!-- pbcaster.STREAMING_STATE_TRANSFER/ -->
        <pbcaster.STATE_TRANSFER/>
        <pbcaster.FLUSH timeout="0" start_flush_timeout="10000"/>
    </config>
</stack>

```

The **<config>** element contains all the configuration data for JGroups. This information is used to configure a JGroups *channel*, which is conceptually similar to a socket, and manages communication between peers in a cluster. Each element within the **<config>** element defines a particular JGroups *protocol*. Each protocol performs one function. The combination of these functions defines the characteristics of the channel as a whole. The next few sections describe common protocols and explain the options available to each.

23.1.1. Common Configuration Properties

The following property is exposed by all of the JGroups protocols discussed below:

- **stats** whether the protocol should gather runtime statistics on its operations that can be exposed via tools like the AS's JMX console or the JGroups Probe utility. What, if any, statistics are gathered depends on the protocol. Default is **true**.



Note

Past versions of JGroups used **down_thread** and **up_thread** attributes. These attributes are no longer used. A **WARN** message will be written to the server log if they are configured for any protocol.

23.1.2. Transport Protocols

The transport protocols send and receive messages to and from the network. They also manage the thread pools used to deliver incoming messages to addresses higher in the protocol stack. JGroups supports **UDP**, **TCP** and **TUNNEL** as transport protocols.



Note

The **UDP**, **TCP**, and **TUNNEL** protocols are mutually exclusive. You can only have one transport protocol in each JGroups **Config** element

23.1.2.1. UDP configuration

UDP is the preferred transport protocol for JGroups. UDP uses multicast (or, in an unusual configuration, 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
  singleton_name="udp-async"
  mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
  mcast_addr="${jboss.partition.udpGroup:228.11.11.11}"
  tos="8"
  ucast_recv_buf_size="20000000"
  ucast_send_buf_size="640000"
  mcast_recv_buf_size="25000000"
  mcast_send_buf_size="640000"
  loopback="true"
  discard_incompatible_packets="true"
  enable_bundling="true"
  max_bundle_size="64000"
  max_bundle_timeout="30"
  ip_ttl="${jgroups.udp.ip_ttl:2}"
  thread_naming_pattern="cl"
  timer.num_threads="12"
  enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"
  diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
  diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

  thread_pool.enabled="true"
  thread_pool.min_threads="8"
  thread_pool.max_threads="200"
  thread_pool.keep_alive_time="5000"
  thread_pool.queue_enabled="true"
  thread_pool.queue_max_size="1000"
  thread_pool.rejection_policy="discard"

  oob_thread_pool.enabled="true"
  oob_thread_pool.min_threads="8"
  oob_thread_pool.max_threads="200"
  oob_thread_pool.keep_alive_time="1000"
  oob_thread_pool.queue_enabled="false"
  oob_thread_pool.rejection_policy="discard"/>

```

JGroups transport configurations have a number of attributes available. First we look at the attributes available to the **UDP** protocol, followed by the attributes that are also used by the **TCP** and **TUNNEL** transport protocols.

The attributes particular to the **UDP** protocol are:

ip_multicast

Specifies whether to use IP multicasting. The default value is **true**. If set to **false**, multiple unicast packets will be sent instead of one multicast packet. Any packet sent via **UDP** is sent as a UDP datagram.

mcast_addr

Specifies the multicast address (class D) for communicating with the group. The value of system property **jboss.partition.udpGroup** is used as the value for this attribute, if set. (Set this property at startup with the **-u** command line switch.) If omitted, the default value is **228.11.11.11**.

mcast_port

Specifies the port to use for multicast communication with the group. If omitted, the default value is **45688**. See [Section 23.6.2, “Isolating JGroups Channels”](#) to ensure JGroups channels are properly isolated from each other.

mcast_send_buf_size, mcast_recv_buf_size, ucast_send_buf_size, ucast_recv_buf_size

Define the socket send and receive buffer sizes that JGroups requests from the operating system. A large buffer helps prevent packets being dropped due to buffer overflow. However, socket buffer sizes are limited at the operating system level, and may require operating system-level configuration. See [Section 23.6.2.3, “Improving UDP Performance by Configuring OS UDP Buffer Limits”](#) for details.

bind_port

Specifies the port that binds the unicast receive socket. The default value is **0** (use an ephemeral port).

port_range

Specifies the range of ports to try if the **bind_port** is not available. The default is **1**, which specifies that only **bind_port** will be tried.

ip_ttl

Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped.

tos

Specifies the traffic class for sending unicast and multicast datagrams.

The following attributes are common to all transport protocols:

singleton_name

The unique name of this transport protocol configuration. The **ChannelFactory** uses this to share transport protocol instances between different channels with the same transport protocol configuration. See [Section 17.1.2, “The JGroups Shared Transport”](#) for details.

bind_addr

Specifies the interface that sends and receives messages. By default, JGroups uses the value of system property **jgroups.bind_addr**. This can be set with the **-b** command line switch. See [Section 23.6, “Other Configuration Issues”](#) for more about binding JGroups sockets.

receive_on_all_interfaces

Specifies that this node should listen on all interfaces for multicasts. The default value is **false**. Specifying this overrides the **bind_addr** property for receiving multicasts. (It does not override **bind_addr** for sending multicasts.)

send_on_all_interfaces

Specifies that the node send UDP packets via all available network interface controllers (NICs). The same multicast message will be sent multiple times so use this attribute with care.

receive_interfaces

A comma-separated list of interfaces on which to receive multicasts, for example, **192.168.5.1,eth1,127.0.0.1**. The multicast receive socket will listen on all listed interfaces.

send_interfaces

A comma-separated list of interfaces on which to send multicasts, for example, **192.168.5.1,eth1,127.0.0.1**. The multicast sender socket will send on all listed interfaces. The same multicast message will be sent multiple times, so use with care.

enable_bundling

Specifies whether to enable message bundling. If **true**, the transport protocol queues outgoing messages until **max_bundle_size** bytes have accumulated or **max_bundle_time** milliseconds have elapsed. The transport protocol then bundles queued messages into one large message and sends it. Messages are unbundled at the receiver. The default value is **false**.

Message bundling can improve performance where senders do not block waiting for a response from recipients, for example, a JBoss Cache instance configured for **REPL_ASYNC**. It adds latency to applications where senders must block waiting for responses, so it is not recommended in some circumstances, for example, a JBoss Cache instance configured for **REPL_SYNC**.

loopback

Specifies whether the thread should carry a message back up the stack for delivery. (Messages sent to the group are always sent to the sending node as well.) If **false**, a message delivery pool thread is used instead of the sending thread. **false** is the default, but **true** is recommended to ensure that a channel receives its own messages should the network interface fail.

discard_incompatible_packets

Specifies whether to discard packets sent by peers that use a different version of JGroups. If **true**, messages tagged with a different JGroups version are silently discarded. If **false**, a warning is logged. *In neither case will the message be delivered.* The default is **false**.

enable_diagnostics

Specifies that the transport should open a multicast socket on **diagnostics_addr** and **diagnostics_port** to listen for diagnostic requests sent by the JGroups **Probe** utility.

thread_pool

The various **thread_pool** attributes configure the behavior of the pool of threads JGroups uses to carry incoming messages up the stack. They provide the constructor arguments for an instance of **java.util.concurrent.ThreadPoolExecutorService**.

```
thread_pool.enabled="true"
thread_pool.min_threads="8"
thread_pool.max_threads="200"
thread_pool.keep_alive_time="5000"
thread_pool.queue_enabled="true"
thread_pool.queue_max_size="1000"
thread_pool.rejection_policy="discard"
```

Here, the pool will have a minimum or *core size* of 8 threads, and a maximum size of 200. If more than 8 pool threads have been created, a thread returning from carrying a message will wait for up to 5000 milliseconds to be assigned a new message to carry, after which it will terminate. If no thread is available to be assigned a new message, the (separate) thread reading messages from the socket will place messages in a queue (**thread_pool.queue_enabled**). This queue will hold up to 1000 messages. If the queue is

full, the thread reading messages from the queue will discard new messages.

oob_thread_pool

The various **oob_thread_pool** attributes are similar to the **thread_pool** attributes in that they configure an instance of `java.util.concurrent.ThreadPoolExecutorService` used to carry messages up the protocol stack. In this case, the pool carries a special type of message known as an Out-of-Band (OOB) message.

OOB messages are exempt from the ordered delivery requirements of protocols such as NAKACK and UNICAST, and can be delivered up the stack even if messages are queued ahead of them. OOB messages are often used internally by JGroups protocols. They can also be used by applications, for example, when JBoss Cache is in **REPL_SYNC** mode, it uses OOB messages for the second phase of its two-phase commit protocol.

23.1.2.2. 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, like so:

```
<TCP singleton_name="tcp"
    start_port="7800" end_port="7800"/>
```

The following attributes are specific to the **TCP** element:

start_port, end_port

Define the range of TCP ports to which the server should bind. The server socket is bound to the first available port, beginning with **start_port**. If no available port is found before the server reaches **end_port**, the server throws an exception. If no **end_port** is provided, or **end_port** is lower than **start_port**, no upper limit is applied to the port range. If **start_port** is equal to **end_port**, JGroups is forced to use the port specified, and will fail if it is unavailable. The default value is **7800**. If set to **0**, the operating system will select a port. (This works only for MPING or TPCGOSSIP. TCPPING requires that nodes and their required ports are listed.)

bind_port

Acts as an alias for **start_port**. If configured internally, sets **start_port**.

recv_buf_size, send_buf_size

Define receive and send puffer sizes. A large buffer size means packets are less likely to be 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 the interval in milliseconds at which to run the reaper. If both values are **0**, no reaping will be done. If either value is greater than zero, reaping will be enabled. The reaper is disabled by default.

sock_conn_timeout

Specifies the maximum time in milliseconds for socket creation. When a peer hangs during initial discovery, instead of waiting forever, other members will be pinged after this timeout period. This reduces the chances of not finding any members at all. The default value 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 value is **true**.

external_addr

Specifies an external IP address to broadcast to other group members (if not the local address). This is useful for Network Address Translation (NAT). Say a node on a private network exists behind a firewall, but can only be routed to via an externally visible address, not the local address to which it is bound. The node can be configured to broadcast its external address while remaining bound to the local one. This lets you avoid using the TUNNEL protocol and a central gossip router. Without setting the **external_addr**, the node behind the firewall broadcasts 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 value is **true**.

tcp_nodelay

Specifies **TCP_NODELAY**. By default, TCP *nagles* messages (bundles smaller messages together into a larger message). To invoke synchronous cluster method calls, we must disable nagling in addition to disabling message bundling. To do this, set **tcp_nodelay** to **true** and **enable_bundling** to **false**. The default value for **tcp_nodelay** is **false**.

**Note**

All of the attributes common to all protocols discussed in the UDP protocol section also apply to TCP.

23.1.2.3. TUNNEL configuration

The TUNNEL protocol uses an external router known as the **GossipRouter** to send messages. Each node must register with this router. All messages are sent to the router and forwarded to their destinations. The TUNNEL approach can be used to set up communication with nodes behind firewalls. A node can establish a TCP connection to the **GossipRouter** through the firewall via port 80. This connection is also used by the router to send messages to nodes behind the firewall, since 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** sub-element in the JGroups **Config** element, like so:

```
<TUNNEL singleton_name="tunnel"
        router_port="12001"
        router_host="192.168.5.1"/>
```

The available attributes in the **TUNNEL** element are listed below.

router_host

Specifies the host on which the **GossipRouter** runs.

router_port

Specifies the port on which the **GossipRouter** listens.

reconnect_interval

Specifies the interval in milliseconds for which **TUNNEL** will attempt to connect to the **GossipRouter** if the connection is not established.

**Note**

All of the attributes common to all protocols discussed in the UDP protocol section also apply to **TUNNEL**.

23.1.3. Discovery Protocols

When a channel on a node first connects, it must determine which other nodes are running compatible channels, and which of these nodes is currently acting as the *coordinator* (the node responsible for letting new nodes join the group). Discovery protocols are used to find active nodes in the cluster and to determine which is the coordinator. This information is then provided to the group membership protocol (GMS), which communicates with the coordinator's GMS to add the newly-connecting node to the group. (For more information about group membership protocols, see [Section 23.1.6, "Group Membership \(GMS\)"](#).)

Discovery protocols also assist merge protocols (see [Section 23.5, "Merging \(MERGE2\)"](#)) to detect cluster-split situations.

The discovery protocols sit on top of the transport protocol, so you can choose to use different discovery protocols depending on your transport protocol. These are also configured as sub-elements in the JGroups **<Config>** element.

23.1.3.1. 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** is the coordinator's address, and **A** is the node's 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 no node responds, it assumes it is the first member of a group.

Here is an example PING configuration for IP multicast.

```
<PING timeout="2000"
    num_initial_members="3"/>
```

Here is another example PING configuration for contacting a Gossip Router.

```
<PING gossip_host="localhost"
    gossip_port="1234"
    timeout="2000"
    num_initial_members="3"/>
```

The available attributes in the **PING** element are listed below.

timeout

Specifies the maximum number of milliseconds to wait for any responses. The default value is **3000**.

num_initial_members

Specifies the maximum number of responses to wait for unless the **timeout** has expired. The default value is **2**.

gossip_host

Specifies the host on which the **GossipRouter** is running.

gossip_port

Specifies the port on which the **GossipRouter** is listening.

gossip_refresh

Specifies the interval, in milliseconds, for the lease from the **GossipRouter**. The default value is **20000**.

initial_hosts

A comma-separated list of addresses to ping for discovery, for example, **host1[12345],host2[23456]**.

If both **gossip_host** and **gossip_port** are defined, the cluster uses the **GossipRouter** for the initial discovery. If **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** period has elapsed or **num_initial_members** responses have been received.

23.1.3.2. TCPGOSSIP

The **TCPGOSSIP** protocol only works with a **GossipRouter**. It works similarly to the **PING** protocol configuration with valid **gossip_host** and **gossip_port** attributes. It works on top of both UDP and TCP transport protocols, like so:

```
<TCPGOSSIP timeout="2000"
  num_initial_members="3"
  initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"/>
```

The available attributes in the **TCPGOSSIP** element are listed below.

timeout

Specifies the maximum number of milliseconds to wait for any responses. The default value is **3000**.

num_initial_members

Specifies the maximum number of responses to wait for unless **timeout** has expired. The default value is **2**.

initial_hosts

A comma-separated list of addresses for **GossipRouters** to register with, for example, **host1[12345],host2[23456]**.

23.1.3.3. TCPING

The **TCPING** protocol takes a set of known members and pings them for discovery. This is a static configuration. It works on top of TCP. Here is an example of the **TCPING** configuration sub-element in the JGroups **Config** element.

```
<TCPING timeout="2000"
  num_initial_members="3"/
  initial_hosts="hosta[2300], hostb[3400], hostc[4500]"
  port_range="3">
```

The available attributes in the **TCPING** element are listed below.

timeout

Specifies the maximum number of milliseconds to wait for any responses. The default value is **3000**.

num_initial_members

Specifies the maximum number of responses to wait for unless the **timeout** has expired. The default value is **2**.

initial_hosts

A comma-separated list of addresses to ping for discovery, for example, **host1[12345],host2[23456]**.

port_range

Specifies the number of consecutive ports to be probed when getting the initial membership, starting from the port specified in the **initial_hosts** parameter. Given the values of **port_range** and **initial_hosts** given in the example code, the **TCPING** layer will try to connect to **hosta:2300, hosta:2301, hosta:2302, hostb:3400, hostb:3401, hostb:3402, hostc:4500, hostc:4501** and **hostc:4502**. The configuration options allow for multiple nodes on the same host to be pinged.

23.1.3.4. 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 **TCPING**, which has to list all group members explicitly, but **MPING** does not 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 **TCPING** or require an external **GossipRouter**.

```
<MPING timeout="2000"
  num_initial_members="3"
  bind_to_all_interfaces="true"
  mcast_addr="228.8.8.8"
  mcast_port="7500"
  ip_ttl="8"/>
```

The available attributes in the **MPING** element are listed below.

timeout

Specifies the maximum number of milliseconds to wait for any responses. The default value is **3000**.

num_initial_members

Specifies the maximum number of responses to wait for unless **timeout** has expired. The default value is **2**.

bind_addr

Specifies the interface on which to send and receive multicast packets.

bind_to_all_interfaces

Overrides the **bind_addr** value and uses all interfaces in multihome nodes.

mcast_addr

Specifies the multicast address for joining a cluster. If omitted, the default is **228.8.8.8**.

mcast_port

Specifies the multicast port number. If omitted, the default is **45566**.

ip_ttl

Specifies the *time to live* (TTL) for IP multicast packets. TTL is the common term in multicast networking, but the value actually refers to how many network hops a packet will be allowed to travel before networking equipment drops it.

23.1.4. 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. If the node is still considered dead after this phase, 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.

23.1.4.1. FD

FD is a failure detection protocol based on *heartbeat* messages. This protocol requires each node to periodically send messages to its neighbour to check that the neighbour is alive. 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. If the node is still considered dead after this check, the group coordinator updates the cluster's view. Here is an example FD configuration:

```
<FD timeout="6000"
    max_tries="5"
    shun="true"/>
```

The available attributes in the **FD** element are listed below.

timeout

Specifies the maximum number of milliseconds to wait for a response to the heartbeat messages. The default value is **3000**.

max_tries

Specifies the number of heartbeat messages that a node can fail to reply to before the node is suspected. The default value is **2**.

shun

Specifies whether a failed node will be shunned. Once shunned, the node will be expelled from the cluster even if it is later revived. The shunned node would have to rejoin the cluster through the discovery process. You can configure JGroups so that shunning leads to automatic rejoins and state transfer (the default behavior).

**Note**

Normal node traffic is considered proof of life, so heartbeat messages are sent only when there is no normal traffic to the node for some time.

23.1.4.2. 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 an 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/>
```

The available attributes in the **FD_SOCKET** element are listed below.

bind_addr

Specifies the interface to which the server socket should bind. If **-Djgroups.bind_address** system property is defined, this XML value will be ignored. This behavior can be reversed by setting the **-Djgroups.ignore.bind_addr=true** system property.

23.1.4.3. VERIFY_SUSPECT

This protocol verifies that a suspected member is dead by pinging them a second time. This verification is performed by the coordinator of the cluster. The suspected member is dropped from the cluster group if confirmed dead. The aim of this protocol is to minimize false suspicions. See the following code for an example:

```
<VERIFY_SUSPECT timeout="1500"/>
```

The available attributes in the **VERIFY_SUSPECT** element are listed below.

timeout

Specifies how long to wait for a response from the suspected member before considering it dead.

23.1.4.4. FD versus FD_SOCKET

FD and **FD_SOCKET** do not individually provide a solid failure detection layer. Their differences are outlined below to show how they complement each other.

FD

- An overloaded machine might be slow in sending heartbeat responses.
- A member will become suspected when suspended in a debugger or profiler.
- Low timeouts lead to a higher probability of false suspicions and higher network traffic.
- High timeouts will not detect and remove crashed members for a long period of time.

FD_SOCKET

- Suspension in a debugger does not mean a member will become suspected because the TCP connection remains open.
- High load is not a problem for the same reason.
- Members will be suspected only when the TCP connection breaks, so hung members will not be detected.
- A crashed switch will not be detected until the connection encounters the TCP timeout (between two and twenty minutes, depending on TCP/IP stack implementation).

A failure detection layer aims to report real failures and avoid reporting false suspicions. Two methods of achieving this are outlined in the following paragraphs.

By default, JGroups configures the **FD_SOCKET** socket with **KEEP_ALIVE**, which means that TCP sends a heartbeat to a socket that has received no traffic in two hours. If a host or immediate switch or router crashed without closing the TCP connection properly, it would be detected shortly after two hours. This is better than never closing the connection (where **KEEP_ALIVE** is off), but may not be helpful. The first solution, therefore, is to lower the timeout value for **KEEP_ALIVE**. This is a kernel-wide value on most operating systems and therefore affects all TCP sockets.

Alternatively, you can combine **FD_SOCKET** and **FD**. The **timeout** in **FD** can be set such that it is much lower than the TCP **timeout**. This can be configured on a per-process basis. **FD_SOCKET** generates a **SUSPECT** message if the socket closes abnormally, but in the case of a crashed switch or host, **FD** ensures that the socket is eventually closed, and a suspect message generated.

The following code shows how the two could be combined:

```
<FD_SOCKET/>
<FD timeout="6000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500"/>
```

This code suspects a member when the socket to its neighbour has been closed abnormally (for example, in a process crash, since the operating system closes all sockets). However, if a host or switch crashes, the sockets would not be closed. As a secondary line of defense, **FD** suspects the neighbour after **50** seconds. Note that if you use this example code and your system is stopped in a debugging breakpoint, the node you are debugging will be suspected after the specified fifty seconds.

Combining **FD** and **FD_SOCKET** provides a solid failure detection layer. This technique is used across the JGroups configurations included in JBoss Enterprise Web Platform.

23.1.5. Reliable Delivery Protocols

Reliable delivery protocols within the JGroups stack ensure that data packets are delivered in the correct order (FIFO) to the destination node. The basis for reliable message delivery is positive and negative delivery acknowledgments: respectively, **ACK** and **NAK**. In **ACK** mode, the sender resends the message until acknowledgement is received. In **NAK** mode, the receiver requests retransmission when it discovers a gap.

23.1.5.1. UNICAST

The **UNICAST** protocol is used for unicast messages. It uses **ACK**. It is configured as a sub-element under the JGroups **Config** element. **UNICAST** is not required for JGroups stacks configured with the TCP transport protocol, since TCP guarantees FIFO delivery of unicast messages. The following is an example of **UNICAST** protocol:

```
<UNICAST timeout="300,600,1200,2400,3600"/>
```

There is only one configurable attribute in the **UNICAST** element.

timeout

Specifies the retransmission timeout in milliseconds. For example, if the timeout is "**100,200,400,800**", the sender resends the message if it has not received an **ACK** after 100 milliseconds the first time, 200 milliseconds the second time, and so on. A low value for the first timeout allows for prompt retransmission of dropped message, but means that messages can be sent more than once if only the acknowledgement was not received before timeout. High values can improve performance if the network is tuned such that datagram loss is infrequent.

23.1.5.2. 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 tracks the sequence numbers to deliver the messages in order. When a gap in the sequence 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, like so:

```
<pbcast.NAKACK max_xmit_size="60000" use_mcast_xmit="false"
  retransmit_timeout="300,600,1200,2400,4800" gc_lag="0"
  discard_delivered_msgs="true"/>
```

The configurable attributes in the **pbcast.NAKACK** element are as follows.

retransmit_timeout

Specifies the retransmission timeout in milliseconds. This is same as the **timeout** attribute in the **UNICAST** protocol.

use_mcast_xmit

Determines whether the sender should send retransmit to the entire cluster rather than just the node requesting the retransmit. This is useful when the sender drops the packet, so that 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 delivered messages on receiver nodes. By default, we save all delivered messages. If the sender can resend the message, we can enable this option and discard delivered messages.

gc_lag

Specifies the number of messages to keep in memory for retransmission, even after the periodic cleanup protocol (see [Section 23.4, "Distributed Garbage Collection \(STABLE\)"](#)). The

default value is **20**.

23.1.6. Group Membership (GMS)

The group membership service in the JGroups stack maintains a list of active nodes. It handles 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, like so:

```
<pbcast.GMS print_local_addr="true"
  join_timeout="3000"
  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. Retries afterward.

join_retry_timeout

Specifies the maximum number of milliseconds to wait after a failed **JOIN** request to resubmit the request.

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 is not a member node.

disable_initial_coord

Specifies whether to prevent this node from becoming the cluster coordinator.

view_bundling

Specifies whether multiple **JOIN** or **LEAVE** requests arriving at the same time are bundled together and handled at the same time. This is more efficient than handling each request separately, as it sends only one new view.

23.1.7. Flow Control (FC)

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="20000000"
    min_threshold="0.10"
    ignore_synchronous_response="true"/>
```

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 more credits.

min_threshold

Specifies percentage value of the threshold. This attribute overrides **min_credits**.

ignore_synchronous_response

Specifies whether threads that have carried messages to the application are allowed to carry outgoing messages back down through flow control without blocking for credits. *Synchronous response* refers to these messages usually being responses to incoming RPC-type messages. We recommend setting this to **true** to help prevent certain deadlock scenarios.



Why is FC needed on top of TCP? TCP has its own flow control!

The **FC** element is required for group communication where group messages must be sent at the highest speed that the slowest receiver can handle.

Say we have a cluster, **{A,B,C,D}**. Node **D** is slow, and the other nodes are fast. When **A** sends a group message, it establishes the following TCP connections: **A-A**, **A-B**, **A-C**, and **A-D**.

A sends 100 million messages to the cluster. TCP's flow control applies to the connections between **A-B**, **A-C** and **A-D** individually, but not to **A-{B,C,D}**, where **{B,C,D}** is the group. It is therefore possible that nodes **A**, **B** and **C** receive the 100 million messages, but that node **D** will only receive one million messages. This is also the reason we need **NAKACK**, even though TCP does its own retransmission.

JGroups has to buffer all messages in memory in case the original sender dies and a node asks for retransmission of a message. Because all members buffer all messages they receive, they must occasionally purge *stable* messages (messages seen by all nodes). This is done with the **STABLE** protocol, which can be configured to run the stability protocol based on either time (for example, every fifty seconds) or size (every 400 kilobytes of data received).

In the example case, the slow node **D** will prevent the group from purging messages other than the one million seen by **D**. In most cases this leads to out-of-memory exceptions, so messages must be sent at a rate that the slowest receiver can handle.



So do I always need FC?

This depends on the application's use of the JGroups channel. If node **A** from the previous example was able to slow its send rate because **D** was not keeping up, **FC** would not be required. Applications that make synchronous group RPC calls are unlikely to require **FC**. In synchronous applications, the thread that makes the call blocks waiting for responses from all group members. This means that the threads on node **A** that make the calls would block waiting for responses from node **D**, naturally slowing the overall rate of calls.

A JBoss Cache cluster configured for **REPL_SYNC** is one example of an application that makes synchronous group RPC calls. If a channel is used only for a cache configured for **REPL_SYNC**, we recommend removing **FC** from its protocol stack.

If your cluster consists of two nodes, including **FC** in a TCP-based protocol stack is unnecessary, since TCP's internal flow control can handle one peer-to-peer relationship.

FC may also be omitted where a channel is used by a JBoss Cache configured for buddy replication with a single buddy. Such a channel acts much like a two-node cluster, where messages are only exchanged with one other node. Other messages related to data gravitation will be sent to all members, but these should be infrequent.



If you remove FC

Be sure to load test your application if you remove the **FC** element.

23.2. Fragmentation (FRAG2)

This protocol fragments messages larger than certain size. Messages are rejoined at the receiving end. This works for both unicast and multicast messages. It is configured in the **FRAG2** sub-element under the JGroups **Config** element, like so:

```
<FRAG2 frag_size="60000"/>
```

The configurable attributes in the **FRAG2** element are as follows.

frag_size

Specifies the maximum size of a fragment, in bytes. Messages larger than this value are fragmented. For stacks that use the UDP transport, this value must be lower than 64 kilobytes (the maximum UDP datagram size). For TCP-based stacks, it must be lower than the value of **max_credits** in the **FC** protocol.



Important

The TCP protocol provides fragmentation, but a JGroups fragmentation protocol is still required if **FC** is used, because if you send a message larger than **FC.max_credits**, the **FC** protocol blocks. The **frag_size** within **FRAG2** must always be less than **FC.max_credits**.

23.3. State Transfer

The state transfer service transfers the state from an existing node (that is, the cluster coordinator) to a newly joining node. It is configured with the **pbcast.STATE_TRANSFER** sub-element under the

JGroups **Config** element, as seen in the following code example. It has no configurable attribute.

```
<pbcast.STATE_TRANSFER/>
```

23.4. Distributed Garbage Collection (STABLE)

In a JGroups cluster, all nodes must 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. 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, like so:

```
<pbcast.STABLE stability_delay="1000"
  desired_avg_gossip="5000"
  max_bytes="400000"/>
```

The configurable attributes in the **pbcast.STABLE** element are as follows.

desired_avg_gossip

Specifies the interval (in milliseconds) between garbage collection runs. Setting this parameter to **0** disables this service.

max_bytes

Specifies the maximum number of bytes to receive before triggering a garbage collection run. Setting this parameter to **0** disables this service.



Note

Set **max_bytes** when you have a high-traffic cluster.

stability_delay

Specifies the delay period before a **STABILITY** message is sent. 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.

23.5. Merging (MERGE2)

When a network error occurs, a cluster may be divided into several partitions. The JGroups **MERGE** service lets partitions communicate with each other and reform into a single cluster. Merging is configured in the **MERGE2** sub-element in the JGroups **Config** element, like so:

```
<MERGE2 max_interval="10000"
  min_interval="2000"/>
```

The configurable attributes in the **MERGE2** element are as follows.

max_interval

Specifies the maximum number of milliseconds between **MERGE** messages.

min_interval

Specifies the minimum number of milliseconds between **MERGE** messages.

JGroups selects a random value between **min_interval** and **max_interval** to send the **MERGE** message.

**MERGE does not merge cluster states**

Merging the application state maintained by the application using a channel must be done by the application. If **MERGE2** is used in conjunction with **TCPPING**, the **initial_hosts** attribute must list all nodes to be merged. Only the listed nodes will be included in the merge process. Alternatively, **MPING** can be used with TCP to provide multicast member discovery capabilities without needing to specify all nodes.

23.6. Other Configuration Issues

23.6.1. Binding JGroups Channels to a Particular Interface

[Section 23.1.2, “Transport Protocols”](#) briefly described the interface to which JGroups binds sockets. Read this section to understand how to configure this interface.

The value set in any **bind_addr** element in an XML configuration file is ignored by JGroups if the **jgroups.bind_addr** (or deprecated **bind.address** system property) is already set. The system property will always override the XML configuration. The **-b** (or **--host**) switch is used to set the **jgroups.bind_addr** system property at server startup.

By default, JBoss Enterprise Web Platform binds most services to the local host if the **-b** switch is not set. Therefore, most users need to set **-b**, and any XML configuration will be ignored.

So, what are *best practices* for managing how JGroups binds to interfaces?

Bind JGroups to the same interface as other services

Use the **-b** switch, like so:

```
./run.sh -b 192.168.1.100 -c production
```

Bind services to one interface and JGroups to another

Specifically setting the system property with **-D** overrides the value specified by **-b**:

```
./run.sh -b 10.0.0.100 -Djgroups.bind_addr=192.168.1.100 -c production
```

The code here is a common usage pattern. It places client traffic on one network and intra-cluster traffic on another.

Bind services to all interfaces

Bind services to all interfaces with the following command on startup:

```
./run.sh -b 0.0.0.0 -c production
```



This will not bind JGroups to all interfaces

JGroups will bind to the machine's default interface. See [Section 23.1.2, “Transport Protocols”](#) to learn how to tell JGroups to send and receive on all interfaces.

Bind services to all interfaces and specify a JGroups interface

Specifically setting the system property with **-D** overrides the value specified by **-b**:

```
./run.sh -b 0.0.0.0 -Djgroups.bind_addr=192.168.1.100 -c production
```

Use different interfaces for different channels

Set the **jgroups.ignore.bind_addr** property to **true** on server startup, like so:

```
./run.sh -b 10.0.0.100 -Djgroups.ignore.bind_addr=true -c production
```

This setting tells JGroups to ignore the **jgroups.bind_addr** system property and use the value specified in the XML. You would then edit the XML configuration files to specify the **bind_addr** to the desired interface.

23.6.2. Isolating JGroups Channels

A number of services independently create JGroups channels: three JBoss Cache services (used for HTTP session replication, EJB3 SFSB replication, and EJB3 entity replication), and **HAPartition**, a clustering service that underlies most JBoss high availability services.



These channels must only communicate with their intended peers

They must not communicate with channels used by other services, or channels for the same service opened on machines outside the group. Nodes communicating improperly is one of the most common issues for users attempting to cluster JBoss Enterprise Web Platform.

JGroups channels communicate based on group name, multicast address and multicast port. Isolating a JGroups channel means ensuring that different channels use different values for the group name, multicast address, and multicast port.

23.6.2.1. Isolating sets of Application Server instances from each other

This section addresses the issue of having multiple independent clusters running within the same environment. For example, you might have a production cluster, a staging cluster, and a QA cluster, or multiple clusters in a QA test lab or development team environment.

To isolate JGroups clusters from other clusters on the network, you must:

- Make sure the channels in the various clusters use different group names. This can be controlled with the command line arguments used to start the server; see [Section 23.6.2.2.1, “Changing the Group Name”](#) for more information.
- Make sure the channels in the various clusters use different multicast addresses. This is also easy to control with the command line arguments used to start the server.
- If you are not running on Linux, Windows, Solaris or HP-UX, you may also need to ensure that the

channels in each cluster use different multicast ports. This is more difficult than using different group names, although it can still be controlled from the command line. See [Section 23.6.2.2.3, “Changing the Multicast Port”](#). Note that using different ports should not be necessary if your servers are running on Linux, Windows, Solaris or HP-UX.

23.6.2.2. Isolating Channels for Different Services on the Same Set of AS Instances

This section addresses the usual case: a cluster of three machines, each of which has, for example, an HAPartition deployed alongside JBoss Cache for web session clustering. The HAPartition channels should not communicate with the JBoss Cache channels. Ensuring proper isolation of these channels is straightforward, and is usually handled by the application server without any alterations on the part of the user.

To isolate channels for different services from each other on the same set of application server instances, each channel must have its own group name. The configurations that are shipped with JBoss Enterprise Web Platform ensure that this is the case. However, if you create a custom service that uses JGroups directly, you must use a unique group name. If you create a custom JBoss Cache configuration, ensure that you provide a unique value in the **clusterName** configuration property.

JGroups uses shared transport by default, so it is common for multiple channels to use the same transport protocol and its sockets. This makes configuration easier, which is one of the main benefits of the shared transport. However, if you decide to create your own custom JGroups protocol stack configuration, be sure to configure its transport protocols with a multicast port that is different from the ports used in other protocol stacks.

23.6.2.2.1. Changing the Group Name

The group name for a JGroups channel is configured via the service that starts the channel. For all the standard clustered services, we make it easy for you to create unique groups names by simply using the **-g** (or **--partition**) switch when starting the server:

```
./run.sh -g QAPartition -b 192.168.1.100 -c production
```

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,

```
<property name="clusterName">${jboss.partition.name:DefaultPartition}-SFSBCache</property>
```

23.6.2.2.2. Changing the multicast address and port

The **-u** (or **--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 production
```

This switch sets the **jboss.partition.udpGroup** system property, which is referenced in all of the standard protocol stack configurations in JBoss Enterprise Web Platform:

```
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}" ....
```



Why is changing the group name insufficient?

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.

23.6.2.2.3. Changing the Multicast Port

On some operating systems (Mac OS X for example), using different **-g** and **-u** values is not sufficient to isolate clusters; the channels running in the different clusters must also use different multicast ports. Unfortunately, setting the multicast ports is not as simple as **-g** and **-u**. By default, a JBoss Enterprise Web Platform instance running the **production** configuration will use up to two different instances of the JGroups UDP transport protocol, and will therefore open two multicast sockets. You can control the ports those sockets use by using system properties on the command line. For example,

```
./run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c production \\  
-Djboss.jgroups.udp.mcast_port=12345
```

The **jboss.jgroups.udp.mcast_port** property controls the multicast port used by the UDP transport protocol shared by all other clustered services.

The set of JGroups protocol stack configurations included in the **\$JBoss_HOME/server/production/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** file includes a number of other example protocol stack configurations that the standard JBoss Enterprise Web Platform distribution doesn't actually use. Those configurations also use system properties to set any multicast ports. So, if you reconfigure some AS service to use one of those protocol stack configurations, use the appropriate system property to control the port from the command line.



Why do I need to change the multicast port if I change the address?

It should be sufficient to just change the address, but unfortunately the handling of multicast sockets is one area where the JVM fails to hide operating system behavior differences from the application. The **java.net.MulticastSocket** class provides different overloaded constructors. On some operating systems, if you use one constructor variant, packets addressed to a particular multicast port are delivered to all listeners on that port, regardless of the multicast address on which they are listening. We refer to this as the *promiscuous traffic* problem. On most operating systems that exhibit the promiscuous traffic problem (Linux, Solaris and HP-UX) JGroups can use a different constructor variant that avoids the problem. However, on some operating systems with the promiscuous traffic problem (Mac OS X), multicast does not work properly if the other constructor variant is used. So, on these operating systems the recommendation is to configure different multicast ports for different clusters.

23.6.2.3. Improving UDP Performance by Configuring OS UDP Buffer Limits

By default, the JGroups channels in JBoss Enterprise Web Platform use the UDP transport protocol to take advantage of IP multicast. However, one disadvantage of UDP is it does not come with the reliable delivery guarantees provided by TCP. The protocols discussed in [Section 23.1.5, “Reliable Delivery Protocols”](#) allow JGroups to guarantee delivery of UDP messages, but those protocols are implemented in Java, not at the operating system network layer. For peak performance from a UDP-based JGroups channel it is important to limit the need for JGroups to retransmit messages by limiting UDP datagram loss.

One of the most common causes of lost UDP datagrams is an undersized receive buffer on the socket. The UDP protocol's **mcast_recv_buf_size** and **ucast_recv_buf_size** configuration attributes are used to specify the amount of receive buffer JGroups *requests* from the operating system, but the actual size of the buffer the operating system provides is limited by operating system-level maximums. These maximums are often very low:

Table 23.1. Default Max UDP Buffer Sizes

Operating System	Default Max UDP Buffer (in bytes)
Linux	131071
Windows	No known limit
Solaris	262144
FreeBSD, Darwin	262144
AIX	1048576

The command used to increase the above limits is operating system-specific. The table below shows the command required to increase the maximum buffer to 25 megabytes. In all cases, root privileges are required:

Table 23.2. Commands to Change Max UDP Buffer Sizes

Operating System	Command
Linux	<code>sysctl -w net.core.rmem_max=26214400</code>
Solaris	<code>ndd -set /dev/udp udp_max_buf 26214400</code>
FreeBSD, Darwin	<code>sysctl -w kern.ipc.maxsockbuf=26214400</code>
AIX	<code>no -o sb_max=8388608</code> (AIX will only allow 1 megabyte, 4 megabytes or 8 megabytes).

23.6.3. JGroups Troubleshooting

23.6.3.1. Nodes do not form a cluster

Ensure that your machine is set up correctly for IP multicast. There are two test programs that can detect this: **McastReceiverTest** and **McastSenderTest**.

1. Start **McastReceiverTest** from the `$JBoss_HOME/server/production/lib` directory, like so:

```
java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -mcast_addr
224.10.10.10 -port 5555
```

2. In another window, start **McastSenderTest** from the same directory:

```
java -cp jgroups.jar org.jgroups.tests.McastSenderTest -mcast_addr 224.10.10.10
-port 5555
```



Note

Use the **-bind_addr** switch to bind to a specific network interface card (NIC). To bind to an NIC with an IP address of **192.168.0.2**, you would use **-bind_addr 192.168.0.2**. This parameter can be used in both senders and receivers.

3. Type in the **McastSenderTest** window. You should be able to see the output in the **McastReceiverTest** window.

If you cannot, try using **-ttl 32** in the sender. If this still fails, consult a system administrator to help you set up IP multicast correctly. Check that multicast will work on the interface you have chosen. If the

machines have multiple interfaces, ask which interface is correct for multicasting.

When multicast is working correctly on each machine in your cluster, verify that your network is working correctly by repeating this test with **McastReceiverTest** on one machine and **McastSenderTest** on another.

23.6.3.2. Causes of missing heartbeats in FD

Sometimes a member is suspected by FD because a heartbeat acknowledgement has not been received for some time (defined by **timeout** and **max_tries**). This may occur for several reasons. As an example, say you have a cluster consisting of nodes A, B, C and D. In this cluster, A pings B, B pings C, C pings D, and D pings A.

C may be suspected in any of the following situations.

- ▶ If B and C are running at 100% CPU for longer than the time defined by **timeout** and **max_tries**. Even if C sends a heartbeat acknowledgement to B, B may not be able to process the acknowledgement.
- ▶ If B or C are garbage collecting, they may not respond or process acknowledgement of a heartbeat message.
- ▶ If the network loses packets, heartbeat messages or acknowledgements may be lost. This can occur when a network has high traffic. Packets are usually dropped in the following order: broadcasts, IP multicasts, then TCP packets.
- ▶ If B or C are processing a callback. Say C receives a remote method call and takes longer than the **timeout** or **max_tries** period to process it. During this time, C does not process any other message, including heartbeats. Therefore B will not receive a heartbeat acknowledgement, and will suspect C.

Chapter 24. JBoss Cache Configuration and Deployment

JBoss Cache provides the underlying distributed caching support used by many of the standard clustered services in a JBoss Enterprise Web Platform cluster. You can also deploy JBoss Cache in your own application to handle custom caching requirements. In this chapter we provide some background on the main configuration options available with JBoss Cache, with an emphasis on how those options relate to the JBoss Cache usage by the standard clustered services the Enterprise Web Platform provides. We then discuss the different options available for deploying a custom cache in the Enterprise Web Platform.

Users considering deploying JBoss Cache for direct use by their own application are strongly encouraged to read the JBoss Cache documentation available from http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/.

See also [Section 17.1.3, “Distributed Caching with JBoss Cache”](#) for information on how the standard JBoss Enterprise Web Platform clustered services use JBoss Cache.

24.1. Key JBoss Cache Configuration Options

JBoss Enterprise Web Platform ships with a reasonable set of default JBoss Cache configurations that are suitable for the standard clustered service use cases (for example, web session replication or JPA/Hibernate caching). Most applications that involve the standard clustered services just work out of the box with the default configurations. You only need to tweak them when you are deploying an application that has special network or performance requirements. In this section we provide a brief overview of some of the key configuration choices. This is by no means a complete discussion; for full details users interested in moving beyond the default configurations are encouraged to read the JBoss Cache documentation available at http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/.

Most JBoss Cache configuration examples in this section use the JBoss Microcontainer schema for building up an **org.jboss.cache.config.Configuration** object graph from XML. JBoss Cache has its own custom XML schema, but the standard JBoss Enterprise Web Platform **CacheManager** service uses the JBoss Microcontainer schema to be consistent with most other internal Enterprise Web Platform services.

Before getting into the key configuration options, let's have a look at the most likely place that a user would encounter them.

24.1.1. Editing the CacheManager Configuration

As discussed in [Section 17.1.3.1, “The JBoss Enterprise Web Platform CacheManager Service”](#), the standard JBoss Enterprise Web Platform clustered services use the CacheManager service as a factory for JBoss Cache instances. So, cache configuration changes are likely to involve edits to the **CacheManager** service.



Note

Users can also use the **CacheManager** as a factory for custom caches used by directly by their own applications; see [Section 24.2.1, “Deployment Via the CacheManager Service”](#).

The **CacheManager** is configured via the **deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml** file. The element most likely to be edited is the **CacheConfigurationRegistry** bean, which maintains a registry of all the named JBC configurations the **CacheManager** knows about. Most edits to this file would involve adding a new JBoss Cache configuration or changing a property of an existing one.

The following is a redacted version of the **CacheConfigurationRegistry** bean configuration:


```

<bean name="CacheConfigurationRegistry"
  class="org.jboss.ha.cachemanager.DependencyInjectedConfigurationRegistry">

  <!-- If users wish to add configs using a more familiar JBC config format
  they can add them to a cache-configs.xml file specified by this
property.
  However, use of the microcontainer format used below is recommended.
  <property name="configResource">META-INF/jboss-cache-configs.xml</property>
  -->

  <!-- The configurations. A Map<String name, Configuration config> -->
  <property name="newConfigurations">
    <map keyClass="java.lang.String"
valueClass="org.jboss.cache.config.Configuration">

      <!-- The standard configurations follow. You can add your own and/or edit
these. -->

      <!-- Standard cache used for web sessions -->
      <entry><key>standard-session-cache</key>
      <value>
        <bean name="StandardSessionCacheConfig"
class="org.jboss.cache.config.Configuration">

          <!-- Provides batching functionality for caches that don't want to
          interact with regular JTA Transactions -->
          <property name="transactionManagerLookupClass">
            org.jboss.cache.transaction.BatchModeTransactionManagerLookup
          </property>

          <!-- Name of cluster. Needs to be the same for all members -->
          <property name="clusterName">${jboss.partition.name:DefaultPartition}-
SessionCache</property>
          <!-- Use a UDP (multicast) based stack. Need JGroups flow control (FC)
          because we are using asynchronous replication. -->
          <property
name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
          <property name="fetchInMemoryState">true</property>

          <property name="nodeLockingScheme">PESSIMISTIC</property>
          <property name="isolationLevel">REPEATABLE_READ</property>
          <property name="cacheMode">REPL_ASYNC</property>

          .... more details of the standard-session-cache configuration
        </bean>
      </value>
    </entry>

    <!-- Appropriate for web sessions with FIELD granularity -->
    <entry><key>field-granularity-session-cache</key>
    <value>

      <bean name="FieldSessionCacheConfig"
class="org.jboss.cache.config.Configuration">
        .... details of the field-granularity-standard-session-cache
configuration
      </bean>

    </value>

    </entry>

    ... entry elements for the other configurations

```

```
</map>  
</property>  
</bean>
```

The actual JBoss Cache configurations are specified using the JBoss Microcontainer's schema rather than one of the standard JBoss Cache configuration formats. When JBoss Cache parses one of its standard configuration formats, it creates a Java Bean of type

org.jboss.cache.config.Configuration with a tree of child Java Beans for some of the more complex sub-configurations (for example, cache loading, eviction, buddy replication). Rather than delegating this task of XML parsing and Java Bean creation to JBC, we let the Enterprise Web Platform's microcontainer do it directly. This has the advantage of making the microcontainer aware of the configuration beans, which in later Enterprise Web Platform releases will be helpful in allowing external management tools to manage the JBC configurations.

The configuration format should be fairly self-explanatory if you look at the standard configurations the Enterprise Web Platform ships; they include all the major elements. The types and properties of the various Java Beans that make up a JBoss Cache configuration can be seen in the JBoss Cache Javadocs. Here is a fairly complete example:

```

<bean name="StandardSFSBCacheConfig" class="org.jboss.cache.config.Configuration">

    <!-- No transaction manager lookup -->

    <!-- Name of cluster. Needs to be the same for all members -->
    <property name="clusterName">${jboss.partition.name:DefaultPartition}-
SFSBCache</property>
    <!-- Use a UDP (multicast) based stack. Need JGroups flow control (FC)
        because we are using asynchronous replication. -->
    <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
    <property name="fetchInMemoryState">true</property>

    <property name="nodeLockingScheme">PESSIMISTIC</property>
    <property name="isolationLevel">REPEATABLE_READ</property>
    <property name="cacheMode">REPL_ASYNC</property>

    <property name="useLockStriping">false</property>

    <!-- Number of milliseconds to wait until all responses for a
        synchronous call have been received. Make this longer
        than lockAcquisitionTimeout.-->
    <property name="syncReplTimeout">17500</property>
    <!-- Max number of milliseconds to wait for a lock acquisition -->
    <property name="lockAcquisitionTimeout">15000</property>
    <!-- The max amount of time (in milliseconds) we wait until the
        state (ie. the contents of the cache) are retrieved from
        existing members at startup. -->
    <property name="stateRetrievalTimeout">60000</property>

    <!--
        SFSBs use region-based marshalling to provide for partial state
        transfer during deployment/undeployment.
    -->
    <property name="useRegionBasedMarshalling">false</property>
    <!-- Must match the value of "useRegionBasedMarshalling" -->
    <property name="inactiveOnStartup">false</property>

    <!-- Disable asynchronous RPC marshalling/sending -->
    <property name="serializationExecutorPoolSize">0</property>
    <!-- We have no asynchronous notification listeners -->
    <property name="listenerAsyncPoolSize">0</property>

    <property name="exposeManagementStatistics">true</property>

    <property name="buddyReplicationConfig">
        <bean class="org.jboss.cache.config.BuddyReplicationConfig">

            <!-- Just set to true to turn on buddy replication -->
            <property name="enabled">false</property>

            <!-- A way to specify a preferred replication group. We try
                and pick a buddy who shares the same pool name (falling
                back to other buddies if not available). -->
            <property name="buddyPoolName">default</property>

            <property name="buddyCommunicationTimeout">17500</property>

            <!-- Do not change these -->
            <property name="autoDataGravitation">false</property>
            <property name="dataGravitationRemoveOnFind">true</property>
            <property name="dataGravitationSearchBackupTrees">true</property>

            <property name="buddyLocatorConfig">
                <bean

```

```

class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
    <!-- The number of backup nodes we maintain -->
    <property name="numBuddies">1</property>
    <!-- Means that each node will *try* to select a buddy on
         a different physical host. If not able to do so
         though, it will fall back to colocated nodes. -->
    <property name="ignoreColocatedBuddies">true</property>
</bean>
</property>
</bean>
</property>
<property name="cacheLoaderConfig">
    <bean class="org.jboss.cache.config.CacheLoaderConfig">
        <!-- Do not change these -->
        <property name="passivation">true</property>
        <property name="shared">false</property>

        <property name="individualCacheLoaderConfigs">
            <list>
                <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
                    <!-- Where passivated sessions are stored -->
                    <property
name="location">${jboss.server.data.dir}/${/}sfsb</property>
                    <!-- Do not change these -->
                    <property name="async">false</property>
                    <property name="fetchPersistentState">true</property>
                    <property name="purgeOnStartup">true</property>
                    <property name="ignoreModifications">false</property>
                    <property name="checkCharacterPortability">false</property>
                </bean>
            </list>
        </property>
    </bean>
</property>

<!-- EJBs use JBoss Cache eviction -->
<property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
        <property name="wakeupInterval">5000</property>
        <!-- Overall default -->
        <property name="defaultEvictionRegionConfig">
            <bean class="org.jboss.cache.config.EvictionRegionConfig">
                <property name="regionName">/</property>
                <property name="evictionAlgorithmConfig">
                    <bean
class="org.jboss.cache.eviction.NullEvictionAlgorithmConfig"/>
                </property>
            </bean>
        </property>
        <!-- EJB3 integration code will programatically create
             other regions as beans are deployed -->
    </bean>
</property>
</bean>

```

Basically, the XML specifies the creation of an **org.jboss.cache.config.Configuration** Java Bean and the setting of a number of properties on that bean. Most of the properties are of simple types, but some, such as **buddyReplicationConfig** and **cacheLoaderConfig** take various types of Java Bean as their values.

Next we'll look at some of the key configuration options.

24.1.2. Cache Mode

JBoss Cache's **cacheMode** configuration attribute combines into a single property two related aspects:

Handling of Cluster Updates

This controls how a cache instance on one node should notify the rest of the cluster when it makes changes in its local state. There are three options:

Synchronous

When a change is made, the cache sends a message to notify its peers of changes and waits for acknowledgement that its peers have applied those changes before returning. If the changes are made as part of a JTA transaction, this is done as part of a two-phase commit process. Any locks are held until acknowledgement is received. Waiting for this acknowledgement to be received from all nodes adds delays, but ensures consistency around the cluster. Synchronous mode is required when there is a high need for consistency, such as when all nodes can access cached data.

Asynchronous

When a change is made, the cache sends a message to notify its peers of changes and returns immediately *without* any acknowledgement that the changes have been applied. Asynchronous mode is most useful in session replication, where only the cache that sends the messages accesses data, and cluster messages are used to provide backup copies in the case of a failure on the sending node. Asynchronous messaging has minor consistency risks, in that a later user request that fails over to another node may see out-of-date state. However, this is usually considered an acceptable risk for session-type applications because of the major performance benefits associated with asynchronous messaging.

Local

No message is sent, and no JGroups channel is used by the cache. Local messaging will provide improved performance when cached data need not be kept consistent around the cluster. One example of this is caching JPA/Hibernate query result sets: Hibernate's second level cache logic uses a separate mechanism to invalidate stale query result sets from the second level cache, so JBoss Cache does not need to send messages around the cluster for a query result set cache.

Replication vs. Invalidation

Nodes in a cluster can reflect changes from the sending node in two ways:

Replication

Other nodes update their state to reflect the new state on the sending node. This means that the sending node must include the changed state, increasing the resource cost of the message. Replication is required if there is no other way for other nodes to obtain the changed state.

Invalidation

Other nodes remove the changed state from their local state. Invalidation mode reduces the resource cost of cluster update messages, since only the cache key of the state changed needs to be transmitted. However, this is only an option if the removed state can be retrieved from another source. It is an excellent option for a clustered JPA/Hibernate entity cache, since in this case cached state can be reread from the database.

The **cacheMode** configuration attribute has five possible values, which form various combinations of the aforementioned aspects:

LOCAL

No cluster messages are required.

REPL_SYNC

Synchronous replication messages are sent.

REPL_ASYNC

Asynchronous replication messages are sent.

INVALIDATION_SYNC

Synchronous invalidation messages are sent.

INVALIDATION_ASYNC

Asynchronous invalidation messages are sent.

24.1.3. Transaction Handling

JBoss Cache integrates with JTA transaction managers to allow transactional access to the cache. When JBoss Cache detects the presence of a transaction, any locks are held for the life of the transaction, changes made to the cache will be reverted if the transaction rolls back, and any cluster-wide messages sent to inform other nodes of changes are deferred and sent in a batch as part of transaction commit (reducing chattiness).

Integration with a transaction manager is accomplished by setting the **transactionManagerLookupClass** configuration attribute; this specifies the fully qualified class name of a class JBoss Cache can use to find the local transaction manager. Inside JBoss Enterprise Web Platform, this attribute would have one of two values:

org.jboss.cache.transaction.JBossTransactionManagerLookup

The standard transaction manager running in the application server. Use this for any custom caches you deploy where you want caching to participate in any JTA transactions.

org.jboss.cache.transaction.BatchModeTransactionManagerLookup

Used in the cache configurations for web session caching. Specifies the **BatchModeTransactionManager**, a mock **TransactionManager** shipped with JBoss Cache. Since this is not a true JTA transaction manager, it should not be used with anything other than JBoss Cache. In JBoss Enterprise Web Platform, it benefits from JBoss Cache's transactional behavior in session replication, without participating in end-user transactions that may run during a request.



Note

The **transactionManagerLookupClass** should not be configured for caches that are used for JPA/Hibernate caching. Hibernate internally configures the cache to use the same transaction manager that Hibernate uses for database access.

24.1.4. Concurrent Access

JBoss Cache is a thread safe caching API, and uses its own efficient mechanisms of controlling

concurrent access. Concurrency is configured via the **nodeLockingScheme** and **isolationLevel** configuration attributes.

There are three possible values for **nodeLockingScheme**:

MVCC (Multi-Versioned Concurrency Control)

A locking scheme commonly used by modern database implementations to control fast, safe concurrent access to shared data. JBoss Cache 3.x uses an innovative implementation of MVCC as the default locking scheme. MVCC is designed to provide the following features for concurrent access:

- readers that do not block writers
- writers that fail fast

It achieves this by using data versioning and copying for concurrent writers. In theory, readers continue reading shared state, while writers copy the shared state, increment the version ID, and write that shared state back after verifying that the version is still valid (that is, that another concurrent writer has not altered the state first).

MVCC is recommended for JPA/Hibernate entity caching.

PESSIMISTIC

Threads or transactions acquire either exclusive or non-exclusive locks on nodes before reading or writing. The lock type acquired depends on the **isolationLevel**, but in most cases a non-exclusive lock is acquired for a read, and an exclusive lock is acquired for a write. Pessimistic locking requires more resources than MVCC, and allows less concurrency, since reader threads must block until a write has completed and released its exclusive lock. This can be a long time if the write is part of a transaction. A write will also be delayed due to ongoing reads.

Pessimistic locking is deprecated as of JBoss Cache 3.0. For session caching use in JBoss Enterprise Web Platform, **PESSIMISTIC** is the default value, since concurrent threads usually do not access the same cache location in session caching and the benefits of MVCC are not as great.

OPTIMISTIC

Improves the concurrency available with pessimistic locking by creating a "workspace" for each request or transaction that accesses the cache. Data accessed by requests (including reads) is *copied* into the workspace, which increases resource usage. All data is versioned, so when non-transactional requests or transaction commits are completed, the version of data in the workspace is compared to the primary cache. If there are inconsistencies, an exception is raised. Otherwise, changes to the workspace are applied to the main cache.

Optimistic locking is deprecated, but is provided to support backwards compatibility. Users are encouraged to use **MVCC** instead, which provides the same benefits at a lower cost to resources.

The **isolationLevel** attribute has two possible values, **READ_COMMITTED** and **REPEATABLE_READ**, which correspond to database style isolation levels. The default isolation level is **REPEATABLE_READ**, which maintains backwards compatibility with previous versions of JBoss Cache. **READ_COMMITTED** provides slightly weaker isolation, but has significant performance benefits.

24.1.5. JGroups Integration

Each JBoss Cache instance internally uses a JGroups **Channel** to handle group communications. Inside JBoss Enterprise Web Platform, we strongly recommend that you use the Enterprise Web

Platform's JGroups Channel Factory service as the source for your cache's **Channel1**. In this section we discuss how to configure your cache to get its channel from the Channel Factory; if you wish to configure the channel in some other way, see the JBoss Cache documentation.

Caches obtained from the CacheManager Service

This is the simplest approach. The **CacheManager** service already contains a reference to the Channel Factory service, so the only configuration task is to configure the name of the JGroups protocol stack configuration to use.

If you are configuring your cache via the CacheManager service's **jboss-cache-manager-jboss-beans.xml** file (see [Section 24.2.1, "Deployment Via the CacheManager Service"](#)), add the following to your cache configuration, where the value is the name of the protocol stack configuration (here **udp**):

```
<property name="multiplexerStack">udp</property>
```

Caches Deployed via a -jboss-beans.xml File

If you are deploying a cache via a JBoss Microcontainer **-jboss-beans.xml** file (see [Section 24.2.3, "Deployment Via a -jboss-beans.xml File"](#)), you need to inject a reference to the Channel Factory service as well as specifying the protocol stack configuration:

```
<property name="runtimeConfig">
  <bean class="org.jboss.cache.config.RuntimeConfig">
    <property name="muxChannelFactory"><inject
      bean="JChannelFactory"/></property>
  </bean>
</property>
<property name="multiplexerStack">udp</property>
```

Caches Deployed via a -service.xml File

If you are deploying a cache MBean via **-service.xml** file (see [Section 24.2.2, "Deployment Via a -service.xml File"](#)), **CacheJmxWrapper** is the class of your MBean; that class exposes a **MuxChannelFactory** MBean attribute. You dependency inject the Channel Factory service into this attribute, and set the protocol stack name via the **MultiplexerStack** attribute:

```
<attribute name="MuxChannelFactory"><inject bean="JChannelFactory"/></attribute>
<attribute name="MultiplexerStack">udp</attribute>
```

24.1.6. Eviction

Eviction allows the cache to control memory by removing data (typically the least frequently used data). If you wish to configure eviction for a custom cache, see the JBoss Cache documentation for all of the available options. For details on configuring it for JPA/Hibernate caching, see the Eviction chapter in the "Using JBoss Cache as a Hibernate Second Level Cache" guide at <http://www.jboss.org/jbossclustering/docs/hibernate-jboss-cache-guide-3.pdf>. For web session caches, eviction should not be configured; the distributable session manager handles eviction itself. For EJB3 SFSB caches, stick with the eviction configuration in the Enterprise Web Platform's standard **sf-sb-cache** configuration (see [Section 17.1.3.1, "The JBoss Enterprise Web Platform CacheManager Service"](#)). The EJB container will configure eviction itself using the values included in each bean's configuration.

24.1.7. Cache Loaders

Cache loading allows JBoss Cache to store data in a persistent store in addition to what it keeps in memory. This data can either be an overflow, where the data in the persistent store is not reflected in memory, or it can be a superset of what is in memory, where everything in memory is also reflected in the persistent store, along with items that have been evicted from memory. Which of these two modes is

used depends on the setting of the **passivation** flag in the JBoss Cache cache loader configuration section. A **true** value means the persistent store acts as an overflow area written to when data is evicted from the in-memory cache.

If you wish to configure cache loading for a custom cache, see the JBoss Cache documentation for all of the available options. Do not configure cache loading for a JPA/Hibernate cache, as the database itself serves as a persistent store; adding a cache loader is redundant.

The caches used for web session and EJB3 SFSB caching use passivation. Next we'll discuss the cache loader configuration for those caches in some detail.

24.1.7.1. CacheLoader Configuration for Web Session and SFSB Caches

HttpSession and SFSB passivation rely on JBoss Cache's Cache Loader passivation for storing and retrieving the passivated sessions. Therefore the cache instance used by your webapp's clustered session manager or your bean's EJB container must be configured to enable Cache Loader passivation.

In most cases you don't need to do anything to alter the cache loader configurations for the standard web session and SFSB caches; the standard JBoss Enterprise Web Platform configurations should suit your needs. The following is a bit more detail in case you're interested or want to change from the defaults.

The Cache Loader configuration for the **standard-session-cache** configuration serves as a good example:

```
<property name="cacheLoaderConfig">
  <bean class="org.jboss.cache.config.CacheLoaderConfig">
    <!-- Do not change these -->
    <property name="passivation">true</property>
    <property name="shared">false</property>

    <property name="individualCacheLoaderConfigs">
      <list>
        <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
          <!-- Where passivated sessions are stored -->
          <property
name="location">${jboss.server.data.dir}${/}session</property>
          <!-- Do not change these -->
          <property name="async">false</property>
          <property name="fetchPersistentState">true</property>
          <property name="purgeOnStartup">true</property>
          <property name="ignoreModifications">false</property>
          <property name="checkCharacterPortability">false</property>
        </bean>
      </list>
    </property>
  </bean>
</property>
```

- **passivation** must be set to **true**.
- **shared** must be set to **false**. Do not passivate sessions to a shared persistent store, or if another node activates the session, the session will be removed from the persistent store and from the memory on other nodes that have passivated it. Backup copies will be lost.
- **individualCacheLoaderConfigs** accepts a list of cache loader configurations. You can also chain cache loaders (see the JBoss Cache documentation for details). For session passivation, a single cache loader is sufficient.
- **class** must refer to the configuration class for a cache loader implementation, for example, **org.jboss.cache.loader.FileCacheLoaderConfig** or **org.jboss.cache.loader.JDBCCacheLoaderConfig**. See the JBoss Cache documentation for more information about the available **CacheLoader** implementations. If you want to use

JDBCCLoader (to persist to a database rather than the filesystem used by **FileCacheLoader**), take note of the comments about the **shared** property. Do not use a shared database. Each node in the cluster must have its own storage location.

- **FileCacheLoaderConfig**'s **location** property defines the root node of the filesystem tree where passivated sessions should be stored. The default is to store them in your JBoss Enterprise Web Platform configuration's **data** directory.
- **async** must be **false** to ensure passivated sessions are promptly written to the persistent store.
- **fetchPersistentState** property must be **true** to ensure passivated sessions are included in the set of session backup copies transferred over from other nodes when the cache starts.
- **purgeOnStartup** should be **true** to ensure out-of-date session data left over from a previous shutdown of a server doesn't pollute the current data set.
- **ignoreModifications** should be **false**
- **checkCharacterPortability** should be **false** for a minor performance improvement.

24.1.8. Buddy Replication

Buddy Replication is a JBoss Cache feature that allows you to suppress replicating your data to all instances in a cluster. Instead, each instance picks one or more *buddies* in the cluster, and only replicates to those specific buddies. This greatly helps scalability, as there is no longer a memory and network traffic impact every time another instance is added to a cluster.

If the cache on another node needs data that it doesn't have locally, it can ask the other nodes in the cluster to provide it; nodes that have a copy will provide it as part of a process called *data gravitation*. The new node will become the owner of the data, placing a backup copy of the data on its buddies. The ability to gravitate data means there is no need for all requests for data to occur on a node that has a copy of it; any node can handle a request for any data. However, data gravitation is expensive and should not be a frequent occurrence; ideally it should only occur if the node that is using some data fails or is shut down, forcing interested clients to fail over to a different node. This makes buddy replication primarily useful for session-type applications with session affinity (also known as *sticky sessions*) where all requests for a particular session are normally handled by a single server.

Buddy replication can be enabled for the web session and EJB3 SFSB caches. Do not add buddy replication to the cache configurations used for other standard clustering services (for example, JPA/Hibernate caching). Services not specifically engineered for buddy replication are highly unlikely to work correctly if it is introduced.

Configuring buddy replication is fairly straightforward. As an example we'll look at the buddy replication configuration section from the **CacheManager** service's **standard-session-cache** config:

```

<property name="buddyReplicationConfig">
  <bean class="org.jboss.cache.config.BuddyReplicationConfig">

    <!-- Just set to true to turn on buddy replication -->
    <property name="enabled">true</property>

    <!-- A way to specify a preferred replication group. We try
         and pick a buddy who shares the same pool name (falling
         back to other buddies if not available). -->
    <property name="buddyPoolName">default</property>

    <property name="buddyCommunicationTimeout">17500</property>

    <!-- Do not change these -->
    <property name="autoDataGravitation">false</property>
    <property name="dataGravitationRemoveOnFind">true</property>
    <property name="dataGravitationSearchBackupTrees">true</property>

    <property name="buddyLocatorConfig">
      <bean
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
        <!-- The number of backup copies we maintain -->
        <property name="numBuddies">1</property>
        <!-- Means that each node will *try* to select a buddy on
             a different physical host. If not able to do so
             though, it will fall back to colocated nodes. -->
        <property name="ignoreColocatedBuddies">true</property>
      </bean>
    </property>
  </bean>
</property>

```

The main things you would be likely to configure are:

- ▶ **buddyReplicationEnabled** — **true** if you want buddy replication; **false** if data should be replicated to all nodes in the cluster, in which case none of the other buddy replication configurations matter.
- ▶ **numBuddies** — to how many backup nodes should each node replicate its state.
- ▶ **buddyPoolName** — allows logical subgrouping of nodes within the cluster; if possible, buddies will be chosen from nodes in the same buddy pool.

The **ignoreColocatedBuddies** switch means that when the cache is trying to find a buddy, where possible it will not choose a buddy on the same physical host as itself. If the only server it can find is running on its own machine, it will use that server as a buddy.

Do not change the settings for **autoDataGravitation**, **dataGravitationRemoveOnFind** or **dataGravitationSearchBackupTrees**. Session replication will not work properly if these are changed.

24.2. Deploying Your Own JBoss Cache Instance

It's quite common for users to deploy their own instances of JBoss Cache inside JBoss Enterprise Web Platform for custom use by their applications. In this section we describe the various ways caches can be deployed.

24.2.1. Deployment Via the CacheManager Service

The standard JBoss clustered services that use JBoss Cache obtain a reference to their cache from the Enterprise Web Platform's **CacheManager** service (see [Section 17.1.3.1, "The JBoss Enterprise Web Platform CacheManager Service"](#)). End user applications can do the same thing.

[Section 24.1.1, “Editing the CacheManager Configuration”](#) shows the configuration of the **CacheManager**'s **CacheConfigurationRegistry** bean. To add a new configuration, you would add an additional element inside that bean's **newConfigurations** **<map>**:

```
<bean name="CacheConfigurationRegistry"
      class="org.jboss.ha.cachemanager.DependencyInjectedConfigurationRegistry">
    ....
    <property name="newConfigurations">
      <map keyClass="java.lang.String"
valueClass="org.jboss.cache.config.Configuration">

        <entry><key>my-custom-cache</key>
          <value>
            <bean name="MyCustomCacheConfig"
class="org.jboss.cache.config.Configuration">
              .... details of the my-custom-cache configuration
            </bean>
          </value>
        </entry>
      </map>
    </property>
  </bean>
```

See [Section 24.1.1, “Editing the CacheManager Configuration”](#) for an example configuration.

24.2.1.1. Accessing the CacheManager

Once you've added your cache configuration to the **CacheManager**, the next step is to provide a reference to your application to the **CacheManager**. There are three ways to do this:

► Dependency Injection

If your application uses the JBoss Microcontainer for configuration, the simplest mechanism is to have it inject the **CacheManager** into your service.

```
<bean name="MyService" class="com.example.MyService">
  <property name="cacheManager"><inject bean="CacheManager"/></property>
</bean>
```

► JNDI Lookup

Alternatively, you can look up the **CacheManager** in JNDI. It is bound under **java:CacheManager**.

```
import org.jboss.ha.cachemanager.CacheManager;

public class MyService {
  private CacheManager cacheManager;

  public void start() throws Exception {
    Context ctx = new InitialContext();
    cacheManager = (CacheManager) ctx.lookup("java:CacheManager");
  }
}
```

► CacheManagerLocator

JBoss Enterprise Web Platform also provides a service locator object that can be used to access the **CacheManager**.

```
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
    private CacheManager cacheManager;

    public void start() throws Exception {
        CacheManagerLocator locator =
CacheManagerLocator.getCacheManagerLocator();
        // Locator accepts as param a set of JNDI properties to help in lookup;
        // this isn't necessary inside the Enterprise Web Platform
        cacheManager = locator.getCacheManager(null);
    }
}
```

Once a reference to the **CacheManager** is obtained; usage is simple. Access a cache by passing in the name of the desired configuration. The **CacheManager** will not start the cache; this is the responsibility of the application. The cache may, however, have been started by another application running in the cache server; the cache may be shared. When the application is done using the cache, it should not stop. Just inform the **CacheManager** that the cache is no longer being used; the manager will stop the cache when all callers that have asked for the cache have released it.

```
import org.jboss.cache.Cache;
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
    private CacheManager cacheManager;
    private Cache cache;

    public void start() throws Exception {
        Context ctx = new InitialContext();
        cacheManager = (CacheManager) ctx.lookup("java:CacheManager");

        // "true" param tells the manager to instantiate the cache if
        // it doesn't exist yet
        cache = cacheManager.getCache("my-cache-config", true);

        cache.start();
    }

    public void stop() throws Exception {
        cacheManager.releaseCache("my-cache-config");
    }
}
```

The **CacheManager** can also be used to access instances of POJO Cache.

```

import org.jboss.cache.pojo.PojoCache;
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
    private CacheManager cacheManager;
    private PojoCache pojoCache;

    public void start() throws Exception {
        Context ctx = new InitialContext();
        cacheManager = (CacheManager) ctx.lookup("java:CacheManager");

        // "true" param tells the manager to instantiate the cache if
        // it doesn't exist yet
        pojoCache = cacheManager.getPojoCache("my-cache-config", true);

        pojoCache.start();
    }

    public void stop() throws Exception {
        cacheManager.releaseCache("my-cache-config");
    }
}

```

24.2.2. Deployment Via a -service.xml File

You can also deploy a JBoss Cache instance as an MBean service via a **-service.xml** file. This is done through the **code** attribute in the **mbean** element, and the **org.jboss.cache.jmx.CacheJmxWrapper**, like so:

```

<?xml version="1.0" encoding="UTF-8"?>

<server>
  <mbean code="org.jboss.cache.jmx.CacheJmxWrapper"
        name="foo:service=ExampleCacheJmxWrapper">

    <attribute name="TransactionManagerLookupClass">
      org.jboss.cache.transaction.JBossTransactionManagerLookup
    </attribute>

    <attribute name="MuxChannelFactory"><inject
bean="JChannelFactory"/></attribute>

    <attribute name="MultiplexerStack">udp</attribute>
    <attribute name="ClusterName">Example-EntityCache</attribute>
    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
    <attribute name="CacheMode">REPL_SYNC</attribute>
    <attribute name="InitialStateRetrievalTimeout">15000</attribute>
    <attribute name="SyncReplTimeout">20000</attribute>
    <attribute name="LockAcquisitionTimeout">15000</attribute>
    <attribute name="ExposeManagementStatistics">true</attribute>

  </mbean>
</server>

```

The **CacheJmxWrapper** is not the cache itself (that is, you can't store stuff in it). Rather, as its name implies, it's a wrapper around an **org.jboss.cache.Cache** that handles integration with JMX. **CacheJmxWrapper** exposes the **org.jboss.cache.Cache** via its **CacheJmxWrapperMBean** MBean interfaces **Cache** attribute; services that need the cache can obtain a reference to it via that attribute.

24.2.3. Deployment Via a -jboss-beans.xml File

Much like it can deploy MBean services described with a **-service.xml**, JBoss Enterprise Web Platform can also deploy services that consist of Plain Old Java Objects (POJOs) if the POJOs are described using the JBoss Microcontainer schema in a **-jboss-beans.xml** file. You create such a file and deploy it, either directly in the **deploy** directory, or packaged in an EAR or SAR. Following is an example:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <!-- First we create a Configuration object for the cache -->
  <bean name="ExampleCacheConfig"
        class="org.jboss.cache.config.Configuration">

    <!-- Externally injected services -->
    <property name="runtimeConfig">
      <bean name="ExampleCacheRuntimeConfig"
            class="org.jboss.cache.config.RuntimeConfig">
        <property name="transactionManager">
          <inject bean="jboss:service=TransactionManager"
                  property="TransactionManager"/>
        </property>
        <property name="muxChannelFactory"><inject
bean="JChannelFactory"/></property>
      </bean>
    </property>

    <property name="multiplexerStack">udp</property>
    <property name="clusterName">Example-EntityCache</property>
    <property name="isolationLevel">REPEATABLE_READ</property>
    <property name="cacheMode">REPL_SYNC</property>
    <property name="initialStateRetrievalTimeout">15000</property>
    <property name="syncReplTimeout">20000</property>
    <property name="lockAcquisitionTimeout">15000</property>
    <property name="exposeManagementStatistics">true</property>

  </bean>

  <!-- Factory to build the Cache. -->
  <bean name="DefaultCacheFactory" class="org.jboss.cache.DefaultCacheFactory">
    <constructor factoryClass="org.jboss.cache.DefaultCacheFactory" />
  </bean>

  <!-- The cache itself -->
  <bean name="ExampleCache" class="org.jboss.cache.Cache">
    <constructor factoryMethod="createCache">
      <factory bean="DefaultCacheFactory"/>
      <parameter class="org.jboss.cache.config.Configuration"><inject
bean="ExampleCacheConfig"/></parameter>
      <parameter class="boolean">>false</false>
    </constructor>
  </bean>

  <bean name="ExampleService" class="org.foo.ExampleService">
    <property name="cache"><inject bean="ExampleCache"/></property>
  </bean>

</deployment>
```

The bulk of the above is the creation of a JBoss Cache **Configuration** object; this is the same as what we saw in the configuration of the **CacheManager** service (see [Section 24.1.1, "Editing the](#)

[CacheManager Configuration](#)). In this case we're not using the **CacheManager** service as a cache factory, so instead we create our own factory bean and then use it to create the cache (the **ExampleCache** bean). The **ExampleCache** is then injected into a (fictitious) service that needs it.

An interesting thing to note in the above example is the use of the **RuntimeConfig** object. External resources like a **TransactionManager** and a JGroups **ChannelFactory** that are visible to the microcontainer are dependency injected into the **RuntimeConfig**. The assumption here is that in some other deployment descriptor in the Enterprise Web Platform, the referenced beans have already been described.

Using the configuration above, the **ExampleCache** cache will not be visible in JMX. Here's an alternate approach that results in the cache being bound into JMX:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <!-- First we create a Configuration object for the cache -->
  <bean name="ExampleCacheConfig"
        class="org.jboss.cache.config.Configuration">

    .... same as above

  </bean>

  <bean name="ExampleCacheJmxWrapper"
        class="org.jboss.cache.jmx.CacheJmxWrapper">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
      (name="foo:service=ExampleCacheJmxWrapper",
        exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
        registerDirectly=true)
    </annotation>

    <property name="configuration"><inject
      bean="ExampleCacheConfig"/></property>

  </bean>

  <bean name="ExampleService" class="org.foo.ExampleService">
    <property name="cache"><inject bean="ExampleCacheJmxWrapper"
      property="cache"/></property>
  </bean>

</deployment>
```

Here the **ExampleCacheJmxWrapper** bean handles the task of creating the cache from the configuration. **CacheJmxWrapper** is a JBoss Cache class that provides an MBean interface for a cache. Adding an **<annotation>** element binds the JBoss Microcontainer **@JMX** annotation to the bean; that in turn results in JBoss Enterprise Web Platform registering the bean in JMX as part of the deployment process.

The actual underlying **org.jboss.cache.Cache** instance is available from the **CacheJmxWrapper** via its **cache** property; the example shows how this can be used to inject the cache into the **ExampleService**.

Part IV. Performance Tuning

Chapter 25. JBoss Enterprise Web Platform 5 Performance Tuning

25.1. Introduction

Performance can be affected by application design, hardware, network profile, operating system, application development, testing and deployment. Performance tuning ensures that your application does not consume unnecessary resources. Read this chapter to understand how to make your applications and your Enterprise Web Platform instance as efficient as possible.

25.2. Hardware tuning

To develop a suitable hardware configuration that suits the performance of your applications on the JBoss Enterprise Web Platform, you need to understand the impact that the selected hardware configuration may have on other applications and overall operating system performance.

To understand hardware performance tuning issues, it is also very critical to understand the hardware architecture of your system.

25.2.1. CPU (Central Processing Unit)

The CPU is the central processing unit of your computer. It consists of:

- a control unit which receives and decides what type of instructions it has received,
- CPU registers that store intermediate processing information temporarily,
- a program counter which holds the location of the succeeding executable tasks,
- instruction register that stores currently executing tasks,
- CPU cache which is a limited memory that holds data currently being processed by the CPU.

Understanding your CPU architecture can help you identify your CPU specifications. For AMD CPUs, refer to http://www.amd.com/us-en/Processors/ProductInformation/0,30_118,00.html. For Intel CPUs, refer to http://www.intel.com/products/processor/index.htm?iid=subhdr+prod_proc.

25.2.2. RAM (Random Access Memory)

Random access memory (RAM) is the next level of storage that can be used to hold executing programs and/or data. RAM chips provide a higher amount of storage than the CPU cache and can improve computer performance. Storing data or programs frequently used in RAM can greatly improve performance as they can be retrieved faster than from the hard disk drives.

RAM is crucial for example when tuning your database management system to manage buffer cache. This involves storing frequently used database information in RAM for quick application access while being careful not to negatively affect overall operating system performance or the performance of other applications.

25.2.3. Hard Disk

Unlike the CPU and RAM, hard disk drives do not require a power source to retain information/data. In case of power loss, information stored in the CPU and RAM is lost while that stored in the hard disk is retained but may be corrupted depending on the type of operation that was in progress during the power loss.

Retrieval and storage of information from disk drives takes much longer as they use mechanical heads to read and write information to the cylinders of the disk. Storage areas in RAM and in the CPU can be accessed with equal speed while on the hard disk, movement of the disk head to the requested disk block/blocks where information is stored is necessary.

Practices such as disk defragmentation and cleanups can help improve file retrieval and overall

performance of your applications. It is therefore crucial to manage the disk storage carefully with the retrieval and processing of data in mind. You also need to identify a suitable file system for your operating system to ensure the best performance possible.

Understanding the main architectural differences and issues that may occur with different computer hardware profiles can help identify a suitable hardware performance and disaster management strategy that would be suitable for your needs.

25.3. Operating System Performance Tuning

Most modern operating systems now ship with performance tuning or profiling tools that can help you monitor CPU, memory, hard disk and network usage in real-time.

On Windows the task manager and performance monitor can be helpful in identifying system performance bottlenecks while in Unix based operating systems **top** and **ps** are used for the same purpose. Linux distributions such as Red Hat Enterprise Linux and Fedora provide a graphical user interface **System Monitor** that is useful to monitor system performance.

Operating system performance tuning is about resource management to respond to individual requests. Managing operating system scalability on the other hand involves managing resource consumption with varying volumes (low to very high) of requests.

Overall operational performance metrics that are critical for the business such as response time to user requests, database, network, CPU and memory performance among other metrics should be identified and tested and logged in real-time where possible or with system deployments

For clustered environments, understanding and monitoring your cluster's performance and identifying overloads early is critical to system failure prevention.

25.3.1. Networking

Network configurations may contribute to performance bottlenecks and may be hard to detect. For example a user may get an error on their browser when trying to load a web application on a dial up connection while the same page may load on a broadband internet connection. The main issue in this scenario may be bandwidth and may not be obviously displayed in the error message displayed.

Identifying network architecture and infrastructure is therefore critical in performance tuning and fixing system bottlenecks.

Most modern operating systems provide you with network hardware configuration tools. Some hardware manufacturers may also provide extended network hardware configuration tools with their drivers.

Most operating systems support different communication protocols which you can tweak. Factors such as TCP buffer memory space, connection buffer limits and acknowledgment options among others should be take into account in your network design.

Deciding to turn DNS lookup on or off in your web servers can also affect your performance but may be necessary to turn on for high security environments. Factoring this and allocating necessary resources or hardware can help improve system performance.

25.4. Tuning the JVM

For Java-based applications, it is recommended to also be familiar with tuning of your Java Virtual Machine (JVM). Some key aspects of your JVM that need tweaking include managing out-of-memory exceptions, Java heap settings and garbage collection. Please refer to the JDK 6 documentation on <http://java.sun.com/j2se/1.6.0/docs/> for further information.

25.5. Tuning your applications

Good application design and development practices are critical to ensuring satisfactory application performance. Data reads or writes and processing by your applications may cause performance bottlenecks due to factors such as timeouts on remote servers memory allocation or network issues among other factors. Understanding how each application works is therefore crucial in identifying performance bottlenecks. Setting expected time duration each code part is expected to take can help develop realistic benchmarks against which the applications can be reviewed. These benchmarks should take into account high and low peak usage times for the applications and not averages as these may highly vary from the peak times.

In addition, using benchmarking tools to test your applications may be a quick way to pinpoint issues in your code which can often be causes for performance bottlenecks. Iterative tests are recommended to identify cache and other hardware issues that may arise due to start up or other factors.

The JBoss Enterprise Web Platform web console, <http://localhost:8080/web-console/>, provides you with monitoring tools starting with the JVM environment statistics on the default page and access to monitoring tools and snapshots.



Performance Monitors and Profilers

A performance monitor informs you on overall application performance such as requests per second. Profiling tools such as [JBoss Profiler](#) will tell you how long it is taking your application to service a request, and how often it services certain types of requests. This can usually be broken down all the way to the individual methods. For example, how many times a method was called and the average/maximum/minimum amount of time spent in the method.

It is also important not to create bottlenecks for other applications while fixing a performance issue in one application.

25.5.1. Instrumentation

Applications should be instrumented for performance analysis. In most cases, the actual production workload differs from the expected workload. Without instrumentation of your applications, you will lack accurate tracking data. Workloads on your applications can also change over time, as the business size, models or environment changes.

In the past, instrumentation had to be embedded in the application. Today, there are many solutions for instrumentation that do not require developers to code. Commercial products, and the JBoss AOP framework can be used for just this purpose. You can also turn on call statistics in the containers, and Hibernate statistics. For more on this please refer to the AOP and Hibernate project pages.

Taking successive thread dumps (includes the current call stack for each Java Enterprise Web Platform thread) can show application developers what is happening in the application. If an application has hit a performance wall, and the performance issue lasts for five minutes, generating a thread dump every minute may help solve the problem. You can use the JVM `jps -l` command to get a list of running Java applications and the process IDs for each. Note the process ID for the `org.jboss.Main` application. Then run the `jstack ProcessID` command (replacing ProcessID with the `org.jboss.Main` process ID) to generate the thread dump. Of course, you should redirect the output of the `jstack` command to save the output: `jstack ProcessID > threaddump1.txt`.

25.6. Tuning JBoss Enterprise Web Platform

Before tuning the JBoss Enterprise Web Platform, please ensure that you are familiar with its components, as outlined in the introduction section of this book. You should also be familiar with any particular services your application may use on the server and tune them to improve performance. It is also important to establish optimal database connections used by your applications and set these on the server. This section discusses these among other JBoss Enterprise Web Platform performance tuning topics.

25.6.1. Memory usage

Memory usage of Java applications including the JBoss Enterprise Web Platform is dictated by the heap space allocated. You could, for example, reduce the currently allocated 1 gigabyte heap space to 800 megabytes to reduce memory footprint (if you have enough headroom).

The Java Virtual Machine (JVM) manages segments generations of memory. If a segment of the heap space is exhausted, you will see a Java OutOfMemoryError (OOM). The application should be restarted to correct any bad state. If the available memory is insufficient, increase the maximum Java memory size. You may need to switch to a 64-bit JVM.

An OOME is also thrown when the permanent memory is exhausted. Permanent memory is not part of the heap. It is a JVM-specific area of memory where information about loaded classes is maintained. If you have a large number of classes, this area may not be sufficient and your applications will fail to deploy or redeploy. The default value for the **-server** switch is 64 megabytes. You can increase permanent memory space to avoid OOMs like so:

```
-XX:MaxPermSize=256m
```

This adds 256 megabytes of permanent space to the 512 megabytes originally provided by the heap for a total of 768 megabytes. Remember that the JVM consumes system memory, and that stack space is also consumed on a per-thread basis (size varies with operating system).

The total memory allocated from the system in the following is 768 megabytes:

```
-XX:MaxPermSize=256m -Xmx512m
```

This is not the total size of the VM and does not include the space the VM allocates for the *C heap* or stack space.



HotSpot JVM

The HotSpot JVM consists of various garbage collection tools which you can use to collect garbage collection information that you can use to tune your applications. You can find more information on the HotSpot Virtual machine on <http://java.sun.com/javase/technologies/hotspot/>.



Monitoring Tools for Java 6

Java 6 includes new tools that help monitor Java applications. **Jmap** can generate a heap dump file (<http://java.sun.com/javase/6/docs/technotes/tools/share/jmap.html>) that can easily be read by the Eclipse Memory Analyzer tool (<http://www.eclipse.org/mat/>). The **jstat** tool (<http://java.sun.com/javase/6/docs/technotes/tools/share/jstat.html>) can help give you a precise picture of your permanent memory space and the other segments on the Java memory heap.

25.6.1.1. VFS Tuning

Most tuning options for the Virtual File System (VFS) can be found in the **VFSutils** class. Its string constants point us to different possible system property settings we can use to configure VFS behavior:

jboss.vfs.forceCopy

Defines how nested JARs are handled. If **true** (the default), a temporary copy of the nested JAR is created and the VFS is re-wired accordingly. If **false**, nested JARs are handled in-memory. No temporary copy is created, but this option consumes more memory.

jboss.vfs.forceNoReaper

Set to **false** by default. This specifies that JAR files are closed asynchronously by a separate reaper thread, which can improve performance. To close JAR files synchronously, force no usage of the reaper thread by setting this to **true**. This can also be defined using the URI query and the **noReaper** query section.

jboss.vfs.forceCaseSensitive

When **true**, forces differentiation between lower and upper case file paths. The default value is **false**.

jboss.vfs.optimizeForMemory

When **true**, re-orders in-memory JAR handling to reduce memory consumption. The default value is **false**.

jboss.vfs.cache

Define this class (**org.jboss.virtual.spi.cache.helpers.NoopVFSCache**) to reuse existing temporary files so that unpacking and wiring does not to be repeated. The VFS registry uses this class definition to keep its existing VFS roots. Every **VirtualFile** lookup from the VFS class uses this singleton cache instance to check for an existing matching cache entry. Matching also considers any existing ancestor that lets you use the same **VirtualFile** instance.

25.6.1.1.1. VFS Cache Tuning

The VFS cache holds VFS roots, from which any **VirtualFile** lookup can access existing **VirtualFile** instances. This is especially useful in the case of temporary files (created from nested JARs), since multiple unpackings for nested JAR file related resources are not required.

By default there is no caching in VFS, since **org.jboss.virtual.spi.cache.helpers.NoopVFSCache** is used. You can provide your own cache implementation or choose from existing VFS implementations.

The cache implementations available as part of the **org.jboss.virtual.plugins.cache** package are:

SoftRefVFSCache

Uses soft reference as a map's entry value.

WeakRefVFSCache

Uses weak reference as a map's entry value.

TimedVFSCache

Evicts cache entries after the **defaultLifetime**.

LRUVFSCache

Evicts cache entries based on LRU, keeping minimum and maximum entries.

CombinedVFSCache

Holds few permanent roots. Any new root is cached in its **realCache** property.

By default, JBoss Enterprise Web Platform uses **CombinedVFSCache**. The cache type is configured in JBoss Microcontainer's bean configuration file, **\$JBoss_HOME/server/\$PROFILE/conf/bootstrap/vfs.xml**, like so:

```
<bean name="VFSCache">
  <constructor factoryClass="org.jboss.virtual.spi.cache.VFSCacheFactory"
factoryMethod="getInstance">
    <!-- Use the CombinedVFSCache implementation -->
    <parameter>org.jboss.virtual.plugins.cache.CombinedVFSCache</parameter>
  </constructor>
  <start ignored="true"/>
  <property name="permanentRoots">
    <map keyClass="java.net.URL"
valueClass="org.jboss.virtual.spi.ExceptionHandler">
      <entry>
        <key>${jboss.lib.url}</key>
        <value><null/></value>
      </entry>
      <entry>
        <key>${jboss.common.lib.url}</key>
        <value><inject bean="VfsNamesExceptionHandler"/></value>
      </entry>
      <entry>
        <key>${jboss.server.lib.url}</key>
        <value><inject bean="VfsNamesExceptionHandler"/></value>
      </entry>
      <entry>
        <key>${jboss.server.home.url}deploy</key>
        <value><inject bean="VfsNamesExceptionHandler"/></value>
      </entry>
    </map>
  </property>
  <property name="realCache">
    <bean class="org.jboss.virtual.plugins.cache.IterableTimedVFSCache"/>
  </property>
</bean>
```

Any new custom VFS root (for example, an additional **deploy** directory) should be added to this configuration.

25.6.1.1.2. Annotation Scanning Tuning

There are currently three ways to limit resources scanning:

- Provide a **ScanningMetadata** through XML or programmatically.
- Add a new deployment filter to **GenScanDeployer** bean in **deployers/metadata-deployer-jboss-beans.xml**.
- Modify **JBossCustomDeployDUF** in **deployers/metadata-deployer-jboss-beans.xml**.

ScanningMetadata can come from the **jboss-scanning.xml** file placed in your **META-INF** directory. This is a simple example of this file:

```
<scanning xmlns="urn:jboss:scanning:1.0">
  <path name="myejbs.jar">
    <include name="com.acme.foo"/>
    <exclude name="com.acme.foo.bar"/>
  </path>
  <path name="my.war/WEB-INF/classes">
    <include name="com.acme.foo"/>
  </path>
</scanning>
```

Here you list the paths inside your deployment, and which packages to include or exclude. If there is no explicit include, everything that is not excluded is included. If there is no path element at all, everything is excluded, as in the following example.

```
<scanning xmlns="urn:jboss:scanning:1.0">
<!-- Purpose: Disable scanning for annotations in contained deployment. -->
</scanning>
```

Another way to limit scanning is to provide the **jboss-classloading.xml** file, which contains classloader metadata.

25.6.2. Database Connection

Database performance tuning involves changing the initial database conceptual schema to improve performance. Irrespective of type, overall database management system performance tuning involves effective and efficient use of your hardware (hard disk, CPU and RAM) and improving database reads and writes.

Resource limits set by your operating system may also set limits on your database management system. A database administrator can analyze a database and identify performance bottlenecks by taking the above factors into consideration and adjusting the necessary database management system parameters such as writing dirty buffers to disk, checkpoints and log file rotations. In some instances hardware upgrades may also be necessary to improve database performance.

Database connections can be costly to establish and manage. Applications that create new connections to the database with every transaction or query and then close that connection add a great deal of overhead. Having a very small connection pool will also throttle the applications as the JBoss Enterprise Web Platform by default queues the request for a default of 30,000 milliseconds (30 seconds) before cancellation and throwing an exception.

We recommend reliance on data source definitions you can setup in the deploy directory of the JBoss Enterprise Web Platform and utilizing the connection pool settings. Connection pooling in the JBoss Enterprise Web Platform allows you to easily monitor your connection usage from the JMX console to determine proper sizing. Your database management system may also shipped with tools that allow you to monitor connections.

Depending on the databases implemented, please ensure you create a data source file in the deploy directory of your configuration as shown below:

```
$JBOSS_HOME/server/$PROFILE/deploy/
```

The filename should be of the form **\$DATABASE_NAME-ds.xml**.



Examples

Examples of datasource definition files for external databases can be found in the **\$JBoss_HOME/docs/examples/jca** directory.

25.6.3. Clustering Tuning

If you are running your application in a cluster, particularly if it involves high volume state replication around the cluster, there are a number of possible performance optimizations. As with any performance optimizations, always load test your application before and after making changes to verify the change has the intended effect, and make one change at a time so it's clear what change has what effect.

25.6.3.1. Ensuring Adequate Network Buffers

The standard clustered services in the Enterprise Web Platform use UDP for intra-cluster communication, in order to take advantage of UDP-based IP multicast. A downside to the use of UDP is some of the lossless transmission guarantees that are provided at the OS network level with TCP instead need to be implemented in Java code. In order to achieve peak performance it is important to reduce the frequency of UDP packets being dropped in the network layers. A frequent cause of lost packets is inadequately sized network buffers on the machines that are hosting the cluster nodes. The Enterprise Web Platform clustering code will *request* adequately sized read and write buffers from the OS when it opens sockets, but most operating systems (Windows seems to be an exception) will only *provide* buffers up to a maximum size. This maximum read and write buffer sizes are configurable at the OS level, and the default values are too low to allow peak performance. So, a simple tuning step is to configure your OS to allow buffers up to the size the Enterprise Web Platform clustering code will request.

The specific configuration steps needed to increase the maximum allowed buffer sizes are OS specific. See your OS documentation for instructions on how to increase these. For Linux systems, maximum values for these buffers sizes that will survive machine restarts can be set by editing the **/etc/sysctl.conf** file:

```
# Allow a 25MB UDP receive buffer for JGroups
net.core.rmem_max = 26214400
# Allow a 1MB UDP send buffer for JGroups
net.core.wmem_max = 1048576
```

25.6.3.2. Isolating Intra-Cluster Traffic

If network resources are a bottleneck for your application, overall performance can be improved by isolating intra-cluster traffic from external request traffic. This requires multiple NICs on your server machines, with request traffic coming in on one NIC and intra-cluster traffic using another. Once you have the hardware set up, it is easy to tell the Enterprise Web Platform nodes to use a different interface for the intra-cluster traffic:

```
./run.sh -c production -b 10.0.0.104 -Djgroups.bind_addr=192.168.100.104
```

In the above example, the **-Djgroups.bind_addr** setting tells the the Enterprise Web Platform to run intra-cluster JGroups traffic over the **192.168.100.104** interface. The **-b** switch specifies that all other traffic should use **10.0.0.104**.

25.6.3.3. JGroups Message Bundling

The JGroups group communication library used by the Enterprise Web Platform provides a feature known as *message bundling*. Message bundling is similar to nagling on a TCP socket — small messages are queued before sending (for up to a configurable maximum time) until a configurable number of bytes have accumulated, and then the queued messages are bundled and sent as one large

message. Use of bundling can have significant performance benefits for high-volume asynchronous session replication. However, it is not enabled by default, as bundling can add significant latency to other types of intra-cluster traffic, particularly clustered Hibernate/JPA Second Level Cache traffic.

If your application uses high volume session replication (web sessions or EJB3 stateful session beans), you might be able to increase performance by configuring the distributed caching layer to use a JGroups channel configured with message bundling enabled. This is done by editing the **\$JBOSS_HOME/server/\$PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-jboss-beans.xml** file. For example, for the cache used by default for web sessions:

```

. . .

<!-- Standard cache used for web sessions -->
<entry><key>standard-session-cache</key>
<value>
  <bean name="StandardSessionCacheConfig"
    class="org.jboss.cache.config.Configuration">

    . . .

    <!-- Replace standard 'udp' JGroups stack with
         one that uses message bundling -->
    <property name="multiplexerStack">udp-async</property>

    . . .

```

For FIELD granularity web sessions, in the same file the same change can be made to the cache configuration with the **field-granularity-session-cache** key. For EJB3 stateful session beans, in the same file the same change can be made to the cache configuration with the **sfsb-cache** key.



Note

Using the **udp-async** JGroups protocol stack for the session caches means an additional JGroups transport protocol will be used. This means additional sockets will be opened compared to a standard Enterprise Web Platform installation.

25.6.3.4. Enabling Buddy Replication for Session Caches

If your application involves high volume replication of web sessions or EJB3 stateful session beans in a cluster of more than two nodes, you can improve performance by enabling "buddy replication" in the web session and stateful session bean caches. With buddy replication, instead of replicating a copy of sessions to all nodes in the cluster, a copy is only replicated to a configurable number of "buddy" nodes.

Buddy replication is enabled done by editing the **\$JBOSS_HOME/server/\$PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-jboss-beans.xml** file. For example, for the cache used by default for web sessions:

```

. . .

<!-- Standard cache used for web sessions -->
<entry><key>standard-session-cache</key>
<value>
  <bean name="StandardSessionCacheConfig"
class="org.jboss.cache.config.Configuration">

    . . .

    <property name="buddyReplicationConfig">
      <bean class="org.jboss.cache.config.BuddyReplicationConfig">

        <!-- Just set to true to turn on buddy replication -->
        <property name="enabled">true</property>

      </bean>
    </property>

  </bean>
</value>
</entry>

. . .

```

For FIELD granularity web sessions, in the same file the same change can be made to the cache configuration with the **field-granularity-session-cache** key. For EJB3 stateful session beans, in the same file the same change can be made to the cache configuration with the **sfsb-cache** key.

25.6.3.5. Reducing the Volume of Web Session Replication

If your application is configured for web session replication, reducing the amount of data being replicated can obviously improve performance. You can do this by not replicating data when a request has not updated the session, and by limiting replication only to the changed session data. See [Section 21.2, “Configuring HTTP session state replication”](#) for a discussion on **replication-trigger** and **replication-granularity** and instructions for configuring your application to limit data replication.

25.6.3.6. Reducing the Volume of EJB3 Stateful Session Bean Replication

If your application is configured for EJB3 Stateful Session Bean replication, you can improve performance by avoiding replication after bean requests that haven't modified state. This can be controlled by having your bean class implement the **org.jboss.ejb3.cache.Optimized** interface. See [Section 19.2, “Stateful Session Beans in EJB 3.0”](#) for details.

25.6.3.7. Be Cautious with JPA/Hibernate Second Level Caching

JPA and Hibernate applications can often gain a performance boost by caching database data in the application server. However, with a clustered application the decision of whether or not to cache data is more complex than in the non-clustered case. This is because in a cluster, when database writes occur on one node, the caching layer needs to send a message to all other nodes in the cluster telling them to update or invalidate their cache content. For types of data that are frequently updated, the cost of the intra-cluster messages can outweigh the benefits of caching.

So, be sure to carefully load test your clustered application when deciding whether to store items in the Hibernate Second Level Cache. Avoid the temptation to turn on caching for all entity types; instead rank your entity types based on how infrequent writes of each type are and how likely it is that more than one transaction will read a particular entity. Then enable caching for one type at a time, testing for the performance impact.

Be doubly cautious about enabling caching of query result sets. When query caching is enabled, any time there is a database write, the clustered cache needs to send *two* messages around the cluster. These messages are used to ensure that any query results that may have been affected by the write are invalidated out of the cache. These messages need to be sent whether or not the entity type that has been written is itself cached. The cost of these messages can easily offset the benefit of query result caching. So, again, be sure to test the effect of caching.

25.6.3.8. Monitoring JGroups via JMX

When the Enterprise Web Platform clustering services create a JGroups **Channel** to use for intra-

cluster communication, they also register with the JMX server a number of MBeans related to that channel; one for the channel itself and one for each of its constituent protocols. For users interested in monitoring the performance-related behavior of a channel, a number of MBean attributes may prove useful.

jboss.jgroups:cluster=<cluster_name>,protocol=UDP,type=protocol

Provides statistical information on the sending and receipt of messages over the network, along with statistics on the behavior of the two thread pools used to carry incoming messages up the channel's protocol stack.

Useful attributes directly related to the rate of transmission and receipt include **MessagesSent**, **BytesSent**, **MessagesReceived** and **BytesReceived**.

Useful attributes related to the behavior of the thread pool used to carry ordinary incoming messages up the protocol stack include **IncomingPoolSize** and **IncomingQueueSize**. Equivalent attributes for the pool of threads used to carry special, unordered "out-of-band" messages up the protocol stack include **OOBPoolSize** and **OOBQueueSize**. Note that **OOBQueueSize** will typically be 0 as the standard JGroups configurations do not use a queue for OOB messages.

jboss.jgroups:cluster=<cluster_name>,protocol=UNICAST,type=protocol

Provides statistical information on the behavior of the protocol responsible for ensuring lossless, ordered delivery of unicast (that is, point-to-point) messages.

The ratio of **NumRetransmissions** to **MessagesSent** can be tracked to see how frequently messages are not being received by peers and need to be retransmitted. The **NumberOfMessagesInReceiveWindows** attribute can be monitored to track how many messages are queueing up on a recipient node waiting for a message with an earlier sequence number to be received. A high number indicates messages are being dropped and need to be retransmitted.

jboss.jgroups:cluster=<cluster_name>,protocol=NAKACK,type=protocol

Provides statistical information on the behavior of the protocol responsible for ensuring lossless, ordered delivery of multicast (that is, point-to-multipoint) messages.

Use the **XmitRequestsReceived** attribute to track how often a node is being asked to retransmit a messages it sent; use **XmitRequestsSent** to track how often a node needs to request retransmission of a message.

jboss.jgroups:cluster=<cluster_name>,protocol=FC,type=protocol

Provides statistical information on the behavior of the protocol responsible for ensuring fast message senders do not overwhelm slow receivers.

Attributes useful for monitoring whether threads seeking to send messages are having to block while waiting for credits from receivers include **Blockings**, **AverageTimeBlocked** and **TotalTimeBlocked**.

25.6.4. Other key configurations

Other key configurations required for performance tuning of your Enterprise Web Platform include the **\$JBOSS_HOME/server/\$PROFILE/deployers/jbossweb.deployer/server.xml** file that sets your HTTP requests pool.

JBoss Enterprise Web Platform 5 has a robust thread pooling, that should be sized appropriately. The server has a **jboss-service.xml** file in the **\$JBOSS_HOME/server/\$PROFILE/conf** directory that

defines the system thread pool. There is a setting that defines the behavior if there isn't a thread available in the pool for execution. The default is to allow the calling thread to execute the task. You can monitor the queue depth of the system thread pool through the JMX Console, and determine from that if you need to make the pool larger.

The new administration console can be used for configuring and managing different aspects of the Enterprise Web Platform environment.

The **default** configuration is appropriate for development, but not necessarily for a production environment. In the default configuration, console logging is enabled. Console logging is ideal for development, especially within the IDE, as you get all the log messages to show in the IDE console view. In a **production** environment, console logging is very expensive and is not recommended. Turn down the verbosity level of logging if its not necessary. Please note that the less you log, the less I/O will be generated, and the better the overall throughput will be.

Other performance tuning aspects include Caching, Clustering and Replication which are discussed in the respective Chapters in this book.

Part V. Appendices

Vendor-Specific Datasource Definitions

This appendix includes datasource definitions for databases supported by JBoss Enterprise Application Platform.

A.1. Deployer Location and Naming

All database deployers should be saved to the **\$JBoss_HOME/server/default/deploy/** 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.

A.2. DB2

Example A.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 A.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 A.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] Hostname 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 A.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/fajjdbc.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] Hostname 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**.

A.3. Oracle

Example A.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 loubyansky 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.920/a9665
      4/toc.htm

    <connection-url>jdbc:oracle:oci:@youroracle-tns-name</connection-url>
    or
    <connection-
url>jdbc:oracle:oci:@(description=(address=(host=youroraclehost)(protocol=tcp)(po
rt=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</exception-
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 A.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 loubyansky 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</exception-
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 standardjbosscmp-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</depends>
  </mbean>

</datasources>

```

Example A.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)(a
ddress=(protocol=tcp)(host=xxxxhost1)(port=1521))(address=(protocol=tcp)(host=xxx
xhost2)(port=1521)))(connect_data=(service_name=xxxxsid)(failover_mode=(type=sele
ct)(method=basic))))</connection-url>
...
```

**Note**

This example has only been tested against Oracle 10g.

A.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/>`.

A.3.2. Type Mapping for Oracle 10g

You need to specify Oracle9i type mapping for Oracle 10g datasource configurations.

Example A.8. Oracle9i Type Mapping

```
....
<metadata>
  <type-mapping>Oracle9i</type-mapping>
</metadata>
....
```

A.3.3. Retrieving the Underlying Oracle Connection Object**Example A.9. Oracle Connection Object**

```
Connection conn = myJBossDatasource.getConnection();
WrappedConnection wrappedConn = (WrappedConnection)conn;
Connection underlyingConn = wrappedConn.getUnderlyingConnection();
OracleConnection oracleConn = (OracleConnection)underlyingConn;
```

A.3.4. Limitations of Oracle 11g

In Oracle 11g R2 (both RAC and standalone), a complex query with `LockMode.UPGRADE` (ie: "for update") may cause a "No more data to read from socket" error. The workaround is to not use `LockMode.UPGRADE` on such queries. See Oracle bug number 9219636 for more details.

A.4. Sybase

Example A.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 additional 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</exception-
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 standardjbosscomp-jdbc.xml (optional)
-->
    <metadata>
      <type-mapping>Sybase</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

[1]

A.4.1. Sybase Limitations

Sybase has some configuration anomalies, which you should be aware of.

DDL statements in transactions

Hibernate, which is an integral part of the Enterprise Platform, allows the SQL Dialect to decide whether or not the database supports DDL statements within a transaction. Sybase does not override this. the default is to query the JDBC metadata to see whether DDL is allowed within transactions. However, Sybase does not correctly report whether it is set up to use this option.

Sybase recommends against using DDL statements in transactions, because of locking issues. Review the Sybase documentation for how to enable or disable the **ddl in tran** option.

Sybase does not throw an exception if a value overflows the constraints of the underlying column.

Sybase ASE does not throw an exception when Parameterized Sql is in use. **jconn3.jar** uses Parameterized Sql for insertion by default, so no exception is thrown if a value overflows the constraints of the underlying column. Since no exception is thrown, Hibernate cannot tell that the insert failed. By using Dynamic Prepare instead of Parameterized SQL, ASE throws an

exception. Hibernate can catch this exception and act accordingly.

For that reason, set the **Dynamic prepare** parameter to **true** in Hibernate's configuration file.

```
<property name="connection.url">jdbc:sybase:Tds:aurum:1503/masterDb?
DYNAMIC_PREPARE=true</property>
```

jconn4.jar uses Dynamic Prepare by default.

SchemaExport cannot create stored procedures in chained transaction mode

On Sybase, SchemaExport cannot be used to create stored procedures while in while in chained transaction mode. The workaround for this case is to add the following code immediately after the definition of the new stored procedure:

```
<database-object>
  <create>
    sp_procxmode paramHandling, 'chained'
  </create>
  <drop/>
</database-object>
```

A.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 A.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</connection-
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 A.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 standardjbosscomp-jdbc.xml (optional)
-->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Example A.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 standardjbosscomp-jdbc.xml (optional)
-->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>
</datasources>
```

A.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 A.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;DatabaseName=p
ubs</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 A.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 A.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 standardjbosscomp-jdbc.xml (optional)
-->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>
</datasources>

```

A.5.2. JSQL Drivers**Example A.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>

```

A.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 A.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 A.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>

```

A.5.4. "Invalid object name 'JMS_SUBSCRIPTIONS' Exception

If you receive an exception like the one in [Example A.20, "JMS_SUBSCRIPTIONS Exception"](#) during startup, specify a **SelectMethod** in the connection URL, as shown in [Example A.21, "Specifying a SelectMethod"](#).

Example A.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(JDBCState
Manager.java:290)
        at
org.jboss.mq.server.JMSDestinationManager.addDestination(JMSDestinationManager.jav
a:656)

```

Example A.21. Specifying a SelectMethod

```

<connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;DatabaseName=j
boss</connection-url>

```

A.5.5. New dialect for Microsoft SQL Server 2008 with JDBC 3.x drivers

To use Hibernate with Microsoft SQL Server 2008 and JDBC 3.x drivers, you need to introduce a new dialect like the following:

```

public class SQLServer2008Dialect extends SQLServerDialect {
    public SQLServer2008Dialect(){
        registerColumnType( Types.DATE, "date" );
        registerColumnType( Types.TIME, "time" );
        registerColumnType( Types.TIMESTAMP, "datetime2" );
        registerFunction( "current_timestamp", new
NoArgSQLFunction("current_timestamp", Hibernate.TIMESTAMP, false) );
    }
}

```

A.6. MySQL Datasource**A.6.1. Installing the Driver****Procedure A.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_HOMEdocs/examples/jca/mysql-ds.xml** example datasource deployer file to **\$JBoss_HOME/server/config_name/deploy/**, for use as a template.

MySQL limitations**Millisecond and microsecond measurements**

MySQL does not currently support millisecond and microsecond measurements when returning database values such as **TIME** and **TIMESTAMP**. Tests which rely on these measurements will fail.

A.6.2. MySQL Local-TX Datasource

Example A.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>
```

A.6.3. MySQL Using a Named Pipe

Example A.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>
```

Example A.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 8.0</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

**Important**

XA Transactions are denied if ***max_prepared_transactions*** uses the default value (0) in PostgreSQL v8.4 and v8.2. PostgreSQL user documentation recommends you set the ***max_prepared_transactions*** value to meet or exceed the value of ***max_connections*** so every session can have a prepared transaction pending.

For more information, refer to the PostgreSQL v8.4 User Documentation, located at <http://www.postgresql.org/docs/8.4/interactive/runtime-config-resource.html#GUC-MAX-PREPARED-TRANSACTIONS>

Example A.25. PostgreSQL XA Datasource

This configuration 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>
```

A.8. Ingres**Example A.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>

Logging Information and Recipes

B.1. Log Level Descriptions

Table B.1, “[log4j Log Level Definitions](#)” lists the typical meanings for different log levels in **log4j**. Your application may interpret these levels differently, depending on your choices.

Table B.1. log4j Log Level Definitions

log4j Level	JDK Level	Description
FATAL		The Application Service is likely to crash.
ERROR	SEVERE	A definite problem exists.
WARN	WARNING	Likely to be a problem, but may be recoverable.
INFO	INFO	Low-volume detailed logging. Something of interest, but not a problem.
DEBUG	FINE	Low-volume detailed logging. Information that is probably not of interest.
	FINER	Medium-volume detailed logging.
TRACE	FINEST	High-volume detailed logging.



Note

The more verbose logging levels are not appropriate for production systems, because of the high level of output they generate.

Example B.1. Restricting Logged Information to a Specific Log Level

```
<!-- Show the evolution of the DataSource pool in the logs
[inUse/Available/Max]-->
<category name="org.jboss.resource.connectionmanager.JBossManagedConnectionPool">
  <priority value="TRACE" class="org.jboss.logging.XLevel"></priority>
</category>
```

B.2. Separate Log Files Per Application

To segregate logging output per application, assign **log4j** categories to specific appenders. This is typically done in the **conf/log4j.xml** deployment descriptor.

Example B.2. Filtering App1 Log Output to a Separate File

```

<appender name="App1Log" class="org.apache.log4j.FileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"></errorHandler>
  <param name="Append" value="false"/>
  <param name="File" value="${jboss.server.home.dir}/log/app1.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
  </layout>
</appender>

...

<category name="com.app1">
  <appender-ref ref="App1Log"></appender-ref>
</category>
<category name="com.util">
  <appender-ref ref="App1Log"></appender-ref>
</category>

```

Example B.3. Using TCLMCFilter

Enterprise Platform 5.1 includes the new class `org.jboss.logging.filter.TCLMCFilter`, which allows you to filter based on the deployment URL.

```

<appender name="App1Log" class="org.apache.log4j.FileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"></errorHandler>
  <param name="Append" value="false"/>
  <param name="File" value="${jboss.server.home.dir}/log/app1.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
  </layout>
  <filter class="org.jboss.logging.filter.TCLMCFilter">
    <param name="AcceptOnMatch" value="true"/>
    <param name="DeployURL" value="app1.ear"/>
  </filter>

  <!-- end the filter chain here -->
  <filter class="org.apache.log4j.varia.DenyAllFilter"></filter>

</appender>

```

B.3. Redirecting Category Output

When you increase the level of logging for one or more categories, it is often useful to redirect the output to a separate file for easier investigation. To do this you add an **appender-ref** to the category.

Example B.4. Adding an appender-ref

```
<appender name="JSR77" class="org.apache.log4j.FileAppender">
  <param name="File" value="{jboss.server.home.dir}/log/jsr77.log"/>
  ...
</appender>

<!-- Limit the JSR77 categories -->
<category name="org.jboss.management" additivity="false">
  <priority value="DEBUG"></priority>
  <appender-ref ref="JSR77"></appender-ref>
</category>
```

All **org.jboss.management** output goes to the **jsr77.log** file. The **additivity** attribute controls whether output continues to go to the root category appender. If **false**, output only goes to the appenders referred to by the category.

Revision History

Revision 5.1.1-103.33.400	2013-10-31	Rüdiger Landmann
Rebuild with publican 4.0.0		
Revision 5.1.1-103.33	2012-07-18	Anthony Towns
Rebuild for Publican 3.0		
Revision 5.1.1-100	Mon Jul 18 2011	Jared Morgan
Incorporated changes for JBoss Enterprise Web Platform 5.1.1 GA. For information about documentation changes to this guide, refer to <i>Release Notes 5.1.1</i> .		
Revision 5.1.0-105	Thu Sep 23 2010	Rebecca Newton
Revised for JBoss Enterprise Web Platform 5.1.0.GA		