JBoss Enterprise Application Platform 5.0

Administration And Configuration Guide

JBoss Community

JBoss Enterprise Application Platform 5.0 Administration And Configuration Guide

Author JBoss Community
Editor JBoss Community
Editor Isaac Rooskov

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588 Research Triangle Park, NC 27709 USA

This book is a guide to the administration and configuration of JBoss Enterprise Application Platform 5.0.

What this Book Covers xi

1. Introduction 1
1.1. JBoss Enterprise Application Platform Use Cases 2

I. JBoss Enterprise Application Platform Infrastructure 3

2. JBoss Enterprise Application Platform 5 architecture 5
2.1. The JBoss Enterprise Application Platform Bootstrap 9
2.2. Hot Deployment 9

II. JBoss Enterprise Application Platform 5 Configuration 11

3. Deployment 13
3.1. Deployable Application Types 13
3.2. Standard Server Profiles 14

4. Microcontainer 15
4.1. An overview of the Microcontainer modules 16
4.2. Configuration 17
4.3. References 18

5. Web Services 19
5.1. The need for web services 19
5.2. What web services are not 19
5.3. Jboss Web Services Attachment support with XOP (XML-binary Optimized Packaging) and SwA 20
5.4. Using SwaRef with JAX-WS endpoints 20
5.5. MTOM/XOP 21
5.6. Enabling MTOM per endpoint 22
5.6.1. The MTOM enabled SOAP 1.1 binding ID 22
5.7. Document/Literal 22
5.8. Document/Literal (Bare) 23
5.9. Document/Literal (Wrapped) 24
5.10. RPC/Literal 24
5.11. RPC/Encoded 26
5.12. Web Service Endpoints 26
5.13. Plain old Java Object (POJO) 26
5.14. The endpoint as a web application 26
5.15. Packaging the endpoint 27
5.16. Accessing the generated WSDL 27
5.17. EJB3 Stateless Session Bean (SLSB) 27
5.18. Endpoint Provider 28
5.19. WebServiceContext 29
5.20. Web Service Clients 30
5.20.1. Service 30
5.20.2. Dynamic Proxy 31
5.20.3. WebServiceRef 33
5.20.4. Dispatch 34
5.20.5. Asynchronous Invocations 35
5.20.6. Oneway Invocations 35
5.21. Common API 36
5.21.1. Handler Framework 36

5.21.2. Message Context 37
5.21.3. Fault Handling 38
5.22. DataBinding 39
5.22.1. Using JAXB with non annotated classes 39
5.23. Attachments 39
5.23.1. MTOM/XOP 39
5.23.2. SwaRef 41
5.24. Tools 43
5.24.1. Bottom-Up (Using wsprovide) 43
5.24.2. Top-Down (Using wsconsume) 46
5.24.3. Client Side 47
5.24.4. Command-line & Ant Task Reference 50
5.24.5. JAX-WS binding customization 50
5.25. Web Service Extensions 51
5.25.1. WS-Addressing 51
5.25.2. WS-BPEL 54
5.25.3. WS-Eventing 54
5.25.4. WS-Security 59
5.25.5. WS-Transaction 63
5.25.6. XML Registries 63
5.25.7. WS-Policy 69
5.26. JBossWS Extensions 73
5.26.1. Proprietary Annotations 73
5.27. Web Services Appendix 76
5.28. References 76

6. JBoss5 Virtual Deployment Framework 77
6.1. MainDeployerImpl 78
6.2. JBoss5StructureDeployerClasses 80
6.3. Deployer Helper and Base Classes 80
6.4. Current Deployers 82
6.5. Virtual File System JBoss5VirtualFileSystem 83

7. JBoss AOP 85
7.1. Some key terms 85
7.2. Creating Aspects in JBoss AOP 87
7.3. Applying Aspects in JBoss AOP 87

8. JBoss Transactions 89
8.1. Why you need JBoss Transactions 89
8.2. JBoss Transactions Java EE 5 Support 89
8.3. JBoss Transactions Web Services Support 90
8.4. How JBossTS address these issues 90

9. Remoting 93
9.1. Summary of JBoss Remoting Features 93
9.2. JBoss Remoting Configuration in the JBoss Enterprise Application Platform 94

10. JBoss Messaging 95
10.1. Configuring JBoss Messaging 95
10.1.1. Configuring the SecurityStore 95
10.1.2. SecurityStore Attributes 96
10.2. Configuring the ServerPeer 96
10.3. Server Attributes 99

10.3.1. ServerPeerID 99
10.3.2. DefaultQueueJNDIContext 99
10.3.3. DefaultTopicJNDIContext 99
10.3.4. PostOffice 99
10.3.5. DefaultDLQ 99
10.3.6. DefaultMaxDeliveryAttempts 100
10.3.7. DefaultExpiryQueue 100
10.3.8. DefaultRedeliveryDelay 100
10.3.9. MessageCounterSamplePeriod 100
10.3.10. FailoverStartTimeout 100
10.3.11. FailoverCompleteTimeout 100
10.3.12. DefaultMessageCounterHistoryDayLimit 100
10.3.13. ClusterPullConnectionFactory 101
10.3.14. DefaultPreserveOrdering 101
10.3.15. RecoverDeliveriesTimeout 101
10.3.16. SuckerPassword 101
10.3.17. StrictTCK 101
10.3.18. Destinations 101
10.3.19. MessageCounters 101
10.3.20. MessageCountersStatistics 101
10.3.21. SupportsFailover 101
10.3.22. PersistenceManager 102
10.3.23. JMSUserManager 102
10.3.24. SecurityStore 102
10.4. MBean operations of the ServerPeer MBean 102
10.4.1. DeployQueue 102
10.4.2. UndeployQueue 102
10.4.3. DestroyQueue 102
10.4.4. DeployTopic 102
10.4.5. UndeployTopic 103
10.4.6. DestroyTopic 103
10.4.7. ListMessageCountersHTML 103
10.4.8. ResetAllMesageCounters 103
10.4.9. ResetAllMesageCounters 103
10.4.10. EnableMessageCounters 103
10.4.11. DisableMessageCounters 103
10.4.12. RetrievePreparedTransactions 104
10.4.13. ShowPreparedTransactions 104

11. Use Alternative Databases with JBoss Enterprise Application Platform 105
11.1. How to Use Alternative Databases 105
11.2. Install JDBC Drivers 105
11.2.1. Special notes on Sybase 105
11.2.2. Configuring JDBC DataSources 106
11.3. Creating a DataSource for the External Database 111
11.4. Common configuration for DataSources and ConnectionFactorys 112
11.4.1. General 112
11.4.2. XA 112
11.4.3. Security parameters 112
11.5. Change Database for the JMS Services 114
11.6. Support Foreign Keys in CMP Services 114
11.7. Specify Database Dialect for Java Persistence API 114

11.8. Change Other JBoss Enterprise Application Platform Services to Use the External Database 115
11.8.1. The Easy Way 115
11.8.2. The More Flexible Way 116
11.9. A Special Note About Oracle DataBases 116
11.10. DataSource configuration 117
11.11. Parameters specific for java.sql.Driver usage 118
11.12. Parameters specific for javax.sql.XADataSource usage 118
11.13. Common DataSource parameters 118
11.14. Generic Datasource Sample 120
11.15. Configuring a DataSource for remote usage 122
11.16. Configuring a DataSource to use login modules 123

12. Pooling 125
12.1. Strategy 125
12.2. Transaction stickness 125
12.3. Workaround for Oracle 126
12.4. Pool Access 126
12.5. Pool Filling 126
12.6. Idle Connections 126
12.7. Dead connections 127
12.7.1. Valid connection checking 127
12.7.2. Errors during SQL queries 127
12.7.3. Changing/Closing/Flushing the pool 127
12.7.4. Other pooling 128

13. Frequently Asked Questions 129
13.1. I have problems with Oracle XA? 129

III. Clustering Guide 131

14. Introduction and Quick Start 133
14.1. Quick Start Guide 133
14.1.1. Initial Preparation 133
14.1.2. Launching a JBoss Enterprise Application Platform Cluster 135
14.1.3. Web Application Clustering Quick Start 137
14.1.4. EJB Session Bean Clustering Quick Start 138
14.1.5. Entity Clustering Quick Start 138

15. Clustering Concepts 141
15.1. Cluster Definition 141
15.2. Service Architectures 142
15.2.1. Client-side interceptor architecture 142
15.2.2. External Load Balancer Architecture 144
15.3. Load-Balancing Policies 144
15.3.1. Client-side interceptor architecture 145
15.3.2. External load balancer architecture 145

16. Clustering Building Blocks 147
16.1. The HAPartition Service 147
16.1.1. DistributedReplicantManager Service 149
16.1.2. DistributedState Service 149
16.1.3. Custom Use of HAPartition 150

16.2. Distributed Caching with JBoss Cache 150
16.2.1. The JBoss Enterprise Application Platform CacheManager Service 150

17. Clustered JNDI Services **155**
17.1. How it works 155
17.2. Client configuration 157
17.2.1. For clients running inside the application server 157
17.2.2. For clients running outside the application server 159
17.3. JBoss configuration 161
17.3.1. Adding a Second HA-JNDI Service 164

18. Clustered Session EJBs **167**
18.1. Stateless Session Bean in EJB 3.0 167
18.2. Stateful Session Beans in EJB 3.0 168
18.2.1. The EJB application configuration 168
18.2.2. Optimize state replication 170
18.2.3. CacheManager service configuration 170
18.3. Stateless Session Bean in EJB 2.x 173
18.4. Stateful Session Bean in EJB 2.x 174
18.4.1. The EJB application configuration 174
18.4.2. Optimize state replication 175
18.4.3. The HASessionState service configuration 175
18.4.4. Handling Cluster Restart 176
18.4.5. JNDI Lookup Process 177
18.4.6. SingleRetryInterceptor 177

19. Clustered Entity EJBs **179**
19.1. Entity Bean in EJB 3.0 179
19.1.1. Configure the distributed cache 179
19.1.2. Configure the entity beans for cache 182
19.1.3. Query result caching 184
19.2. Entity Bean in EJB 2.x 186

20. HTTP Services **189**
20.1. Configuring load balancing using Apache and mod_jk 189
20.1.1. Download the software 189
20.1.2. Configure Apache to load mod_jk 190
20.1.3. Configure worker nodes in mod_jk 191
20.1.4. Configuring JBoss to work with mod_jk 193
20.2. Configuring HTTP session state replication 194
20.2.1. Enabling session replication in your application 195
20.2.2. Using FIELD level replication 197
20.3. Monitoring session replication 200
20.4. Using Clustered Single Sign On 200
20.4.1. Configuration 201
20.4.2. SSO Behavior 202
20.4.3. Limitations 202
20.4.4. Configuring the Cookie Domain 202

21. JBoss Messaging Clustering Notes **205**
21.1. Unique server peer id 205
21.2. Clustered destinations 205
21.3. Clustered durable subs 205
21.4. Clustered temporary destinations 205

21.5. Non clustered servers 205
21.6. Message ordering in the cluster 205
21.7. Idempotent operations 206
21.7.1. Clustered connection factories 206

22. Clustered Deployment Options **207**
22.1. Clustered Singleton Services 207
22.1.1. HASingleton Deployment Options 207
22.1.2. Determining the master node 211
22.2. Farming Deployment 212

23. JGroups Services **213**
23.1. Configuring a JGroups Channel's Protocol Stack 213
23.1.1. Common Configuration Properties 214
23.1.2. Transport Protocols 215
23.1.3. Discovery Protocols 218
23.1.4. Failure Detection Protocols 221
23.1.5. Reliable Delivery Protocols 224
23.1.6. Group Membership (GMS) 225
23.1.7. Flow Control (FC) 225
23.1.8. Fragmentation (FRAG2) 227
23.1.9. State Transfer 228
23.1.10. Distributed Garbage Collection (STABLE) 228
23.1.11. Merging (MERGE2) 228
23.2. Other Configuration Issues 229
23.2.1. Binding JGroups Channels to a particular interface 229
23.2.2. Isolating JGroups Channels 230
23.3. JGroups Troubleshooting 232
23.3.1. Nodes do not form a cluster 232
23.3.2. Causes of missing heartbeats in FD 232

24. JBoss Cache Configuration and Deployment **235**
24.1. Key JBoss Cache Configuration Options 235
24.1.1. Editing the CacheManager Configuration 235
24.1.2. Cache Mode 240
24.1.3. Transaction Handling 241
24.1.4. Concurrent Access 242
24.1.5. JGroups Integration 243
24.1.6. Eviction 244
24.1.7. Cache Loaders 244
24.1.8. Buddy Replication 246
24.2. Deploying Your Own JBoss Cache Instance 247
24.2.1. Deployment Via the CacheManager Service 247
24.2.2. Deployment Via a **`-service.xml`** File 250
24.2.3. Deployment Via a **`-jboss-beans.xml`** File 251

IV. Performance Tuning **255**

25. JBoss Enterprise Application Platform 5 Performance Tuning **257**
25.1. Introduction 257
25.2. Hardware tuning 257
25.2.1. CPU (Central Processing Unit) 257
25.2.2. RAM (Random Access Memory) 258

25.2.3. Hard Disk 258
25.3. Operating System Performance Tuning 258
25.3.1. Networking 259
25.4. Tuning the JVM 259
25.5. Tuning your applications 259
25.5.1. Instrumentation 260
25.6. Tuning JBoss Application Server 260
25.6.1. Memory usage 260
25.6.2. Database Connection 261
25.6.3. Other key configurations 265

V. Index **267**

Index **269**

What this Book Covers

The primary focus of this book is the presentation of the standard JBoss Enterprise Application 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.

Introduction

JBoss Enterprise Application Platform 5 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).
The JBoss Microcontainer project is standalone and replaces the JBoss JMX Microkernel used in the 4.x JBoss Enterprise Application Platforms.

The JBoss Microcontainer integrates nicely with the JBoss Aspect Oriented Programming framework (JBoss AOP). JBoss AOP is discussed in *Chapter 7, JBoss AOP* Support for JMX in JBoss Enterprise Application Platform 5 remains strong and MBean services written against the old Microkernel are expected to work.

JBoss Enterprise Application Platform 5 is designed around the advanced concept of a Virtual Deployment Framework (VDF). The JBoss Enterprise Application Platform 5 Virtual Deployment Framework (VDF) takes the aspect oriented design of many of the earlier JBoss containers and applies it to the deployment layer. It is also based on the POJO microntainer rather than JMX as in previous releases. More information about the Virtual Deployment Framework (VDF) can be found in *Chapter 6, JBoss5 Virtual Deployment Framework*.

A sample Java EE 5 application that can be run on top of JBoss Enterprise Application Platform 5.0.0.GA and above which demonstrates many interesting technologies is the Seam Booking Application available with this distribution. This example application makes use of the following technologies running on JBoss Enterprise Application Platform 5:

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

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

- JBoss EJB3 included with JBoss Enterprise Application 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 Messaging is a high performance JMS provider in the JBoss Enterprise Middleware Stack (JEMS), included with JBoss Enterprise Application Platform 5 as the default messaging provider. It is also the backbone of the JBoss ESB infrastructure. JBoss Messaging is a complete rewrite of JBossMQ, which is the default JMS provider for the JBoss Enterprise Application Platform 4.x series.

- JBossCache 2.0 that 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.
- JBossWS 2 is the web services stack for JBoss Enterprise Application Platform 5 providing Java EE compatible web services, JAXWS-2.0.
- JBoss Transactions is the default transaction manager for JBoss Enterprise Application 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 Application 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.

JBoss Enterprise Application Platform 5 includes numerous features and bug fixes, many of them carried over from the JBoss Enterprise Application Platform 4.x codebase. See the Detailed Release Notes section for the full details.

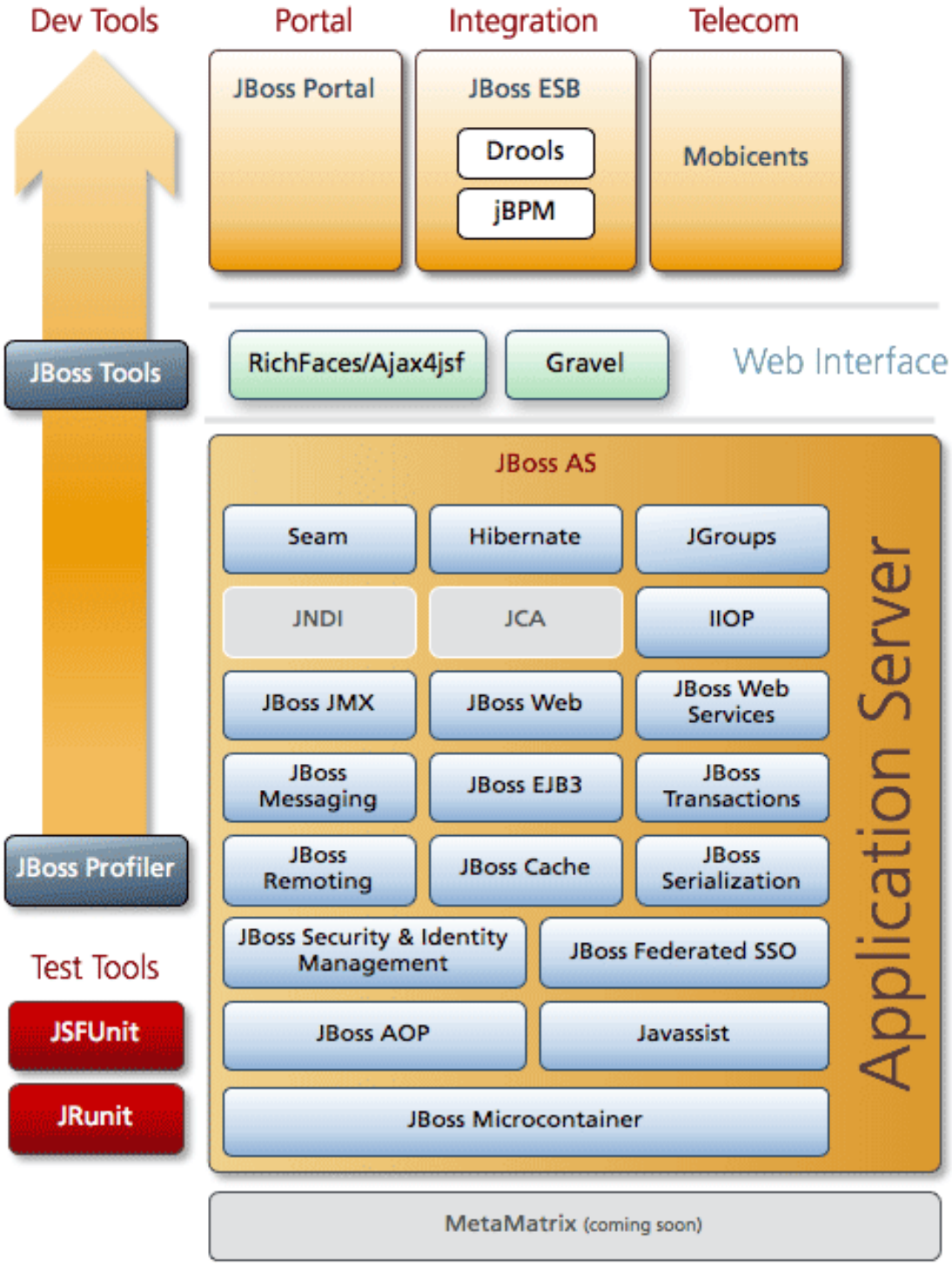
1.1. JBoss Enterprise Application Platform Use Cases

- 99% of web apps involve a database
- Mission critical web applications likely to be clustered.
- Simple web applications with JSPs/Servlets upgrades to JBoss Enterprise Application 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), Java Messaging (JMS), caching etc.
- Cross application middleware (JMS, Corba, JMX etc).

Part I. JBoss Enterprise Application Platform Infrastructure

JBoss Enterprise Application Platform 5 architecture

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



The directory structure of JBoss Enterprise Application Platform 5 resembles that of the 4.x series with some notable differences:

```
-<JBOSS_HOME>/ - the path to your JBoss Enterprise Application Platform
 installation.
  |-- bin - contains start scripts and run.jar
  |-- client - client jars
  |-- common - static jars shared across server configuration
  |   |-- lib
  |   |   |-- antlr.jar
  |   |   |-- ... many more jars
  |-- docs - docs, schemas/dtds, examples
  |   |-- dtd
  |   |-- examples
  |   |   |-- binding-manager
  |   |   |   `-- sample-bindings.xml
  |   |   |-- jca
  |   |   |-- jms
  |   |   |-- jmx
  |   |   |-- netboot
  |   |   |   `-- netboot.war
  |   |   `-- varia
  |   |       |-- deployment-service
  |   |       |-- derby-plugin.jar
  |   |       |-- entity-resolver-manager
  |   |       |   `-- xmlresolver-service.xml
  |   |       `-- jboss-bindings.xml
  |   `-- schema
  |-- lib - core bootstrap jars.
  |   |-- concurrent.jar
  |   |-- dom4j.jar
  |   |-- getopt.jar
  |   |-- javassist.jar
  |   |-- ...
  |   |-- endorsed - added to the server JVM java.endorsed.dirs path
  |   |   |-- serializer.jar
  |   |   |-- xalan.jar
  |   |   `-- xercesImpl.jar
  `-- server - contains the same server configuration/profile directories.
  |-- default
  |   |-- conf - contains server configuration files used when starting the
 server.
  |   |   |-- bootstrap/
  |   |   |   |-- aop.xml - JBoss AOP integration and AspectManager beans
  |   |   |   |-- bindings.xml - Rewrite of the ServiceBindingManager as a
 POJO bean
  |   |   |   |-- classloader.xml - the root class loading beans for the
 peer class loading model
  |   |   |   |-- deployers.xml - Core deployers for -jboss-beans.xml and -
service.xml
```

```
 |   |   |   |-- jmx.xml - JBoss JMX kernel initialization
 |   |   |   |-- profile-repository.xml - full featured repository based
profile service referenced by bootstrap.xml
 |   |   |   |-- profile.xml - simple disk based profile service
referenced by bootstrap-norepo.xml
 |   |   |   |-- vfs.xml - JBoss VFS caching beans
 |   |   |-- bootstrap.xml - bootstrap deployment definition file
 |   |   |-- bootstrap-norepo.xml - bootstrap deployment definition file
for the non-repository profile service
 |   |   |-- java.policy - stub for java security policy
 |   |   |-- jax-ws-catalog.xml - oasis catalog driven schema/dtd
namespace configuration
 |   |   |-- jboss-log4j.xml - the server log4j configuration
 |   |   |-- jboss-service.xml - legacy static mbeans, to be removed in
future
 |   |   |-- jbossjta-properties.xml - JBossTS properties
 |   |   |-- jndi.properties - server default JNDI properties
 |   |   |-- login-config.xml - security authentication domain definitions
 |   |   |-- props -
 |   |   |   |-- jbossws-roles.properties
 |   |   |   |-- jbossws-users.properties
 |   |   |   |-- jmx-console-roles.properties
 |   |   |   `-- jmx-console-users.properties
 |   |   |-- standardjboss.xml - legacy EJB2 container definitions
 |   |   |-- standardjbosscmp-jdbc.xml - legacy CMP2 definitions
 |   |   `-- xmdesc - JBoss XMBean descriptors
 |   |       |-- AttributePersistenceService-xmbean.xml
 |   |       |-- ClientUserTransaction-xmbean.xml
 |   |       |-- JNDIView-xmbean.xml
 |   |       |-- Log4jService-xmbean.xml
 |   |       |-- NamingBean-xmbean.xml
 |   |       |-- NamingService-xmbean.xml
 |   |       |-- TransactionManagerService-xmbean.xml
 |   |       |-- org.jboss.deployment.JARDeployer-xmbean.xml
 |   |       |-- org.jboss.deployment.MainDeployer-xmbean.xml
 |   |       `-- org.jboss.deployment.SARDeployer-xmbean.xml
 |   |-- data - location for services data
 |   |   |-- hypersonic
 |   |   |-- jboss.identity
 |   |   |-- tx-object-store
 |   |   `-- xmbean-attrs
 |   |-- deploy - this is where services and your java applications are
deployed.
    You can deploy an application on the JBoss application server by simply
    copying the application's (WAR, EAR or JAR files) into this directory.
 |   |-- deployers/ - new vdf deployers
 |   |   |-- alias-deployers-jboss-beans.xml - Deployers that know how to
handle The know
how to handle deployment aliases.
 |   |   |-- bsh-deployer - beanshell deployer
```

```
  |   |   |-- clustering-deployer-jboss-beans.xml - add dependencies on
 needed clustering services
to clustered EJB3, EJB2 beans and to distributable web applications.
  |   |   |-- dependency-deployers-jboss-beans.xml - aliases.txt, jboss-
dependency.xml
  |   |   |-- directory-deployer-jboss-beans.xml
  |   |   |-- ear-deployer-jboss-beans.xml - ear deployers
  |   |   |-- ejb-deployer-jboss-beans.xml - ejb2.x deployers
  |   |   |-- ejb3.deployer - ejb3 deployers
  |   |   |-- hibernate-deployer-jboss-beans.xml - deployers for -
hibernate.xml descriptors
  |   |   |-- jboss-aop-jboss5.deployer - aspect deployer
  |   |   |-- jboss-jca.deployer - JCA deployers
  |   |   |-- jbossweb.deployer - war deployers
  |   |   |-- jbossws.deployer - web services deployers
  |   |   |-- jsr77-deployers-jboss-beans.xml - JSR77 mbean view creation
 deployers
  |   |   |-- metadata-deployer-jboss-beans.xml - metadata handlers
  |   |   |-- seam.deployer - seam integration deployer
  |   |   |-- security-deployer-jboss-beans.xml - security deployers
  |   |-- lib - default server static jars, empty by default
  |   |-- log - default root for log files, controlled by
 jboss.server.log.dir
  |   |   |-- boot.log
  |   |   |-- server.log
  |   |-- tmp
  |   `-- work
  |       `-- jboss.web
  |           `-- localhost
  `-- minimal - a minimal server configuration
  |   |-- conf - contains server configuration files used when starting the
 server.
  |   |-- bootstrap/
  |   |   |   |-- aop.xml
  |   |   |   |-- classloader.xml
  |   |   |   |-- deployers.xml
  |   |   |   |-- jmx.xml
  |   |   |   |-- profile.xml
  |   |-- bootstrap.xml
  |   |-- jboss-log4j.xml
  |   |-- jboss-service.xml
  |   |-- jndi.properties
  |   |-- xmdesc
  |       |-- NamingBean-xmbean.xml
  |       `-- NamingService-xmbean.xml
  | |-- deploy/
  | | |-- hdscanner-jboss-beans.xml
  | |-- deployers/
  | |-- lib
  | | |-- jboss-minimal.jar
  | | |-- jnpserver.jar
```

```
  | | |-- log4j.jar
```

2.1. The JBoss Enterprise Application Platform Bootstrap

The JBoss Enterprise Application Platform 5 bootstrap is similar to the JBoss Enterprise Application Platform 4.x versions in that the org.jboss.Main entry point loads an org.jboss.system.server.Server implementation. In JBoss Enterprise Application Platform 4.x this was a JMX based microkernel. In JBoss Enterprise Application Platform 5 this is a JBoss Microcontainer.

The default JBoss Enterprise Application 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-name}* is the name of the profile being loaded and corresponds to the `server -c` command line argument. The default *{profile-name}* is *default*. The deployers, deploy

2.2. Hot Deployment

Hot deployment in JBoss Enterprise Application 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 deployment.

Part II. JBoss Enterprise Application Platform 5 Configuration

Deployment

Deploying applications on JBoss Enterprise Application Platform is achieved by copy the application into the **`JBOSS_HOME/server/default/deploy`** directory. You can replace *default* with different server profiles such as *all* or *minimal*. We will cover those later in this chapter. The JBoss Enterprise Application 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 Application Platform is still running.

3.1. Deployable Application Types

With JBoss Enterprise Application Platform 4.x, a deployer existed to handle a specified deployment type and that was the only deployer that would process the deployment. In JBoss Enterprise Application Platform 5, multiple deployers transform the metadata associated with a deployment until its processed by a deployer that creates a runtime component from the metadata. Deployment has to contain a descriptor that causes the component metadata to be added to the deployment. The types of deployments for which deployers exists by default in the JBoss Enterprise Application Platform include:

- The WAR application archive (e.g., myapp.war) packages a Java EE web application in a JAR file. It contains servlet classes, view pages, libraries, and deployment descriptors in WEB-INF such as web.xml, faces-config.xml, and jboss-web.xml etc..
- The EAR application archive (e.g., myapp.ear) packages a Java EE enterprise application in a JAR file. It typically contains a WAR file for the web module, JAR files for EJB modules, as well as META-INF deployment descriptors such as application.xml and jboss-app.xml etc.
- 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 Application Platform component deployers.
- The SAR application archive (e.g., myservice.sar) packages a JBoss service in a JAR file. It is mostly used by JBoss Enterprise Application Platform internal services that have not been updated to support MC beans style deployments.
- The **`*-ds.xml`** file defines connections to external databases. The data source can then be reused by all applications and services in JBoss Enterprise Application Platform via the internal JNDI.
- You can deploy **`*-jboss-beans.xml`** files with MC beans definitions. If you have the approriate JAR files available in the deploy or lib directories, the MC beans can be deployed using such a standalone XML file. This is a
- 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 Application Platform internal services that have not been updated to support POJO style deployment, such as the JMS queues.
- You can also deploy JAR files containing EJBs or other service objects directly in JBoss Enterprise Application Platform. The list of suffixes that are recognized as JAR files is specified in the **`conf/bootstrap/deployers.xml`** JARStructure bean constructor set.

Exploded Deployment

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

3.2. Standard Server Profiles

The JBoss Enterprise Application Platform ships with five 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 all` command would start the server in the *all* profile. Each profile is contained in a directory named **`JBOSS_HOME/server/[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 name]`** directory depends on the profile service implementation and is subject to change as the management layer and embedded server evolve.

- The minimal profile starts the core server container without any of the enterprise services. It is a good starting point if you want to build a customized version of JBoss Enterprise Application Platform that only contains the services you need.
- The *default* profile is the mostly common used profile for application developers. It supports the standard Java EE 5.0 programming APIs (e.g., Annotations, JPA, and EJB3).
- The *standard* profile is the profile that has been tested for JavaEE compliance. The major differences with the existing configurations is that call-by-value and deployment isolation are enabled by default, along with support for **`rmiiiop`** and **`juddi`** (taken from the *all* config).
- The *all* profile is the default profile with clustering support and other enterprise extensions.
- The *web* profile is a new experimental lightweight configuration created around JBoss Web that will follow the developments of the JavaEE 6 web profile. Except for the **`servlet/jsp`** container it provides support for JTA/JCA and JPA. It also limits itself to allowing access to the server only through the http port. Please note that this configuration is not JavaEE certified and will most likely change in the following releases.

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

Microcontainer

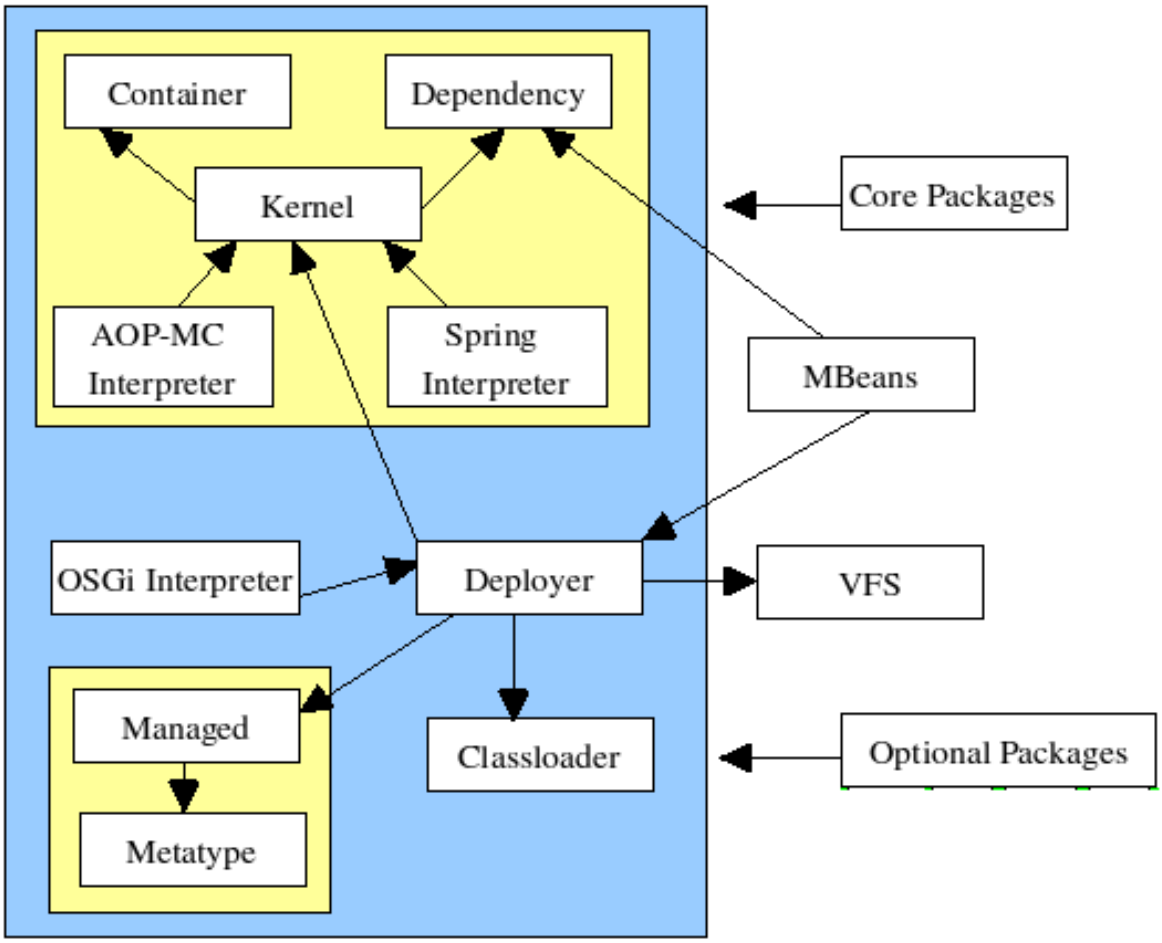
JBoss Enterprise Application Platform 5.0 uses the Microcontainer to integrate enterprise services together with a Servlet/JSP container, EJB container, deployers and management utilities in order to provide a standard Java EE environment. If you need additional services then you can simply deploy these 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. You can even use the Microcontainer to do this in other environments such as Tomcat and GlassFish by plugging in different classloading models during the service deployment phase.

Since JBoss Microcontainer is very lightweight and deals with POJOs, it can also be used to deploy services into a Java ME runtime environment. This opens up new possibilities for mobile applications that can now take advantage of enterprise services without requiring a full JEE application server.

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.

4.1. An overview of the Microcontainer modules

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



- **`aop-mc-int`** handles integration between the JBossAOP and Microcontainer projects
- **`classloader`** new peer classloader model, prepared to handle OSGi bundle model.
- The **`container`** module contains: reflection, the integration point for manipulating class information at runtime (for example, overriding annotations or obtaining an aop instance advisor), joinpoint (the joinpoint model including the join point factory), classadaptor (the integration and configuration spi) and metadata (the base metadata types and repository).
- **`dependency`** management is handled by the controller. The controller is the core component for keeping track of contexts to make sure the configuration and lifecycle are done in the correct order including dependencies and classloading considerations.
- **`deployers`** load components from various models, POJOs, JMX, spring, Java EE, etc. into the Microcontainer runtime.
- **`kernel`** kernel defines the core kernel spi including, boostrap, configuration, POJO deployments, dependency, events, bean metadata, and bean registry.
- The **`managed`** module defines the base objects defining the management view of a component.

- The **metatype** module defines the base types found in the management view of a component.
- **guice-int** contains the integration classes for guice.
- **osgi-int** contains the integration classes that adapt the OSGi model onto the Microcontainer.
- **reliance-identity** defines identity as a MC POJO service
- **reliance-rules** defines your dependencies with Drools
- **reliance-jbpm** defines your dependencies with jBPM
- **spring-int** contains the integration classes that adapt the spring model onto the Microcontainer.

4.2. Configuration

To configure the Microcontainer bootstrap you can use the **JBOSS_HOME/server/<server_configuration>/conf/bootstrap.xml** and **JBOSS_HOME/server/<server_configuration>/conf/bootstrap/*.xml** files where *<server_configuration>* represents the name of the server profile, for example, *all*, *default* or *minimal*. The **bootstrap.xml** simply references Microcontainer deployment descriptors that should be loaded in the indicated order. The current *default* profile **bootstrap.xml** references are:

- **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
- **bindings.xml** - Rewrite of the ServiceBindingManager as a POJO bean
- **profile-repository.xml** - full featured repository based profile service referenced by **bootstrap.xml**

The main beans are:

- *ProfileService* : This bean loads the deployments associated with the named server profile, *default*, *all* or the name that is passed to the server using the `-c` option. It's an extension of always looking to the filesystem **server/name/conf/jboss-service.xml** and **server/name/deploy** to load deployments.
- *AspectManager* : the AOP aspects
- *MainDeployer* : An update of the JMX based MainDeployer from earlier versions to a one based on the Microcontainer, **JBoss5VirtualFileSystem**, and Virtual Deployment Framework(VDF). Deployer aspects are registered with the MainDeployer as an ordered list via inject of the deployers property.
- *ServiceClassLoaderDeployer* : Manages the class loading aspect of deployment.
- *JARDeployer* : This bean is a structural deployment aspect which handles the legacy nested deployment behavior of adding non-deployable jars to the current deployment classpath.

- *FileStructure* : this bean is a structural deployment aspect which recognizes well know deployment file types specified by suffix.
- *AspectDeployer* : handles aop descriptor deployments.
- *BeanDeployer* : this bean translates **`deployer-beans.xml`** into **`KernelDeployment`** for the descriptor beans.
- *KernelDeploymentDeployer* : Translates a **`KernelDeployment`** into the constituent **`BeanMetaData`** instances for the kernel beans.
- *BeanMetaDataDeployer* : Creates the kernel beans from the deployment **`BeanMetaData`**.
- *SARDeployer* : this bean is a port of the legacy JMX **`SARDeployer`** to the VDF. It handles the legacy **`jboss-service.xml`** style of mbean deployment descriptors and maps this into a **`ServiceDeployment`** POJO.
- *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 the instantiation of a JMX kernel and **`MBeanServer`** in the jboss domain. It is used by the **`SARDeployer`**. It will be used by other management deployment aspects in the future to expose kernel beans via JMX.
- *VFSDeployerScanner* : A scanner bean that loads the deployers directory contents into the basic profile service.
- *VFSDeploymentScanner* : A scanner bean that loads the deploy directory contents into the basic profile service.
- *HDScanner* : A bean that queries the profile service for changes in deploy directory contents and redeploys updated content, undeploys removed content, and add new deployment content to the profile service.

4.3. References

More information on the JBoss Microcontainer project can be obtained from *http://labs.jboss.com/jbossmc/*.

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 Application 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.

5.1. 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 elimenates the need to rewrite whole functionalities because they were developed by another enterprise department using a different software language.

5.2. 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.

5.3. Jboss Web Services Attachment support with XOP (XML-binary Optimized Packaging) and SwA

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.

WS-I Attachment Profile 1.0 defines a mechanism to reference MIME attachment parts using swaRef. In this mechanism the content of XML element of type wsi:swaRef is sent as a MIME attachment and the element inside SOAP Body holds the reference to this attachment in the CID URI scheme as defined by RFC 2111.

5.4. 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>
```

5.5. MTOM/XOP

This chapter 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) *http://www.w3.org/TR/soap12-mtom/*)
- XML-binary Optimized Packaging (XOP) (*http://www.w3.org/TR/xop10/*)

image/jpeg	java.awt.Image
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
application/octet-stream	javax.activation.DataHandler

Table 5.1. Supported MTOM parameter types

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

5.6. 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 {

 [...]
 }
```

5.6.1. 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);
```

5.7. 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>
```

5.8. 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;

   ...
}
```

5.9. 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.

5.10. 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.

5.11. RPC/Encoded

SOAP encodeding style is defined by the infamous *chapter 5*[1] of the *SOAP-1.1*[2] specification. It has inherent interoperability issues that cannot be fixed. The *Basic Profile-1.0*[3] prohibits this encoding style in *4.1.7 SOAP encodingStyle Attribute*[4]. 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

5.12. 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.

5.13. Plain old Java Object (POJO)

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

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

5.14. The endpoint as a web application

A JAX-WS java service endpoint (JSE) is deployed as a web application.

```
<web-app ...>
```

[1] http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383512
[2] http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
[3] http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html
[4] http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html#refinement16448072

```
 <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>
```

5.15. 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.

5.16. 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 (Java to WSDL)*[5]

5.17. 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)*[6] endpoints. EJB-2.1 endpoints are supported using the JAX-RPC progamming model.

```

  @Stateless
  @Remote(EJB3RemoteInterface.class)
```

[5] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Top_Down_.28Java_to_WSDL.29
[6] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#_Plain_old_Java_Object_.28POJO.29

```
@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/jbossws/services
```

It is also possible to generate the abstract contract offline using JbossWS tools. For details of that please see *#Top Down (Java to WSDL)*[7]

5.18. 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.

[7] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Top_Down_.28Java_to_WSDL.29

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)

5.19. 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);
```

```
}
}
```

5.20. Web Service Clients

5.20.1. 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).

5.20.1.1. Service Usage

Static case

Most clients will start with a WSDL file, and generate some stubs using jbossws tools like *wsconsume*. This usually gives a mass of files, one of which is the top of the tree. This is the service implementation class.

The generated implementation class can be recognised as it will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a java.net.URL) and the service name (a javax.xml.namespace.QName) respectively.

Usually you will use the no-argument constructor. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the WebServiceClient annotation that decorates the generated class.

The following code snippet shows the generated constructors from the generated class:

```

// Generated Service Class

 @WebServiceClient(name="StockQuoteService", targetNamespace="http://
example.com/stocks", wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
public StockQuoteService()
{
super(new URL("http://example.com/stocks.wsdl"), new QName("http://
example.com/stocks", "StockQuoteService"));
}

public StockQuoteService(String wsdlLocation, QName serviceName)
{
super(wsdlLocation, serviceName);
```

```
}

...
}
```

Section *#Dynamic Proxy*[8] 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*[9].

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.

5.20.1.2. 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*[10] 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.

5.20.1.3. 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.

5.20.2. Dynamic Proxy

You can create an instance of a client proxy using one of `getPort` methods on the ***`Service`***[11].

```
/**
```

[8] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Dynamic_Proxy
[9] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Dispatch
[10] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Handler_Framework
[11] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#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 (WSDL to Java)*[12].

A generated static ***Service***[13] 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()
{
```

[12] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#_Top_Down_.28WSDL_to_Java.29
[13] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Service

```
return (TestEndpoint)super.getPort(TESTENDPOINTPORT, TestEndpoint.class);
}
 }
```

5.20.3. 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 theWSDL location information specified in the **`WebService`** annotation of the referenced generated service class.

```
public class EJB3Client implements EJB3Remote
{
 @WebServiceRef
 public TestEndpointService service4;

 @WebServiceRef
 public TestEndpoint port3;
```

WebServiceRef Customization

In Jboss Enterprise Application Platform 5.0 we offer a number of overrides and extensions to the **`WebServiceRef`** annotation. These include

- define the port that should be used to resolve a container-managed port
- define default Stub property settings for Stub objects
- define the URL of a final WSDL document to be used

Example:

```
<service-ref>
<service-ref-name>OrganizationService</service-ref-name>
<wsdl-override>file:/wsdlRepository/organization-service.wsdl</wsdl-
override>
</service-ref>
```

```
..
<service-ref>
<service-ref-name>OrganizationService</service-ref-name>
<config-name>Secure Client Config</config-name>
<config-file>META-INF/jbossws-client-config.xml</config-file>
<handler-chain>META-INF/jbossws-client-handlers.xml</handler-chain>
</service-ref>

<service-ref>
<service-ref-name>SecureService</service-ref-name>
<service-class-
name>org.jboss.tests.ws.jaxws.webserviceref.SecureEndpointService</service-
class-name>
<service-qname>{http://org.jboss.ws/wsref}SecureEndpointService</service-
qname>
<port-info>
<service-endpoint-
interface>org.jboss.tests.ws.jaxws.webserviceref.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>
```

5.20.4. Dispatch

XMLWeb Services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The Dispatch interface provides support for this mode of interaction.

Dispatch supports two usage modes, identified by the constants javax.xml.ws.Service.Mode.MESSAGE and javax.xml.ws.Service.Mode.PAYLOAD respectively:

Message

In this mode, client applications work directly with protocol-specific message structures. 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)));
```

5.20.5. 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);
}
```

5.20.6. 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;
}
}
```

5.21. Common API

This sections describes concepts that apply equally to *#Web Service Endpoints*[14] and *#Web Service Clients*[15]

5.21.1. 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.

5.21.1.1. 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`**.

[14] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Web_Service_Endpoints
[15] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#Web_Service_Clients

5.21.1.2. 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`**.

5.21.1.3. 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)

5.21.1.4. 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!
```

5.21.2. Message Context

MessageContext is the super interface for all JAX-WS message contexts. It extends Map<String,Object> with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the put method to insert a property in the message context that one or more other handlers in the handler chain may subsequently obtain via the get method.

Properties are scoped as either APPLICATION or HANDLER. All properties are available to all handlers for an instance of an MEP on a particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during

the execution of an MEP instance. APPLICATION scoped properties are also made available to client applications (see section 4.2.1) and service endpoint implementations. The defaultscope for a property is HANDLER.

5.21.2.1. Accessing the message context

There is currently no portable way of doing this in 4.0.5. **`@WebServiceContext`** injection will be available with 4.2. In the meantime you can access the message context like this:

```
CommonMessageContext msgContext =
 MessageContextAssociation.peekMessageContext();
msgContext.setProperty(<Name>, <Value>);
```

5.21.2.2. Logical Message Context

`#Logical Handlers` are passed a message context of type LogicalMessageContext when invoked. LogicalMessageContext extends MessageContext with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of amessage. 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.

5.21.2.3. SOAP Message Context

SOAP handlers are passed a SOAPMessageContext when invoked. SOAPMessageContext extends MessageContext with methods to obtain and modify the SOAP message payload.

5.21.3. Fault Handling

An implementation may thow 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

5.22. DataBinding

5.22.1. Using JAXB with non annotated classes

Since 2.0.2

JAXB is heavily driven by Java Annotations on the Java Bindings. It currently doesn't support an external binding configuration. This recently became an issue for us on JBossESB since the JBossWS 2.0.0 native SOAP stack uses JAXB to perform the SOAP to Java bindings (see 1, 2). It's an issue for JBossESB simply because it needs to be able to support user definition of JBossWS native Webservice Endpoints (e.g. JSR 181) using Java typesets that have not been "JAXB Annotated" (see JAXB Introductions On JBossWS).

In order to support this, we built on a JAXB RI feature whereby it allows you to specify a RuntimeInlineAnnotationReader implementation during JAXBContext creation (see JAXBRIContext).

We call this feature "JAXB Annotation Introduction" and we've made it available for general consumption i.e. it can be checked out, built and used from SVN:

- *http://anonsvn.jboss.org/repos/jbossws/projects/jaxbintros/*

Complete documentation can be found here:

- *JAXB Introductions*[16]

5.23. Attachments

5.23.1. 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)*[17]
- *XML-binary Optimized Packaging (XOP)*[18]

5.23.1.1. 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*[19] 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

5.23.1.2. 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`**):

[19] http://java.sun.com/j2ee/1.4/docs/api/javax/activation/DataHandler.html

```
[...]
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.

5.23.2. SwaRef

Since 2.0

WS-I Attachment Profile 1.0[20] defines mechanism to reference MIME attachment parts using *swaRef*[21]. In this mechanism the content of XML element of type wsi: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*[22].

5.23.2.1. 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
```

[20] http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html
[21] http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html#Referencing_Attachments_from_the_SOAP_Envelope
[22] http://www.ietf.org/rfc/rfc2111.txt

```
@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>
```

5.23.2.2. 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.

5.24. 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 abstact contract (WSDL) that defines your service (top-down development). If this is a new service (no existing contract), the bottom-up approach is the fastest route; you only need to add a few annotations to your classes to get a service up and running. However, if you are developing a service with an already defined contract, it is far simpler to use the top-down approach, since the provided tool will generate the annotated code for you.

Bottom-up use cases:

- Exposing an already existing EJB3 bean as a Web Service
- Providing a new service, and you want the contract to be generated for you

Top-down use cases:

- Replacing the implementation of an existing Web Service, and you can't break compatibility with older clients
- Exposing a service that conforms to a contract specified by a third party (e.g. a vender that calls you back using an already defined protocol).
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front

The following JAX-WS command line tools are included in JBossWS:

Command	Description
wsprovide[23]	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
wsconsume[24]	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[25]	Executes a Java client (has a main method) using the JBossWS classpath.

5.24.1. 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*[26] 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*[27] 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'>
```

[26] http://jbws.dyndns.org/mediawiki/index.php?title=Wsprovide
[27] http://jbws.dyndns.org/mediawiki/index.php?title=Wsprovide

```
<operation name='echo' parameterOrder='echo'>
<input message='tns:Echo_echo'/>
<output message='tns:Echo_echoResponse'/>
</operation>
</portType>
```

Note

Remember that **when deploying on JBossWS you do not need to run this tool.** You only need it for generating portable artifacts and/or the abstract contract for your service.

Let us create a POJO endpoint for deployment on JBoss Enterprise Application Platform. A simple **`web.xml`** needs to be created:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/
xml/ns/j2ee/web-app_2_4.xsd"
version="2.4">

<servlet>
<servlet-name>Echo</servlet-name>
<servlet-class>echo.Echo</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>Echo</servlet-name>
<url-pattern>/Echo</url-pattern>
</servlet-mapping>
</web-app>
```

The **`web.xml`** and the single class can now be used to create a WAR:

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

The war can then be deployed:

```
```

```
cp echo.war /usr/local/jboss-4.2.0.GA-ejb3/server/default/deploy
```

This will internally invoke *wsprovide*[28], 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.

5.24.2. 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*[29] 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*[30] 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

[28] http://jbws.dyndns.org/mediawiki/index.php?title=Wsprovide
[29] http://jbws.dyndns.org/mediawiki/index.php?title=Wsconsume
[30] http://jbws.dyndns.org/mediawiki/index.php?title=Wsconsume

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;
}
}
```

5.24.3. 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*[31]. 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*[32]:

```
$ 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
```

[31] http://jbws.dyndns.org/mediawiki/index.php?title=Wsprovide
[32] http://jbws.dyndns.org/mediawiki/index.php?title=Wsconsume

```
{
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*[33] 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]));
...
```

5.24.4. Command-line & Ant Task Reference

- *wsconsume reference page*[34]
- *wsprovide reference page*[35]
- *wsrunclient reference page*[36]

5.24.5. 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:

[33] http://jbws.dyndns.org/mediawiki/index.php?title=Wsrunclient

- *binding customization*[37]

5.25. Web Service Extensions

5.25.1. WS-Addressing

This section describes how *WS-Addressing*[38] can be used to provide a staful service endpoint.

5.25.1.1. Specifications

WS-Addressing is defined by a combination of the following specifications from the W3C Candidate Recommendation 17 August 2005. The WS-Addressing API is standardized by *JSR-261 - Java API for XML Web Services Addressing*[39]

- *Web Services Addressing 1.0 - Core*[40]
- *Web Services Addressing 1.0 - SOAP Binding*[41]

5.25.1.2. Addressing Endpoint

The following endpoint implementation has a set of operation for a typical stateful shopping chart application.

```
@WebService(name = "StatefulEndpoint", targetNamespace = "http://
org.jboss.ws/samples/wsaddressing", serviceName = "TestService")
@EndpointConfig(configName = "Standard WSAddressing Endpoint")
@HandlerChain(file = "WEB-INF/jaxws-handlers.xml")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class StatefulEndpointImpl implements StatefulEndpoint,
 ServiceLifecycle
{
@WebMethod
public void addItem(String item)
{ ... }

@WebMethod
public void checkout()
{ ... }

@WebMethod
public String getItems()
{ ... }
}
```

It uses the *JAX-WS Endpoint Configuration# Standard WSAddressing Endpoint*[42] to enable the server side addressing handler. It processes the incomming WS-Addressing header elements and provides access to them through the JSR-261 API.

[38] http://www.w3.org/TR/ws-addr-core
[39] http://www.jcp.org/en/jsr/detail?id=261
[42] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Endpoint_Configuration#_Standard_WSAddressing_Endpoint

The endpoint handler chain

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 javaee_web_services_1_2.xsd">

<handler-chain>
<protocol-bindings>##SOAP11_HTTP</protocol-bindings>
<handler>
<handler-name>Application Server Handler</handler-name>
<handler-class>org.jboss.test.ws.jaxws.samples.wsaddressing.ServerHandler</
handler-class>
</handler>
</handler-chain>

</handler-chains>
```

Defines an application specific hander that assignes and processes stateful client IDs.

5.25.1.3. Addressing Client

On the client side there are simmilar handlers that does the reverse. It uses the JSR-261 API to add WS-Addressing header elements including the clientid association.

The client sets a custom handler chain in the binding

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

List<Handler> customHandlerChain = new ArrayList<Handler>();
customHandlerChain.add(new ClientHandler());
customHandlerChain.add(new WSAddressingClientHandler());
bindingProvider.getBinding().setHandlerChain(customHandlerChain);
```

The WSAddressingClientHandler is provided by JBossWS and reads/writes the addressing properties and puts then into the message context.

A client connecting to the stateful endpoint

```
public class AddressingStatefulTestCase extends JBossWSTest
{
public void testAddItem() throws Exception
{
port1.addItem("Ice Cream");
port1.addItem("Ferrari");

port2.addItem("Mars Bar");
```

```
port2.addItem("Porsche");
}

public void testGetItems() throws Exception
{
String items1 = port1.getItems();
assertEquals("[Ice Cream, Ferrari]", items1);

String items2 = port2.getItems();
assertEquals("[Mars Bar, Porsche]", items2);
}
}
```

SOAP message exchange

Below you see the SOAP messages that are beeing 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>
```

5.25.2. WS-BPEL

WS-BPEL is not supported with JAX-WS, please refer to *JAX-RPC User Guide#WS-BPEL*[43].

5.25.3. WS-Eventing

WS-Eventing specifies a set of operations that allow an event consumer to register (subscribe) with an event producer (source) to receive events (notifications) in an asynchronous fashion.

5.25.3.1. Specifications

WS-Eventing is defined by the combination of the following specifications:

- *WS-Eventing specification*[44]
- *WS-Addressing Specifications*[45]

The following section will introduce the main eventing actors and their responsiblities.

Note

The original eventing specification builds upon WS-Addressing 2004/08. JBossWS however decided to stick to the latest version, which is the W3C candidate release.

[43] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-RPC_User_Guide#WS-BPEL

5.25.3.2. Collaboration

1. An event sink (web service client) sends a subscribtion request to the event source endpoint. This includes the event sink endpoint address where notifications should delivered. Upon successful subscription the sink receives a leased subscription ID that can be used to identify the client in subsequent requests.

2. A successfully registered event sink directs management requests (Renew, GetStatus, Unsubscribe) to the subscription manager endpoint using the previously received subscription ID. The subscription manager endpoint address was returned as part of the subscription response in the first place.

3. The actual event sink (application) emits notification messages through the JBossWS-Eventing module. JBossWS-Eventing dispatches the notification to any subscriber endpoint that is registered with a particular event source.s

4. Besides notifications JBossWS-Eventing may emit lifecycle events at any time. For instance, to inform an event sink that a subscription was canceled. This can be the case when the subscription expired or the event source was undeployed.

It is the users responsibilty to supply the web service endpoints (EventSourceEndpoint, SubscriptionManagerEndpoint) that are required for a complete event source deployment. Fortunatly JBossWS-Eventing already ships with a implementation that can be used right away. All that is left to do is the packaging of standard JSR-109 deployment archive that includes the event source specific WSDL and points to the JBossWS-Eventing endpoint implementations.

The relevant steps are:

- Create a custom WSDL that describes your event source, in respect to the notification schema and the fact that is actually contains an event source port

- Use the JBossWS SEI and endpoint implementations (webservices.xml, web.xml).

5.25.3.3. Setup an event source endpoint

With JAX-WS the event source setup has actually become quiet easy. All you need to do is to subclass your endpoint implementation from **`AbstractEventSourceEndpoint`** and a subscription manager from **`AbstractSubscriptionManagerEndpoint`** and finally point that implementation to a *event source specific WSDL*[46].

```
package org.jboss.test.ws.jaxws.samples.wseventing;
..
import javax.jws.WebService;

import org.jboss.logging.Logger;
import org.jboss.ws.annotation.EndpointConfig;
import org.jboss.ws.extensions.eventing.jaxws.AbstractEventSourceEndpoint;

/**
* @author Heiko.Braun@jboss.org
* @version $Id$
```

[46] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide#The_WSDL_that_describes_an_event_source

```
* @since 18.01.2007
*/
@WebService(
    (1)
name = "EventSource",
portName = "EventSourcePort",
targetNamespace = "http://schemas.xmlsoap.org/ws/2004/08/eventing",
wsdlLocation = "/WEB-INF/wsdl/sysmon.wsdl",
 (2)
endpointInterface =
 "org.jboss.ws.extensions.eventing.jaxws.EventSourceEndpoint")
@EndpointConfig(configName = "Standard WSAddressing Endpoint")
    (3)
public class SysmonRegistrationEndpoint extends AbstractEventSourceEndpoint
 {  (4)

private static final Logger log =
 Logger.getLogger(SysmonRegistrationEndpoint.class);

protected Logger getLogger()
{
return log;
}
}
```

1. Of course we need a **`@WebService`** annotation
2. It's important to override the WSDL here
3. You need to tell JBossWS that it requires WS-Addressing for this endpoint
4. Subclass a predefined implementation that knows how to delegate to the actual eventing service implementation

5.25.3.4. The WSDL that describes an event source

Even though we are already using the annotation driven approach, JBossWS eventing still requires an event source specific WSDL.

The following excerpt shows the relevant WSDL details that describe an event source.

```


<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions
targetNamespace="http://www.jboss.org/sysmon"
xmlns:tns="http://www.jboss.org/sysmon"
xmlns:wse='http://schemas.xmlsoap.org/ws/2004/08/eventing'
xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
xmlns:wsa10='http://www.w3.org/2005/08/addressing'
xmlns:xs='http://www.w3.org/2001/XMLSchema'
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
```

```
<wsdl:import
(1)        namespace='http://schemas.xmlsoap.org/ws/2004/08/eventing'
location='jbwse.wsdl' />

<wsdl:types>

<xs:schema targetNamespace='http://schemas.xmlsoap.org/ws/2004/08/
eventing'>
(2)      <xs:include schemaLocation='jbwse.xsd'/>
</xs:schema>

(3)   <xs:schema
targetNamespace="http://www.jboss.org/sysmon"
elementFormDefault="qualified"
blockDefault="#all">
<xs:element name="SystemStatus">
<xs:complexType>
<xs:sequence>
<xs:element name="Time " type="xs:dateTime"/>
<xs:element name="HostName" type="xs:string"/>
<xs:element name="HostAddress" type="xs:string"/>
<xs:element name="ActiveThreadCount" type="xs:int"/>
<xs:element name="FreeMemory" type="xs:string"/>
<xs:element name="MaxMemory" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

</wsdl:types>

<wsdl:message name='SystemInfoMsg'>
<wsdl:part name='body' element='tns:SystemStatus'/>
</wsdl:message>

(4) <wsdl:portType name='SystemInfo' wse:EventSource='true'>
<wsdl:operation name='SysmonOp'>
<wsdl:output message='tns:SystemInfoMsg'/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="SystemInfoBinding" type="tns:SystemInfo">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
http"/>
<wsdl:operation name="SysmonOp">
<soap:operation soapAction=""/>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
```

```
</wsdl:binding>

</wsdl:definitions>
```

1. Import the default eventing WSDL, that includes service and port declarations.
2. Include the default eventing Types
3. Specifiy the notitification message schema.
4. Declare a port type, attributed `"wse:EventSource='true'"` that points to your notification message schema.

5.25.3.5. Emitting notifications

JBossWS-Eventing registeres a event dispatcher within local JNDI tree that can be used to emit notifications from applications.

```
java:/EventDispatcher
```

The event dispatcher interface:

```
public interface EventDispatcher
{
void dispatch(URI eventSourceNS, Element payload);
}
```

Example notification

```
(1)   URI eventSourceURI = new URI("http://http://www.jboss.org/sysmon/
SystemInfo");
(2)   Element payload = DOMUtils.parse("SOME XML STRING");
try
{
InitialContext iniCtx = getInitialContext();
(3)      EventDispatcher delegate = (EventDispatcher)
iniCtx.lookup(EventingConstants.DISPATCHER_JNDI_NAME);
(4)      delegate.dispatch(eventSourceURI, payload);
}
catch (Exception e)
{
//
}
```

1. Address your event source correctly (TargetNamespace+PortTypeName)
2. Create your payload

3. Lookup dispatcher from JNDI

4. Dispatch notification.

The SubscriptionManager MBean is the actual core component that drives the JBossWS-Eventing implementation. It can be accessed through the jmx-console.

```
jboss.ws.eventing:service=SubscriptionManager
```

Management operations exist to monitor and maintain active subscritions and deployed event sources. The current implementation is backed by a ThreadPoolExecutor, that asynchronously delivers messages to event sink endpoints. It can be configured through the following attributes:

- corePoolSize - average number of idle threads
- maximumPoolSize - maximum number of threads
- eventKeepAlive - keep alive before an undelivered event message is discarded.

5.25.4. WS-Security

WS-Security addresses message level security. It standardizes authorization, encryption, and digital signature processing of web services. Unlike transport security models, such as SSL, WS-Security applies security directly to the elements of the web service message. This increases the flexibility of your web services, by allowing any message model to be used (point to point, multi-hop relay, etc).

This chapter describes how to use WS-Security to sign and encrypt a simple SOAP message.

Specifications

WS-Security is defined by the combination of the following specifications:

- *SOAP Message Security 1.0*[47]
- *Username Token Profile 1.0*[48]
- *X.509 Token Profile 1.0*[49]
- *W3C XML Encryption*[50]
- *W3C XML Signature*[51]
- *Basic Security Profile 1.0 (Still in Draft)*[52]

5.25.4.1. Endpoint configuration

JBossWS uses handlers to identify ws-security encoded requests and invoke the security components to sign and encrypt messages. In order to enable security processing, the client and server side need to include a corressponding handler configuration. The preferred way is to reference a predefined *JAX-WS Endpoint Configuration*[53] or *JAX-WS Client Configuration*[54] respectively.

[53] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Endpoint_Configuration
[54] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Client_Configuration

Note

You need to setup both the endpoint configuration and the WSSE declarations. That's two separate steps.

5.25.4.2. 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>jbosssws</key-store-password>
(3)  <trust-store-file>WEB-INF/wsse.truststore</trust-store-file>
(4)  <trust-store-password>jbosssws</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 encypted using the {EXT} and {CLASS} commands. Please see samples for their usage.
3. This specifies that the trust store we wish to use is **`WEB-INF/wsse.truststore`**, which is located in our war file.
4. This specifies that the trust store password is also "jbossws". Password can be encrypted using the {EXT} and {CLASS} commands. Please see samples for their usage.
5. Here we start our root config block. The root config block is the default configuration for all services in this war file.
6. This means that the server must sign the message body of all responses. Type means that we are to use a X.509v3 certificate (a standard certificate). The alias option says that the certificate 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. Use the proprietary @EndpointConfig annotation to set the config name. See *JAX-WS_Endpoint_Configuration*[55] for the list of available config names.

```
@WebService
@EndpointConfig(configName = "Standard WSSecurity Endpoint")
public class HelloJavaBean
{
...
}
```

5.25.4.3. 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.

5.25.4.3.1. 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"
```

[55] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Endpoint_Configuration

```
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 incomming SOAP message with the details of the security headers ommited. The idea is, that the SOAP body is still plain text, but it is signed in the security header and can therefore not manipulated in transit.

Incomming SOAPMessage

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Header>
<wsse:Security env:mustUnderstand="1" ...>
<wsu:Timestamp wsu:Id="timestamp">...</wsu:Timestamp>
<wsse:BinarySecurityToken ...>
...
</wsse:BinarySecurityToken>
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
...
</ds:Signature>
</wsse:Security>
</env:Header>
<env:Body wsu:Id="element-1-1140197309843-12388840" ...>
<ns1:echoUserType xmlns:ns1="http://org.jboss.ws/samples/wssecurity">
<UserType_1 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<msg>Kermit</msg>
</UserType_1>
</ns1:echoUserType>
</env:Body>
</env:Envelope>
```

5.25.4.4. Installing the BouncyCastle JCE provider (JDK 1.4)

The information below has originaly been provided by *The Legion of the Bouncy Castle*[56].

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:

```
```

[56] http://www.bouncycastle.org/specifications.html#install

```
security.provider.<n>=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Where <n> is the preference you want the provider at.

Note

Issues may arise if the Sun provided providers are not first.

Where you put the jar is mostly up to you, although with jdk1.4 the best (and in some cases only) place to have it is in **`$JAVA_HOME/jre/lib/ext`**. Under Windows there will normally be a JRE and a JDK install of Java if you think you have installed it correctly and it still doesn't work chances are you have added the provider to the installation not being used.

5.25.4.5. Keystore, truststore - What?

Note

If you having a hard time understanding how the different trust- and keystore configurations are used for signature and encryption, then read this thread first: *http://www.jboss.org/index.html?module=bb&op=viewtopic&t=94406*

5.25.5. WS-Transaction

Support for the WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity specifications will be provided by technology recently acquired from Arjuna Technologies Ltd. This technology will be present within the JBoss Transactions 4.2.1 release. Further information can be obtained from the *JBoss Transactions Project*[57]

5.25.6. XML Registries

J2EE 1.4 mandates support for Java API for XML Registries (JAXR). Inclusion of a XML Registry with the J2EE 1.4 certified Application Server is optional. Starting jboss-4.0.2, JBoss ships a UDDI v2.0 compliant registry, the Apache jUDDI registry. We also provide support for JAXR Capability Level 0 (UDDI Registries) via integration of Apache Scout.

This chapter describes how to configure the jUDDI registry in JBoss and some sample code outlines for using JAXR API to publish and query the jUDDI registry.

5.25.6.1. Apache jUDDI Configuration

Configuration of the jUDDI registry happens via an MBean Service that is deployed in the juddi-service.sar archive in the "all" configuration. The configuration of this service can be done in the **`jboss-service.xml`** of the META-INF directory in the **`juddi-service.sar`**

Let us look at the individual configuration items that can be changed.

DataSources configuration

[57] http://labs.jboss.org/portal/jbosstm

```
<!-- 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');
```

5.25.6.2. 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.ConnectionFactoryIm
jaxr.query.url=http://localhost:8080/juddi/inquiry
jaxr.publish.url=http://localhost:8080/juddi/publish
juddi.proxy.transportClass=org.jboss.jaxr.juddi.transport.SaajTransport
```

Please remember to change the hostname from "localhost" to the hostname of the UDDI service/ JBoss Server.

You can pass the System Properties to the JVM in the following ways:

- When the client code is running inside JBoss (maybe a servlet or an EJB). Then you will need to pass the System properties in the **`run.sh`** 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);
```

5.25.6.3. 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*[58] From J2EE 1.4 JavaDoc: "This is the principal interface implemented by a JAXR provider. A registry client can get this interface from a Connection to a registry. It provides the methods that are used by the client to discover various capability specific interfaces implemented by the JAXR provider."
- *javax.xml.registry.BusinessLifeCycleManager*[59] From J2EE 1.4 JavaDoc: "The **`BusinessLifeCycleManager`** interface, which is exposed by the Registry Service, implements the life cycle management functionality of the Registry as part of a business level API. There is no authentication information provided, because the Connection interface keeps that state and context on behalf of the client."
- *javax.xml.registry.BusinessQueryManager*[60] From J2EE 1.4 JavaDoc: "The **`BusinessQueryManager`** interface, which is exposed by the Registry Service, implements the business style query interface. It is also referred to as the focused query interface."

Let us now look at some of the common programming tasks performed while using the JAXR API:

Getting a JAXR Connection to the registry.

```
String queryurl = System.getProperty("jaxr.query.url", "http://
localhost:8080/juddi/inquiry");
String puburl = System.getProperty("jaxr.publish.url", "http://
localhost:8080/juddi/publish");
..
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL", queryurl);
props.setProperty("javax.xml.registry.lifeCycleManagerURL", puburl);

String transportClass = System.getProperty("juddi.proxy.transportClass",
 "org.jboss.jaxr.juddi.transport.SaajTransport");
System.setProperty("juddi.proxy.transportClass", transportClass);
```

```
// Create the connection, passing it the configuration properties
factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

Authentication with the registry.

```
/**
* Does authentication with the uddi registry
*/
protected void login() throws JAXRException
{
PasswordAuthentication passwdAuth = new PasswordAuthentication(userid,
 passwd.toCharArray());
Set creds = new HashSet();
creds.add(passwdAuth);

connection.setCredentials(creds);
}
```

Save a Business

```
/**
* Creates a Jaxr Organization with 1 or more services
*/
protected Organization createOrganization(String orgname) throws
 JAXRException
{
Organization org = blm.createOrganization(getIString(orgname));
org.setDescription(getIString("JBoss Inc"));
Service service = blm.createService(getIString("JBOSS JAXR Service"));
service.setDescription(getIString("Services of XML Registry"));
//Create serviceBinding
ServiceBinding serviceBinding = blm.createServiceBinding();
serviceBinding.setDescription(blm.createInternationalString("Test Service
 Binding"));

//Turn validation of URI off
serviceBinding.setValidateURI(false);
serviceBinding.setAccessURI("http://testjboss.org");
..
// Add the serviceBinding to the service
service.addServiceBinding(serviceBinding);

User user = blm.createUser();
org.setPrimaryContact(user);
PersonName personName = blm.createPersonName("Anil S");
TelephoneNumber telephoneNumber = blm.createTelephoneNumber();
```

```
telephoneNumber.setNumber("111-111-7777");
telephoneNumber.setType(null);
PostalAddress address = blm.createPostalAddress("111", "My Drive",
 "BuckHead", "GA", "USA", "1111-111", "");
Collection postalAddresses = new ArrayList();
postalAddresses.add(address);
Collection emailAddresses = new ArrayList();
EmailAddress emailAddress = blm.createEmailAddress("anil@apache.org");
emailAddresses.add(emailAddress);

Collection numbers = new ArrayList();
numbers.add(telephoneNumber);
user.setPersonName(personName);
user.setPostalAddresses(postalAddresses);
user.setEmailAddresses(emailAddresses);
user.setTelephoneNumbers(numbers);

ClassificationScheme cScheme = getClassificationScheme("ntis-gov:naics",
 "");
Key cKey = blm.createKey("uuid:C0B9FE13-324F-413D-5A5B-2004DB8E5CC2");
cScheme.setKey(cKey);
Classification classification = blm.createClassification(cScheme, "Computer
 Systems Design and Related Services", "5415");
org.addClassification(classification);
ClassificationScheme cScheme1 = getClassificationScheme("D-U-N-S", "");
Key cKey1 = blm.createKey("uuid:3367C81E-FF1F-4D5A-B202-3EB13AD02423");
cScheme1.setKey(cKey1);
ExternalIdentifier ei = blm.createExternalIdentifier(cScheme1, "D-U-N-S
 number", "08-146-6849");
org.addExternalIdentifier(ei);
org.addService(service);
return org;
}
```

Query a Business

```

/**
* Locale aware Search a business in the registry
*/
public void searchBusiness(String bizname) throws JAXRException
{
try
{
// Get registry service and business query manager
this.getJAXREssentials();

// Define find qualifiers and name patterns
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);
Collection namePatterns = new ArrayList();
```

```
String pattern = "%" + bizname + "%";
LocalizedString ls = blm.createLocalizedString(Locale.getDefault(),
 pattern);
namePatterns.add(ls);

// Find based upon qualifier type and values
BulkResponse response = bqm.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.

5.25.6.4. 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*[61]

5.25.6.5. Resources

- *JAXR Tutorial and Code Camps*[62]
- *J2EE 1.4 Tutorial*[63]
- *J2EE Web Services by Richard Monson-Haefel*[64]

5.25.7. WS-Policy

Since 2.1

The Web Services Policy Framework (WS-Policy) provides a general purpose model and corresponding syntax to describe the policies of a Web Service.

WS-Policy defines a base set of constructs that can be used and extended by other Web services specifications to describe a broad range of service requirements and capabilities.

Current JBoss implementations can instrument a webservice with policies attached at endpoint, port or port-type scope level only. There are two different methods to attach policies: providing a wsdl decorated with policies and policy attachments as defined by specifications, or using JBoss proprietary annotations. The first way has the advantage of being standard, while the second one is much more simple to implement. Of course the wsdl generated by these annotations conforms to standard defined in specifications and can be used with any ws-policy compliant client.

ws-policy specifications only define policy requirements and their attachment method to wsdl through specific extensions. It is out of the scope of ws-policy specifications and thus implementation to define and use the content of assertions. The way these assertions (called domain assertions or domain policies) have to be deployed and used is left to other specification like WS-Security-Policy or more generally to domain specific implementation.

5.25.7.1. Specification

WS-Policy is defined by the combination of the following specifications:

```
* <ulink url="http://www.w3.org/Submission/WS-Policy/"> WS-Policy
 specification</ulink>
* <ulink url="http://www.w3.org/Submission/WS-PolicyAttachment/"> WS-
Policy-Attachment specification</ulink>
```

5.25.7.2. Using policies in a user provided WSDL

To attach policies in this manner, the only thing you have to do in a webservice class is to provide a custom wsdl. This will cause JBossws to skip wsdl generation at deploy time, since the wsdl file you provided will be published. Please refer to specification (WS-Policy-Attachment) to learn how to modify wsdl to attach a policy.

Here you find an example of a webservice class and provided wsdl with a policy containing a domain assertion for JBoss WS-Security.

```
@WebService(name = "Hello",
targetNamespace = "http://org.jboss.ws/samples/wssecuritypolicy",
wsdlLocation="WEB-INF/wsdl/HelloService.wsdl")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class HelloJavaBean
{
private Logger log = Logger.getLogger(HelloJavaBean.class);
..
@WebMethod
public UserType echoUserType(@WebParam(name = "user") UserType in0)
{
log.info(in0);
return in0;
}
}
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='HelloService' targetNamespace='http://org.jboss.ws/
samples/wssecuritypolicy' xmlns='http://schemas.xmlsoap.org/wsdl/'
 xmlns:ns1='http://org.jboss.ws/samples/wssecurity' xmlns:soap='http://
schemas.xmlsoap.org/wsdl/soap/' xmlns:tns='http://org.jboss.ws/samples/
wssecuritypolicy' xmlns:wsp='http://schemas.xmlsoap.org/ws/2004/09/policy'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
<types>
<xs:schema targetNamespace='http://org.jboss.ws/samples/wssecurity'
 version='1.0' xmlns:xs='http://www.w3.org/2001/XMLSchema'>
<xs:complexType name='UserType'>
<xs:sequence>
<xs:element minOccurs='0' name='msg' type='xs:string'/>
</xs:sequence>
</xs:complexType>
</xs:schema>
</types>
<wsp:Policy wsu:Id='X509EndpointPolicy' xmlns:wsu='http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'>
<wsp:All>
<sp:jboss-ws-security xmlns:sp='http://www.jboss.com/ws-security/schema/
jboss-ws-security_1_0.xsd'>
<sp:key-store-file>WEB-INF/wsse.keystore</sp:key-store-file>
<sp:key-store-password>jbossws</sp:key-store-password>
<sp:trust-store-file>WEB-INF/wsse.truststore</sp:trust-store-file>
<sp:trust-store-password>jbossws</sp:trust-store-password>
<sp:config>
<sp:encrypt alias='wsse' type='x509v3'/>
<sp:requires>
<sp:encryption/>
</sp:requires>
</sp:config>
</sp:jboss-ws-security>
</wsp:All>
</wsp:Policy>
```

```
<message name='Hello_echoUserType'>
<part name='user' type='ns1:UserType'/>
</message>
<message name='Hello_echoUserTypeResponse'>
<part name='return' type='ns1:UserType'/>
</message>
<portType name='Hello'>
<operation name='echoUserType' parameterOrder='user'>
<input message='tns:Hello_echoUserType'/>
<output message='tns:Hello_echoUserTypeResponse'/>
</operation>
</portType>
<binding name='HelloBinding' type='tns:Hello'>
<wsp:PolicyReference URI='#X509EndpointPolicy'/>
<soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/
>
<operation name='echoUserType'>
<soap:operation soapAction=''/>
<input>
<soap:body namespace='http://org.jboss.ws/samples/wssecuritypolicy'
 use='literal'/>
</input>
<output>
<soap:body namespace='http://org.jboss.ws/samples/wssecuritypolicy'
 use='literal'/>
</output>
</operation>
</binding>
<service name='HelloService'>
<port binding='tns:HelloBinding' name='HelloPort'>
<soap:address location='REPLACE_WITH_ACTUAL_URL'/>
</port>
</service>
</definitions>
```

Please note in the WSDL file the wsp:Policy element and the wsp:PolicyReference in 'HelloBinding' binding Element.

5.25.7.3. Using policies with JBoss annotations

Using JBoss proprietary annotation you only have to provide the policy xml, leaving wsdl generation to the JBossWS deployer.

There are two annotations to use, the first one (@PolicyAttachment) containing an array of the second one (@Policy): this lets you have many policies attached to a class or method. In future domain policy implementations might ship domain annotations extending the @Policy annotation to provide needed metadata directly as annotation parameters. The current @Policy annotation takes a reference to a xml file containing a generic policy description written respecting ws-policy specification rules.

```
/**
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface PolicyAttachment {
Policy[] value();
}
...
@Retention(RetentionPolicy.RUNTIME)
public @interface Policy {

public String policyFileLocation();

public PolicyScopeLevel scope();
}
```

And here you have the previous section example re-implemented using annotations and xml policy file:

```
@WebService(name = "Hello", targetNamespace = "http://org.jboss.ws/samples/
wssecurityAnnotatedpolicy")
@PolicyAttachment({@Policy( policyFileLocation="WEB-INF/Policy.xml", scope
 = PolicyScopeLevel.WSDL_PORT ) })
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class HelloJavaBean
{
private Logger log = Logger.getLogger(HelloJavaBean.class);

@WebMethod
public UserType echoUserType(@WebParam(name = "user") UserType in0)
{
log.info(in0);
return in0;
}
}
<?xml version="1.0" encoding="UTF-8"?>
...
<wsp:Policy wsu:Id="X509EndpointPolicy"  xmlns:wsp="http://
schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd">
<wsp:ExactlyOne>
<wsp:All>
<sp:jboss-ws-security xmlns:sp="http://www.jboss.com/ws-security/schema/
jboss-ws-security_1_0.xsd">
<sp:key-store-file>WEB-INF/wsse.keystore</sp:key-store-file>
<sp:key-store-password>jbossws</sp:key-store-password>
<sp:trust-store-file>WEB-INF/wsse.truststore</sp:trust-store-file>
<sp:trust-store-password>jbossws</sp:trust-store-password>
<sp:config>
<sp:encrypt type="x509v3" alias="wsse"/>
<sp:requires>
```

```
<sp:encryption/>
</sp:requires>
</sp:config>
</sp:jboss-ws-security>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>
```

5.26. JBossWS Extensions

This section describes propriatary JBoss extensions to JAX-WS.

5.26.1. Proprietary Annotations

For the set of standard annotations, please have a look at *JAX-WS Annotations*[65]

5.26.1.1. EndpointConfig

```
/**
* Defines an endpoint or client configuration.
* This annotation is valid on an endpoint implementaion bean or a SEI.
*
* @author Heiko.Braun@jboss.org
* @since 16.01.2007
*/
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = { ElementType.TYPE })
public @interface EndpointConfig {
...
/**
* The optional config-name element gives the configuration name that must
 be present in
* the configuration given by element config-file.
*
* Server side default: Standard Endpoint
* Client side default: Standard Client
*/
String configName() default "";
...
/**
* The optional config-file element is a URL or resource name for the
 configuration.
*
* Server side default: standard-jaxws-endpoint-config.xml
* Client side default: standard-jaxws-client-config.xml
*/
String configFile() default "";
}
```

[65] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Annotations

5.26.1.2. 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;
}
```

5.26.1.3. SecurityDomain

```
/**
* Annotation for specifying the JBoss security domain for an EJB
*
* @author <a href="mailto:bill@jboss.org">Bill Burke</a>
**/
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface SecurityDomain
{
/**
* The required name for the security domain.
*
* Do not use the JNDI name
*
*    Good: "MyDomain"
*    Bad:  "java:/jaas/MyDomain"
*/
String value();

/**
* The name for the unauthenticated pricipal
*/
String unauthenticatedPrincipal() default "";
}
```

[66] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Endpoint_Configuration

5.27. Web Services Appendix

JAX-WS Endpoint Configuration[66]

JAX-WS Client Configuration[67]

JAX-WS Annotations[68]

Common features and properties[69]

5.28. References

1. *JSR-224 - Java API for XML-Based Web Services (JAX-WS) 2.0*[70]
2. *JSR 222 - Java Architecture for XML Binding (JAXB) 2.0*[71]
3. *JSR-261 - Java API for XML Web Services Addressing*[72]
4. *SOAP-1.2 - Messaging Framework*[73]
5. *JSR-250 - Common Annotations for the Java Platform*[74]
6. *JSR 181 - Web Services Metadata for the Java Platform*[75]

[67] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Client_Configuration
[68] http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_Annotations
[69] http://jbws.dyndns.org/mediawiki/index.php?title=Common_features_and_properties

JBoss5 Virtual Deployment Framework

As indicated in *Chapter 1, Introduction* the JBoss Enterprise Application Platform 5 is designed around the advanced concept of a Virtual Deployment Framework (VDF). This chapter discusses the JBoss5 Virtual Deployment Framework further. The following UML diagram illustrates an overview of the key JBoss5 Deployment Framework classes.

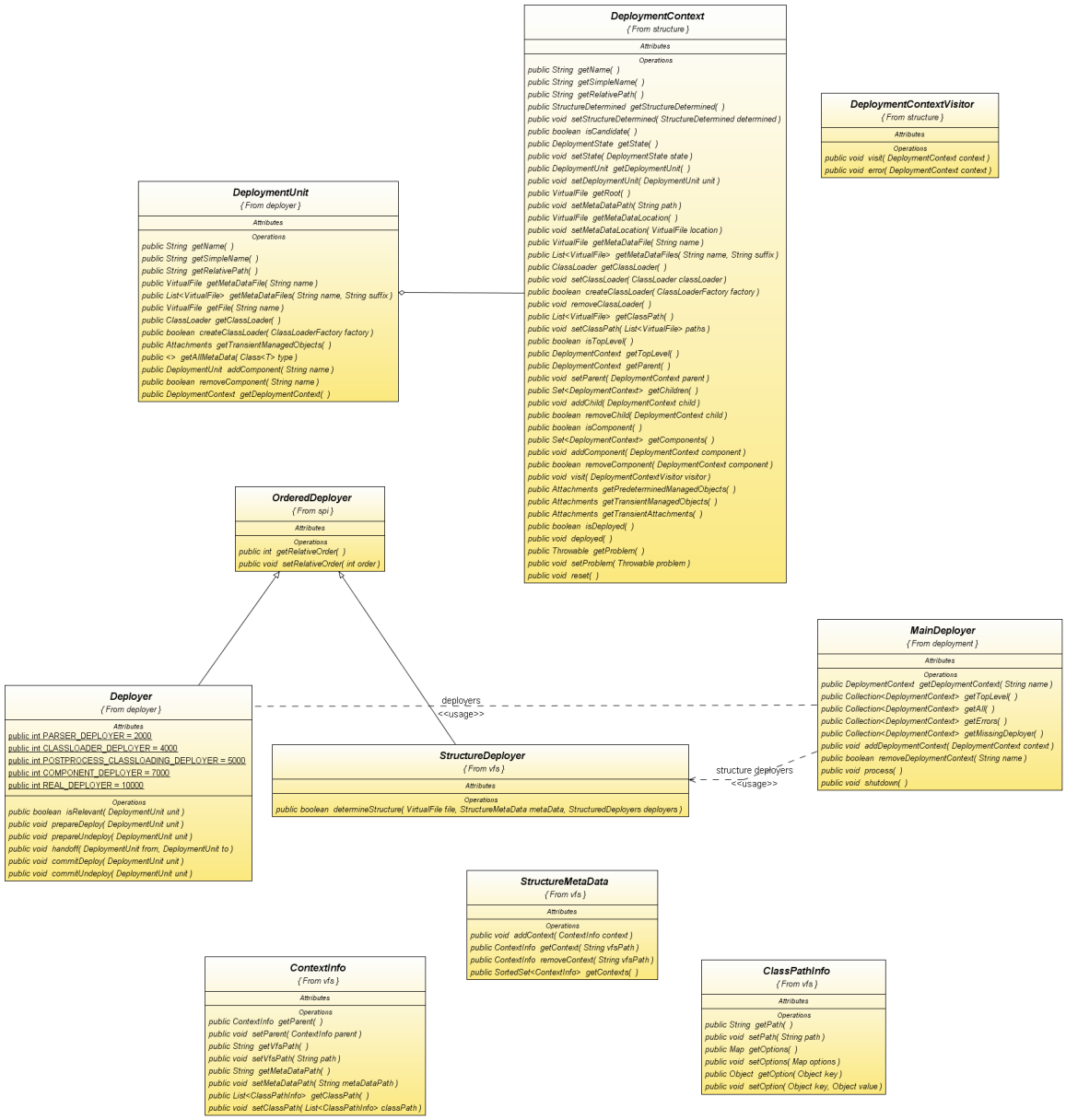


Figure 6.1. The JBoss5 Deployment Framework Classes

The key classes in the above diagram are:

- *MainDeployer* : this interface defines the contract for the MainDeployer. The MainDeployer handles parsing of deployment archives into Deployment instances and deployment of those instances into the microcontainer. This update of the JMX based MainDeployer moves it to one based on the Microcontainer, JBoss5VirtualFileSystem, and Virtual Deployment Framework (VDF). Deployers are registered with the MainDeployer as an ordered list of deployers. MainDeployer contains two sets of deployers:

  - *StructureDeployers* used to analyze the structure of a DeploymentContext when `addDeploymentContext(DeploymentContext)` is invoked. For each StructureDeployer the `determineStructure(DeploymentContext)` method is invoked to analyze the deployment. A StructureDeployer returns true to indicate that the deployment was recognized and no further StructureDeployer should analyze the DeploymentContext.

- Deployers used to translate a DeploymentUnit into runtime kernel beans when the MainDeployer.process is run. The Deployer methods are:
  - isRelevant() : does the deployer want to process the unit.
  - prepareDeploy() : take the new deployment to the ready stage
  - prepareUndeploy() : get ready to undeploy
  - handoff(new, old) : handover control from new to old
  - commitDeploy() : new deployment is now in control
  - commitUndeploy() : old deployment is out of here
  - getRelativeOrder() : specify the relative order of the deployer in a chain

.
- DeploymentUnit : a representation of a runtime unit of work a Deployer operates on.
- DeploymentContext : a representation of structural aspects of deployable content.
- ManagedObject : a representation of the manageable properties for a deployment.
- VFS : the api for representing the read-only file system of the deployment.
- VirtualFile : the api for a file in the deployment.
- DomainClassLoader and ClassLoadingDomain : A generalization of the legacy JMX based unified class loading model. This is still in progress. The **`org.jboss.vfs.classloding.VFSClassLoader`** is the current simple implementation.

6.1. MainDeployerImpl

The **`org.jboss.deployers.plugins.deployment.MainDeployerImpl`** implementation of the **`org.jboss.deployers.spi.deployment.MainDeployer`** interfaces, includes the following standard method details:

- `DeploymentContext getDeploymentContext(String name)`: obtain the **`DeploymentContext`** associated with the given name from all of the **`DeploymentContext`**s that have been added to the **`MainDeployer`**. This includes top level and all child contexts.
- `Collection <DeploymentContext> getTopLevel()`: get a list of all of the top level **`DeploymentContext`**s added via the `addDeploymentContext(DeploymentContext)` method.
- `Collection <DeploymentContext> getAll()`: get all of the **`DeploymentContext`**s, top-level and child associated with the **`MainDeployer`**.

- `Collection <DeploymentContext> getErrors()`: get the **`DeploymentContext`**s that have failed to be structurally analyzed or deployed.
- `Collection <DeploymentContext> getMissingDeployer()`: get the **`DeploymentContext`**s that are not deployed (`isDeployed() == false`) and are not root **`.jar`** files.
- `void addDeploymentContext(DeploymentContext context) throws DeploymentException`: add a top-level deployment context. This runs a structural analysis of the **`DeploymentContext`** if its **`StructureDetermined`** state is not PREDETERMINED. If the structural analysis succeeds, the **`DeploymentContext`** is added for deployment during process.
- `boolean removeDeploymentContext(String name) throws DeploymentException`: remove the top-level deployment associated with name.
- `void process()`: runs through all**`DeploymentContext`**s that have been removed and undeploys each top-level **`DeploymentContext`**. The undeployment involves invoking the `performUndeploy(DeploymentUnit)` method on each `DeploymentContext.getDeploymentUnit()` method. Then for each **`DeploymentContext`**, `performUndeploy(DeploymentUnit)` on the component `DeploymentContext.getDeploymentUnit()` is performed. Next, the top-level **`DeploymentContext`**s that have been added are deployed by invoking `commitDeploy` on each deployer. The details of the deployment process are that each deployer is run on top-level context **`DeploymentUnit`** by invoking `Deployer.commitDeploy(DeploymentUnit)`, followed by the deployment of each context of the top-level **`DeploymentContext`** components (`DeploymentContext.getComponents()`).
- `void shutdown()`: removes all top-level **`DeploymentContext`**s, and then invokes the undeployment process.

In addition, the implementation adds the following methods.

- `public synchronized void addDeployer(Deployer deployer)`: add a component deployer for non-structurual processing.
- `public synchronized void removeDeployer(Deployer deployer)`: removes a component Deployer.
- `public synchronized Set <Deployer> getDeployers()`: get the registered component deployers.
- `public synchronized void setDeployers(Set<Deployer> deployers)`: set the component deployers.
- `public synchronized void addStructureDeployer(StructureDeployer deployer)`: add a structural deployer.
- `public synchronized void removeStructureDeployer(StructureDeployer deployer)`: remove a structural deployer.
- `public synchronized Set<StructureDeployer> getStructureDeployers()`: obtain the registered structural deployers.
- `public synchronized void setStructureDeployers(Set<StructureDeployer> deployers)` : set the structural deployers.

6.2. JBoss5StructureDeployerClasses

org.jboss.deployers.plugins.structure.vfs.AbstractStructureDeployer

- org.jboss.deployers.plugins.structure.vfs.file.FileStructure
- org.jboss.deployers.plugins.structure.vfs.jar.JARStructure
- org.jboss.deployers.plugins.structure.vfs.war.WARStructure

6.3. Deployer Helper and Base Classes

JBoss5BaseDeployerClasses

`org.jboss.deployers.plugins.deployer.AbstractDeployer`: forces `isRelevant` to return true and `getRelativeOrder` to return Integer.MAX_VALUE.

- `org.jboss.deployers.plugins.deployers.helpers.AbstractSimpleDeployer`: collapses the **`Deployer`** contract to `deploy(DeploymentUnit)` and `undeploy(DeploymentUnit)` by forcing:
  - `prepareDeploy` to not undertake anything
  - `commitDeploy` to call `deploy`
  - `prepareUndeploy` to call `undeploy`
  - `commitUndeploy` to not undertake anything
  - `handoff` to not undertake anything
  - `org.jboss.deployers.plugins.deployers.helpers.AbstractClassLoaderDeployer` implements `org.jboss.deployers.spi.classloader.ClassLoaderFactory` and `deploy(DeploymentUnit u)` as `u.createClassLoader(this)`.
- `org.jboss.deployers.plugins.deployers.helpers.AbstractTopLevelClassLoaderDeployer` adds `createTopLevelClassLoader(DeploymentContext)` and `removeTopLevelClassLoader(DeploymentContext)` methods. It also implements `createClassLoader` to invoke `createTopLevelClassLoader` if `context.isTopLevel()` is true. Otherwise it will return the value of `context.getTopLevel().getClassLoader()`.
- `org.jboss.deployers.plugins.deployers.helpers.AbstractRealDeployer<T>` adds an attachment type T known as the deploymentType and a **`SimpleDeploymentVisitor<T>`** visitor. The deployment implementation obtains a deploymentType metadata from the deployment unit and then delegates to the `visitor.deploy(DeploymentUnit, metadata)` method for each deploymentType metadata. Undeploy similarly delegates to `visitor.undeploy(DeploymentUnit, metadata)`.
- `org.jboss.deployers.plugins.deployers.helpers.AbstractComponentDeployer<D, C>`: In addition to a deployment type D, a component type C is introduced along with a **`SimpleDeploymentVisitor<C>`** compVisitor. `Deployer.deploy(DeploymentUnit)` invokes `super.deploy(unit)` to process the deployment type metadata, and then obtains `unit.getAllMetaData(C)` and delegates to `compVisitor.deploy(unit, metadata)` to process the component metadata. Undeploy similarly invokes `super.undeploy(unit)` and the

delegates to `compVisitor.undeploy(unit, metadata)`. The component visitor is expected to create **`DeploymentUnit`** components (`DeploymentUnit.addComponent(String)`) for the component metadata.

- `org.jboss.deployers.plugins.deployers.helpers.AbstractTypedDeployer<T>` adds an attachment type T known as the deploymentType and accessor and contains new features.
  - `org.jboss.deployers.plugins.deployers.helpers.AbstractParsingDeployer<T>` adds a notion of obtaining an instance of the deploymentType by parsing a metadata file. The helper methods added include:
- `protected T getMetaData(DeploymentUnit unit, String key)` returns `unit.getAttachment(key, getDeploymentType());`
- `protected void createMetaData(DeploymentUnit unit, String name, String suffix)` calls `createMetaData(unit, name, suffix, getDeploymentType().getName());`
- `protected void createMetaData(DeploymentUnit unit, String name, String suffix, String key)` calls `parse(unit, name)` if suffix is null and `parse(unit, name, suffix)` otherwise. The result is added as an attachment to `unit.getTransientManagedObjects()` under key with expected type T.
- `protected T parse(DeploymentUnit unit, String name)` locates `VirtualFile unit.getMetaDataFile(name)`, and if it is found, calls `T result = parse(unit, file); init(unit, result, file);`
- `protected T parse(DeploymentUnit unit, String name, String suffix)` locates `List<VirtualFile> files = unit.getMetaDataFiles(name, suffix)`, and if found, calls `T result = parse(unit, files.get(0)); init(unit, result, file);`
- `protected abstract T parse(DeploymentUnit unit, VirtualFile file)` is an abstract method.
- `protected void init(DeploymentUnit unit, T metaData, VirtualFile file)` is empty.
- `org.jboss.deployers.plugins.deployers.helpers.JAXPDeployer<T>` implements `parse(DeploymentUnit unit, VirtualFile file)` to obtain the org.w3c.dom.Document corresponding to a file using JAXP **`DocumentBuilder`** and a file using **`InputStream`**. This is parsed into deploymentType T by `calling parse(unit, file, document)`.
  - `protected abstract T parse(DeploymentUnit unit, VirtualFile file, Document document) throws Exception` is an abstract method.
  - `org.jboss.deployers.plugins.deployers.helpers.XSLDeployer<T>` adds an `xslPath` that corresponds to a class loader resource for an XSL document. It also overrides the `parse(DeploymentUnit unit, VirtualFile file)` method to transform the JAXP document obtained from `JAXPDeployer.doParse`, and then parses this into deploymentType T by calling the abstract method, `parse(unit, file, document)`.
- `org.jboss.deployers.plugins.deployers.helpers.ObjectModelFactoryDeployer<T>` adds an abstract JBossXB **`ObjectModelFactory`** accessor that is used from within an overriden

`parse(DeploymentUnit unit, VirtualFile file)` to unmarshall the XML document represented by file into an instance of deploymentType T.

- `org.jboss.deployers.plugins.deployers.helpers.SchemaResolverDeployer<T>` uses JBossXB **`UnmarshallerFactory`** with a **`SchemaBindingResolver`** from within an overriden `parse(DeploymentUnit unit, VirtualFile file)` to unmarshall the XML document represented by the file into an instance of deploymentType T. The XML document must have a valid schema with JBossXB annotations.
- `org.jboss.deployers.plugins.deployers.helpers.AbstractSimpleRealDeployer<T>` adds two abstract methods:
  - `public abstract void deploy(DeploymentUnit unit, T deployment);`
  - `public abstract void undeploy(DeploymentUnit unit, T deployment);`
  - Overrides `deploy(DeploymentUnit unit)` to obtain the **`deploymentType`** instance using `unit.unit.getAttachment(getDeploymentType())`, and invokes `deploy(DeploymentUnit unit, T deployment)`.
  - Overrides `undeploy(DeploymentUnit unit)` to obtain the **`deploymentType`** instance using `unit.unit.getAttachment(getDeploymentType())`, and invokes `undeploy(DeploymentUnit unit, T deployment)`.

6.4. Current Deployers

- **`org.jboss.deployers.plugins.deployers.kernel.BeanDeployer`**
- **`org.jboss.deployers.plugins.deployers.kernel.KernelDeploymentDeployer`**
- **`org.jboss.deployers.plugins.deployers.kernel.BeanMetaDataDeployer`**
- **`ServiceDeployments`**
- **`org.jboss.system.deployers.SARDeployer`**
- **`org.jboss.system.deployers.ServiceClassLoaderDeployer`**
- **`org.jboss.system.deployers.ServiceDeploymentDeployer`**
- **`org.jboss.system.deployers.ServiceDeployer`**
- JBoss5WebDeployments
  - **`org.jboss.deployment.WebAppParsingDeployer`**
  - **`org.jboss.deployment.JBossWebAppParsingDeployer`**
  - **`org.jboss.web.tomcat.tc6.deployers.TomcatDeployer`**
- **`org.jboss.resource.deployers.RARDeployer`**
- **`org.jboss.resource.deployers.RARParserDeployer`**

6.5. Virtual File System JBoss5VirtualFileSystem

The virtual file system model of the deployment framework provides a consistent API for accessing logical files in logical file systems referenced by a URI/URL.

- Virtual File System (VFS): the main API for accessing read-only file system of the deployment. A VFS instance represents a virtual file system mount for a given root URI/URL.
- **`VirtualFile`**: the API for a file in the deployment.

JBoss AOP

JBoss AOP is a 100% Pure Java Aspected 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's 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.

7.1. 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 joinpiont 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, i.e., the behavior that is triggered when a method is called. It could also be thought of as the code that does 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.

Introductions

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's 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* block to each and every method or constructor you want to benchmark.

2. The profiling code really doesn't belong sprinkled throughout your application code. It makes your code bloated and harder to read, as you have to enclose the timings within a try/finally block.

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's dispersed throughout your entire code base. And this is just a tiny example! 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.

7.2. 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.

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

The following listing 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 Metric'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.

7.3. 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/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. Listing Three shows how to define a pointcut for the metrics example.

The following listing demonstrates defining a pointcut 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 >]]></programlisting>
```

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 do not like XML. See our Reference Guide for more information.

JBoss AOP has a rich set of pointcut expressions that you can use to define various points/events in your Java application so that you can apply your aspects. 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're able to pull together crosscutting behavior into one object and apply it easily and simply, without polluting and bloating your code with features that ultimately don't belong mingled with business logic. Instead, common crosscutting concerns can be maintained and extended in one place.

Notice too that the code within the BankAccountDAO class has no idea that it's being profiled. This is what aspect-oriented programmers deem orthogonal concerns. Profiling is an orthogonal concern. In the OOP code snippet in Listing One, profiling was part of the application code. With AOP, you can remove that code. A modern promise of middleware is transparency, and AOP (pardon the pun) clearly delivers.

Just as important, orthogonal behavior could be bolted on after development. In Listing One, 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 (obliviousness, some may say) 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.

JBoss Transactions

JBoss Transactions runs in the *all* server configuration or customized configurations based on the *all* configuration.

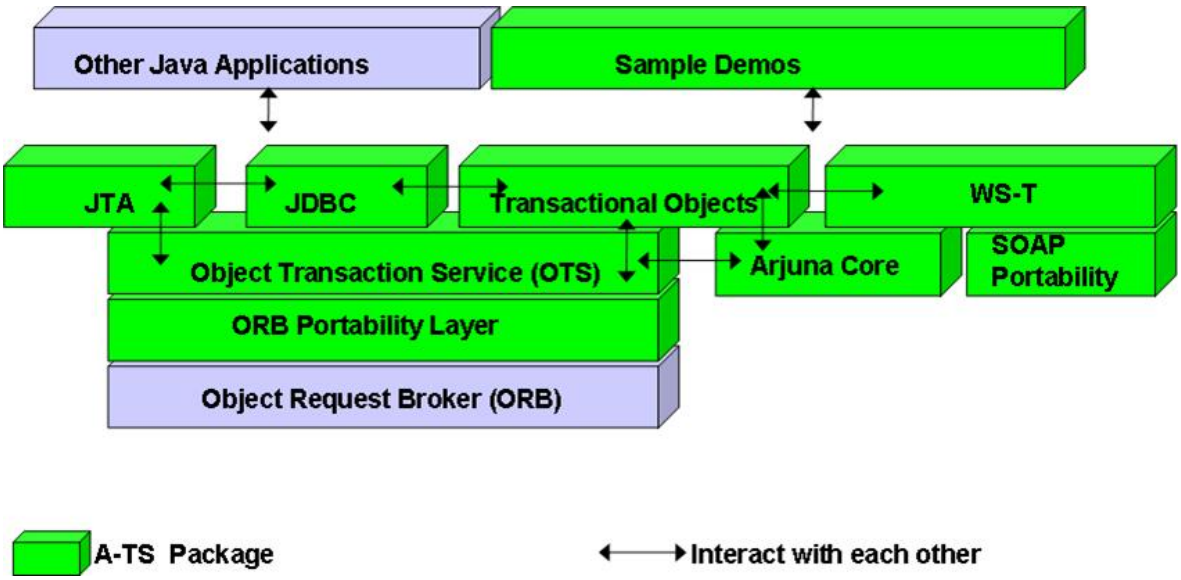


Figure 8.1. Transactions Architecture

8.1. Why you need JBoss Transactions

In todays business environment data corruption can have serious consequences for the enterprise including service unavailability, system reconciliation costs, and damage to customer relationships and business reputation. The JBoss Transaction Service (JBossTS) protects businesses from data corruption by guaranteeing complete, accurate business transactions for Java based applications (including those written for the JEE and EJB frameworks) thereby eliminating the risks and costs associated with time-consuming manual reconciliation that follow failures.

8.2. JBoss Transactions Java EE 5 Support

In the modern business environment of system consolidations, worldwide utilization, and *always on* availability, enterprises need distributed transaction processing infrastructure. This enables businesses to build reliable, sophisticated applications that can guarantee absolute completion and accuracy of business processes. Transaction services ensure that sequences of database updates have been accurately and reliably committed as a single complete unit of work or that, in the event of failure, the database information is recovered. *Multimodal Transaction Processing* is the term coined by Gartner to describe the new generation of transactional application required to face the challenges posed by new business requirements, technologies and innovative computing architectures.

Note

"Multimodal transaction processing will emerge. Users' adoption of client/server, the Internet, service-oriented architecture, Web services, mobile and wireless devices, and event-driven architectures means that the next generation of transaction processing applications will have to be implemented in very different ways to respond to new business strategies, including multichannel, the real-time enterprise and business process

> fusion." Predicts 2004: Prepare for Multimodal Transaction Processing, M. Pezzini, Gartner, 19 December 2003

JBoss Transaction Services is a middleware solution that supports mission-critical applications in distributed computing environments. It plays a critical role in building reliable, sophisticated e-business applications guaranteeing absolute completion and accuracy of business processes; supporting *multimodal transaction processing* by enabling reliable transactions to span from front-end e-commerce applications to back office systems and beyond the enterprise firewall to business partners across any system, anywhere in the world.

Building on the industry proven Java EE 5 transaction technology, native support is included for Web service transactions by providing all of the components necessary to build interoperable, reliable, multi-party, Web service-based applications with minimal effort. The product supports the **`WS-AtomicTransaction`** and **`WS-BusinessActivity`** specifications.

8.3. JBoss Transactions Web Services Support

In traditional ACID transaction systems, transactions are short lived, resources (such as databases) are locked for the duration of the transaction and participants have a high degree of trust with each other. With the arival of the Internet and Web services, the scenario that is now emerging requires involvement of participants unknown to each other in distributed transactions. These transactions have the following characteristics:

1. Transactions may be of a long duration, sometimes lasting hours, days, or more.
2. Participants may not allow their resources to be locked for long durations.
3. The communication infrastructure between participants may not be reliable.
4. Some of the ACID properties of traditional transactions are not mandatory.
5. A transaction may succeed even if only some of the participants choose to confirm and others cancel.
6. All participants may choose to have their own coordinator (Transaction Manager), because of lack of trust.
7. All activities are logged.
8. Transactions that have to be rolled back have the concept of compensation.

8.4. How JBossTS address these issues

Programming interfaces are based on the Java API for XML Transactioning (JAXTX) and the product includes protocol support for the **`WS-AtomicTransaction`** and **`WS-BusinessActivity`** specifications. JBoss Transaction Services included with the JBoss Enterprise Application Platform is designed to support multiple coordination protocols and assist to future-proof transactional applications. For a more detailed description of the product capabilities, see the full feature list below.

JBoss Transaction Services is a pure Java multi-modal transaction service that supports distributed transactions in CORBA, JEE and Web services environments.

1. Standards compliance

   a. CORBA Object Transaction Service (OTS)
   b. Java Enterprise (JEE) transactions
      i. Java Transaction API (JTA)
      ii. Java Transaction Service (JTS)
   c. Web services transactions
      i. Web Services Coordination (WS-Coordination)
      ii. Web Services Atomic Transaction (WS-AtomicTransaction)
      iii. Web Services Business Activity Framework (WS-BusinessActivity)
2. JEE and CORBA transactioning features
   a. Complete distributed transaction support
   b. Automated failure recovery system
   c. Flexible deployment: centralized and distributed transaction manager options
   d. Interposition support for improved distributed transaction performance
   e. POA ORB support
   f. Support for both checked and unchecked transaction behaviour
   g. Support for both flat and nested transaction models, with nested-aware resources and resource adapters
   h. Support for CosTransaction::Current API
   i. Direct and indirect transaction management
   j. Synchronization interface support
   k. Transaction heuristic support
   l. Explicit and implicit transaction context propagation
   m. Multi-thread aware
3. Web services transactioning features
   a. Ensures reliable coordination and application data consistency for business processes that involve Web services.
   b. Supports transaction models for both intra-enterprise (EAI) and inter-enterprise (supply chain) Web services integration.
   c. Allows for consistent real-time updates across any component or resource involved in the business process.

d. Fully automated crash recovery provides fast, unattended restoration of service after component failures.

e. Future-proof, generic coordination engine architecture supports pluggable protocols.

f. Currently supports the WS-Coordination WS- AtomicTransaction and WS-BusinessActivity specifications. Supports the leveraging of existing transaction infrastructure investments.

g. Architected for portability across a wide- range of Web services platforms. Supports the open source JBoss application server for highly cost effective development and deployment.

h. Close integration with enterprise Java standards, allowing Web services transactions to seamlessly integrate with JEE application servers, messaging systems and database back-ends.

i. Easy to use Java programming interfaces, based on the emerging JAXTX standard. A rich programming framework reduces overhead in adding transactioning capabilities to Web services.

j. Leverages Arjuna's long history in transaction software, including the industry proven coordination engine, ArjunaCore - as used in the Bluestone and HP application servers.

Remoting

The main objective of JBoss Remoting is to provide a single API for most network based invocations and related services that use pluggable transports and data marshallers. The JBoss Remoting API provides the ability for making synchronous and asynchronous remote calls, push and pull callbacks, and automatic discovery of remoting servers. The intention is to allow for the addition of different transports to fit different needs, yet still maintain the same API for making the remote invocations and only requiring configuration changes, not code changes, to fit these different needs.

JBoss Remoting can be run as a service within the container with this chapter discussing the JBoss Remoting service configurations.

9.1. Summary of JBoss Remoting Features

The features available with JBoss Remoting are:

- *Server identification*: A simple URL based identifier which allows for remoting servers to be identified and called upon.
- *Pluggable transports*: Uses different protocol transports but the same remoting API. The provided transports are:
  - Socket (SSL Socket)
  - RMI (SSL RMI)
  - HTTP(S)
  - Multiplex (SSL Multiplex)
  - Servlet (SSL Servlet)
  - BiSocket (SSL BiSocket)
- *Pluggable data marshallers*: Uses different data marshallers and unmarshallers to convert the invocation payloads into desired data formats for wire transfer.
- *Pluggable serialization*: Uses different serialization implementations for data streams. The provided serialization implementations are:
  - Java serialization
  - JBoss serialization
- *Automatic discovery*: Detects remoting servers as they come on and off line. Provided detection implementations are:
  - Multicast
  - JNDI
- *Server grouping*: Ability to group servers by logical domains, so communication only occurs with servers within specified domains.
- *Callbacks*: Receive server callbacks via push and pull models. The pull model specifically allows for persistent stores and memory management.

- *Asynchronous calls*: Make asynchronous, or one way, calls to a server.
- *Local invocation*: If you are making an invocation on a remoting server that is within the same process space, Remoting will automatically make this call by reference to improve performance.
- *Remote classloading*: Allows for classes, such as custom marshallers, that do not exist within the client, to be loaded from server.
- *Sending of streams*: Allows for clients to send input streams to the server, which can be read from the server on demand.
- *Clustering*: Seamless client failover for remote invocations.
- *Connection failure notification* - notification if client or server has failed.
- *Data Compression*: Uses the compression marshaller and unmarshaller for the compresssion of large payloads.

All the features within JBoss Remoting were created with ease of use and extensibility in mind. If you have a suggestion for a new feature or an improvement to a current feature, please log these in the issue tracking system at *http://jira.jboss.com*.

9.2. JBoss Remoting Configuration in the JBoss Enterprise Application Platform

As indicated earlier in this chapter, JBoss Remoting manages synchronous and asynchronous remote calls, push and pull callbacks, and automatic discovery of Remoting servers. You can configure JBoss Remoting through the JBoss Messaging service configuration file **`JBOSS_HOME/server/<your_configuration>/deploy/messaging/remoting-service.xml`**.

JBoss Messaging

JBoss Messaging is the new enterprise messaging system from JBoss. It is a complete rewrite of JBossMQ, the legacy JBoss JMS provider. It is the default JMS provider on JBoss Enterprise Application Platform 5.

JBoss Messaging is a high Performance JMS 1.1 compliant implementation integrated with JBoss Transactions. It also offers:

- Clustered Queues and Topics by Default
- Intelligent Message Redistributions
- Transparent Failover
- In memory message Replication

JBoss Messaging is an integral part of Red Hat's strategy for messaging.

JBoss Messaging provides an open source and standards-based messaging platform that brings enterprise-class messaging to the mass market. It also implements a high performance, robust messaging core that is designed to support the largest and most heavily utilized SOA, enterprise service buses (ESBs) and other integration needs ranging from the simplest to the highest network demands.

It allows you to smoothly distribute your application load across your cluster, intelligently balancing and utilizing each nodes CPU cycles, with no single point of failure. This provides a highly scalable and performance implementation for clustering.

JBoss Messaging includes a JMS front-end to deliver messaging in a standards-based format as well as being designed to be able to support other messaging protocols in the future.

JBoss Messaging is committed to AMQP (*AMQP*[1])- the new messaging standard from Red Hat and others. Later versions of JBoss Messaging will support AMQP, and JBoss Messaging is focused on becoming the premier AMQP Java broker.

10.1. Configuring JBoss Messaging

The JBoss Messaging service configuration is spread among several configuration files. Depending on the functionality provided by the services it configures, the configuration data is distributed between **`<JBOSS_HOME>/server/<configuration>/deploy/messaging-service.xml, remoting-service.xml, connection-factories-service.xml, destinations-service.xml and xxx-persistence-service.xml`** (where **`xxx`** is the name of your database). The default will be **`hsqldb-persistence-service.xml`** for the hsqldb database.

10.1.1. Configuring the SecurityStore

SecurityStore is a pluggable object, and it has a default implementation in **`messaging-service.xml`**.

```
<server>
 <mbean code="org.jboss.jms.server.security.SecurityMetadataStore"
```

[1] http://www.amqp.org/

```
 name="jboss.messaging:service=SecurityStore">

 <attribute name="DefaultSecurityConfig">
 <security>
 <role name="guest" read="true" write="true" create="true"/>
 </security>
 </attribute>

 <attribute name="SecurityDomain">java:/jaas/messaging</attribute>

 <attribute name="SuckerPassword">CHANGE ME!!</attribute>
</mbean>
...
...file truncated..
```

10.1.2. SecurityStore Attributes

The following are SecurityStore attributes from the **`messaging-service.xml`** file above.

DefaultSecurityConfig

Default security configuration is used when the security configuration for a specific queue or topic has not been overridden in the destination's deployment descriptor. It has exactly the same syntax and semantics as in JBossMQ.

The DefaultSecurityConfig attribute element should contain one <security> element. The <security> element can contain multiple <role> elements. Each <role> element defines the default access for that particular role.

If the read attribute is true then that role will be able to read (create consumers, receive messaages or browse) destinations by default. If the write attribute is true then that role will be able to write (create producers or send messages) to destinations by default. If the create attribute is true then that role will be able to create durable subscriptions on topics by default.

SecurityDomain

The JAAS security domain to be used by this server peer.

SuckerPassword

This defines how the SecurityStore will authenticate the sucker user (**`JBM.SUCKER`**).

10.2. Configuring the ServerPeer

The **`ServerPeer`** is the heart of the JBoss Messaging JMS. All JBoss Messaging services are rooted at the server peer and the server's configuration resides in the **`messaging-service.xml`** configuration file. An example of a Server Peer configuration is presented below, though not all values for the server peer's attributes are specified in the example.

```
<!-- ServerPeer MBean configuration
 ================================ -->
```

```
 <mbean code="org.jboss.jms.server.ServerPeer"
 name="jboss.messaging:service=ServerPeer"
 xmbean-dd="xmdesc/ServerPeer-xmbean.xml">

 <!-- The unique id of the server peer - in a cluster each node MUST have a
 unique value - must be an integer -->

 <attribute name="ServerPeerID">${jboss.messaging.ServerPeerID:0}</
attribute>

 <!-- The default JNDI context to use for queues when they are deployed
 without specifying one -->

 <attribute name="DefaultQueueJNDIContext">/queue</attribute>

 <!-- The default JNDI context to use for topics when they are deployed
 without specifying one -->

 <attribute name="DefaultTopicJNDIContext">/topic</attribute>

 <attribute name="PostOffice">jboss.messaging:service=PostOffice</
attribute>

 <!-- The default Dead Letter Queue (DLQ) to use for destinations.
 This can be overridden on a per destinatin basis -->

 <attribute
 name="DefaultDLQ">jboss.messaging.destination:service=Queue,name=DLQ</
attribute>

 <!-- The default maximum number of times to attempt delivery of a message
 before sending to the DLQ (if configured).
 This can be overridden on a per destinatin basis -->

 <attribute name="DefaultMaxDeliveryAttempts">10</attribute>

 <!-- The default Expiry Queue to use for destinations. This can be
 overridden on a per destinatin basis -->

 <attribute
 name="DefaultExpiryQueue">jboss.messaging.destination:service=Queue,name=ExpiryQueue
attribute>

 <!-- The default redelivery delay to impose. This can be overridden on a
 per destination basis -->

 <attribute name="DefaultRedeliveryDelay">0</attribute>

 <!-- The periodicity of the message counter manager enquiring on queues
 for statistics -->
```

```
 <attribute name="MessageCounterSamplePeriod">5000</attribute>

 <!-- The maximum amount of time for a client to wait for failover to start
 on the server side after
 it has detected failure -->

 <attribute name="FailoverStartTimeout">60000</attribute>

 <!-- The maximum amount of time for a client to wait for failover to
 complete on the server side after
 it has detected failure -->

 <attribute name="FailoverCompleteTimeout">300000</attribute>

 <attribute name="StrictTck">false</attribute>

 <!-- The maximum number of days results to maintain in the message counter
 history -->

 <attribute name="DefaultMessageCounterHistoryDayLimit">-1</attribute>

 <!-- The name of the connection factory to use for creating connections
 between nodes to pull messages -->

 <attribute
 name="ClusterPullConnectionFactoryName">jboss.messaging.connectionfactory:service=Clusterl
attribute>

 <!-- When redistributing messages in the cluster. Do we need to preserve
 the order of messages received
 by a particular consumer from a particular producer? -->

 <attribute name="DefaultPreserveOrdering">false</attribute>

 <!-- Max. time to hold previously delivered messages back waiting for
 clients to reconnect after failover -->

 <attribute name="RecoverDeliveriesTimeout">300000</attribute>

 <!-- The password used by the message sucker connections to create
 connections.
 THIS SHOULD ALWAYS BE CHANGED AT INSTALL TIME TO SECURE SYSTEM
 <attribute name="SuckerPassword"></attribute>
 -->

 <!-- The name of the server aspects configuration resource
 <attribute name="ServerAopConfig">aop/jboss-aop-messaging-server.xml</
attribute>
 -->
 <!-- The name of the client aspects configuration resource
```

```
 <attribute name="ClientAopConfig">aop/jboss-aop-messaging-client.xml</
attribute>
 -->

 <depends optional-attribute-
name="PersistenceManager">jboss.messaging:service=PersistenceManager</
depends>

 <depends optional-attribute-
name="JMSUserManager">jboss.messaging:service=JMSUserManager</depends>

 <depends>jboss.messaging:service=Connector,transport=bisocket</depends>
 <depends optional-attribute-name="SecurityStore"
 proxy-
type="org.jboss.jms.server.SecurityStore">jboss.messaging:service=SecurityStore</
depends>
</mbean>
...
```

10.3. Server Attributes

This section discusses the MBean attributes of the ServerPeer MBean.

10.3.1. ServerPeerID

The **`ServerPeerID`** is the unique ID of the server peer that every node you deploy *must* have. This applies whether the different nodes form a cluster, or are only linked via a message bridge. The ID must be a valid integer.

10.3.2. DefaultQueueJNDIContext

The default JNDI context to use when binding queues. Defaults to **`/queue`**.

10.3.3. DefaultTopicJNDIContext

The default JNDI context to use when binding topics.wa Defaults to **`/topic`**.

10.3.4. PostOffice

This is the post office that the ServerPeer uses. You will not normally need to change this attribute. The post office is responsible for routing messages to queues and maintaining the mapping between addresses and queues.

10.3.5. DefaultDLQ

This is the name of the default DLQ (Dead Letter Queue) the server peer will use for destinations. The DLQ can be overridden on a per destination basis - see the destination MBean configuration for more details. A DLQ is a special destination where messages are sent when the server has attempted to deliver them unsuccessfully more than a certain number of times. If the DLQ is not specified at all then the message will be removed after the maximum number of delivery attempts. The maximum

number of delivery attempts can be specified using the attribute DefaultMaxDeliveryAttempts for a global default or individually on a per destination basis.

10.3.6. DefaultMaxDeliveryAttempts

The default for the maximum number of times delivery of a message will be attempted before sending the message to the DLQ, if configured.

The default value is 10.This value can also be overridden on a per destination basis.

10.3.7. DefaultExpiryQueue

This is the name of the default expiry queue the server peer will use for destinations. The expiry can be overridden on a per destination basis - see the destination MBean configuration for more details. An expiry queue is a special destination where messages are sent when they have expired. Message expiry is determined by the value of Message::getJMSExpiration() If the expiry queue is not specified at all then the message will be removed after it is expired.

10.3.8. DefaultRedeliveryDelay

When redelivering a message after failure of previous delivery it is often beneficial to introduce a delay perform redelivery in order to prevent thrashing of delivery-failure, delivery-failure etc.

The default value is 0 which means there will be no delay.

Change this if your application could benefit with a delay before redelivery. This value can also be overridden on a per destination basis.

10.3.9. MessageCounterSamplePeriod

Periodically the server will query each queue to gets its statistics. This is the period.

The default value is 10000 milliseconds.

10.3.10. FailoverStartTimeout

The maximum number of milliseconds the client will wait for failover to start on the server side when a problem is detected.

The default value is 60000 (one minute).

10.3.11. FailoverCompleteTimeout

The maximum number of milliseconds the client will wait for failover to complete on the server side after it has started. The default value is 300000 (five minutes).

10.3.12. DefaultMessageCounterHistoryDayLimit

JBoss Messaging provides a message counter history which shows the number of messages arriving on each queue of a certain number of days. This attribute represents the maxiumum number of days for which to store message counter history. It can be overridden on a per destination basis.

10.3.13. ClusterPullConnectionFactory

The name of the connection factory to use for pulling messages between nodes. If you wish to turn off message sucking between queues altogether, but retain failover, then you can ommit this attribute altogether.

10.3.14. DefaultPreserveOrdering

If true, then strict JMS ordering is preserved in the cluster. See the cluster configurations section for more details. Default is false.

10.3.15. RecoverDeliveriesTimeout

When failover occurs, already delivered messages will be kept aside, waiting for clients to reconnect. In the case that clients never reconnect (e.g. the client is dead) then eventually these messages will timeout and be added back to the queue. The value is in ms. The default is 5 mins.

10.3.16. SuckerPassword

JBoss Messaging internally makes connections between nodes in order to redistribute messages between clustered destinations. These connections are made with the user name of a special reserved user. On this parameter you define the password used as these connections are made. After JBossMessaging 1.4.1.GA you will need to define the Sucker Password on the ServerPeer and on the SecurityMetadataStore.

Warning

This must be specified at install time, or the default password will be used. Any one who then knows the default password will be able to gain access to any destinations on the server. This value MUST be changed at install time.

10.3.17. StrictTCK

Set to true if you want strict JMS TCK semantiocs

10.3.18. Destinations

Returns a list of the destinations (queues and topics) currently deployed.

10.3.19. MessageCounters

JBoss Messaging provides a message counter for each queue.

10.3.20. MessageCountersStatistics

JBoss Messaging provides statistics for each message counter for each queue.

10.3.21. SupportsFailover

Set to false to prevent server side failover occurring in a cluster when a node crashes.

10.3.22. PersistenceManager

This is the persistence manager that the ServerPeer uses. You will not normally need to change this attribute.

10.3.23. JMSUserManager

This is the JMS user manager that the ServerPeer uses. You will not normally need to change this attribute.

10.3.24. SecurityStore

This is the pluggable SecurityStore. If you redefine this SecurityStore, notice it will need to authenticate the MessageSucker user ("JBM.SUCKER") with all the special permissions required by clustering.

10.4. MBean operations of the ServerPeer MBean

10.4.1. DeployQueue

This operation lets you programmatically deploy a queue. There are two overloaded versions of this operation. If the queue already exists but is undeployed it is deployed. Otherwise it is created and deployed. The **`name`** parameter represents the name of the destination to deploy. The **`jndiName`** parameter (optional) represents the full jndi name where to bind the destination. If this is not specified then the destination will be bound in **`<DefaultQueueJNDIContext>/<name>`**.

The first version of this operation deploys the destination with the default paging parameters. The second overloaded version deploys the destination with the specified paging parameters. See the section on configuring destinations for a discussion of what the paging parameters mean.

10.4.2. UndeployQueue

This operation lets you programmatically undeploy a queue. The queue is undeployed but is NOT removed from persistent storage. This operation returns true if the queue was successfull undeployed. otherwise it returns false.

10.4.3. DestroyQueue

This operation lets you programmatically destroy a queue. The queue is undeployed and then all its data is destroyed from the database.

Warning

Be cautious when using this method since it will delete all data for the queue.

This operation returns true if the queue was successfully destroyed. otherwise it returns false.

10.4.4. DeployTopic

This operation lets you programmatically deploy a topic.

There are two overloaded versions of this operation.

If the topic already exists but is undeployed it is deployed. Otherwise it is created and deployed.

The name parameter represents the name of the destination to deploy.

The jndiName parameter (optional) represents the full jndi name where to bind the destination. If this is not specified then the destination will be bound in <DefaultTopicJNDIContext>/<name>.

The first version of this operation deploys the destination with the default paging parameters. The second overloaded version deploys the destination with the specified paging parameters. See the section on configuring destinations for a discussion of what the paging parameters mean.

10.4.5. UndeployTopic

This operation lets you programmatically undeploy a topic. The queue is undeployed but is NOT removed from persistent storage. This operation returns true if the topic was successfully undeployed. otherwise it returns false.

10.4.6. DestroyTopic

This operation lets you programmatically destroy a topic.

The topic is undeployed and then all its data is destroyed from the database.

Warning

Be careful when using this method since it will delete all data for the topic.

This operation returns true if the topic was successfully destroyed. otherwise it returns false.

10.4.7. ListMessageCountersHTML

This operation returns message counters in an easy to display HTML format.

10.4.8. ResetAllMesageCounters

This operation resets all message counters to zero.

10.4.9. ResetAllMesageCounters

This operation resets all message counter histories to zero.

10.4.10. EnableMessageCounters

This operation enables all message counters for all destinations. Message counters are disabled by default.

10.4.11. DisableMessageCounters

This operation disables all message counters for all destinations. Message counters are disabled by default.

10.4.12. RetrievePreparedTransactions

Retrieves a list of the Xids for all transactions currently in a prepared state on the node.

10.4.13. ShowPreparedTransactions

Retrieves a list of the Xids for all transactions currently in a prepared state on the node in an easy to display HTML format.

Use Alternative Databases with JBoss Enterprise Application 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 Application Platform to use alternative databases. We cover the procedures for all officially supported databases on the JBoss Application Server. They include: MySQL 5.0, PostgreSQL 8.1, Oracle 9i and 10g R2, DB2 7.2 and 8, Sybase ASE 12.5, as well as MS SQL 2005.

Please note that in this chapter, we explain how to use alternative databases to support all services in JBoss Enterprise Application 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 Application Platform, you can still use any backend database by setting up the appropriate data source connection.

We assume that you have already installed the external database server, and have it running. You should create an empty database named **`jboss`**, accessible via the username / password pair **`jbossuser / jbosspass`**. The **`jboss`** database is used to store JBoss Enterprise Application Platform internal data -- JBoss Enterprise Application Platform will automatically create tables and data in it.

11.2. Install JDBC Drivers

For the JBoss Application Server and our applications to use the external database, we also need to install the database's JDBC driver. The JDBC driver is a JAR file, which you'll need to copy into your JBoss Enterprise Application Platform's **`<JBoss_Home>/server/all/lib`** directory. Replace **`all`** with the server configuration you are using if needed. This file is loaded when JBoss starts up. So if you have the JBoss Enterprise Application Platform running, you'll need to shut down and restart. The availability of JDBC drivers for different databases are as follows.

- IBM DB2 JDBC drivers can be downloaded from the IBM web site *http://www-306.ibm.com/software/data/db2/java/*.
- Sybase JDBC drivers can be downloaded from the Sybase jConnect product page *http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect*
- MS SQL Server JDBC drivers can be downloaded from the MSDN web site *http://msdn.microsoft.com/data/jdbc/*.

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 the sybase manuals for more options.

Enable JAVA services

To use any java service like JMS, CMP, timers etc. configured with Sybase, java should be enabled on Sybase Adaptive Server. To do this use:

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

Refer to the sybase manuals for more information.

If java is not enabled you might see this exception being thrown when you try to use any of the above services.

```
com.sybase.jdbc2.jdbc.SybSQLException: Cannot run this command because Java
 services are not enabled. A user with System Administrator (SA) role must
 reconfigure the system to enable Java
```

CMP Configuration

To use Container Managed Persistence for user defined Java objects with Sybase Adaptive Server Enterprise the java classes should be installed in the database. The system table 'sysxtypes' contains one row for each extended, Java-SQL datatype. This table is only used for Adaptive Servers enabled for Java. Install java classes using the installjava program.

```
installjava -f <jar-file-name> -S<sybase-server> -U<super-user> -P<super-
pass> -D<db-name>
```

Refer the installjava manual in Sybase for more options.

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 hence requires the java classes to be compiled with the same.

11.2.2. Configuring JDBC DataSources

Rather than configuring the connection manager factory related MBeans discussed in the previous section via a mbean services deployment descriptor, JBoss provides a simplified datasource centric descriptor. This is transformed into the standard **`jboss-service.xml`** MBean services deployment descriptor using a XSL transform applied by the **`org.jboss.deployment.XSLSubDeployer`** included in the **`jboss-jca.sar`** deployment. The simplified configuration descriptor is deployed the same as other deployable components. The descriptor must be named using a **`*-ds.xml`** pattern in order to be recognized by the **`XSLSubDeployer`**.

The schema for the top-level datasource elements of the **`*-ds.xml`** configuration deployment file is shown in *Figure 11.1, "The simplified JCA DataSource configuration descriptor top-level schema elements"*.

Figure 11.1. The simplified JCA DataSource configuration descriptor top-level schema elements

Multiple datasource configurations may be specified in a configuration deployment file. The child elements of the datasources root are:

- **mbean**: Any number mbean elements may be specified to define MBean services that should be included in the **`jboss-service.xml`** descriptor that results from the transformation. This may be used to configure services used by the datasources.
- **no-tx-datasource**: This element is used to specify the (**`org.jboss.resource.connectionmanager`**) **`NoTxConnectionManager`** service configuration. **`NoTxConnectionManager`** is a JCA connection manager with no transaction support. The **`no-tx-datasource`** child element schema is given in *Figure 11.2, "The non-transactional DataSource configuration schema"*.
- **local-tx-datasource**: This element is used to specify the (**`org.jboss.resource.connectionmanager`**) **`LocalTxConnectionManager`** service configuration. **`LocalTxConnectionManager`** implements a **`ConnectionEventListener`** that implements **`XAResource`** to manage transactions through the transaction manager. To ensure that all work in a local transaction occurs over the same **`ManagedConnection`**, it includes a xid to **`ManagedConnection`** map. When a Connection is requested or a transaction started with a connection handle in use, it checks to see if a **`ManagedConnection`** already exists enrolled in the global transaction and uses it if found. Otherwise, a free **`ManagedConnection`** has its **`LocalTransaction`** started and is used. The **`local-tx-datasource`** child element schema is given in *Figure 11.3, "The non-XA DataSource configuration schema"*
- **xa-datasource**: This element is used to specify the (**`org.jboss.resource.connectionmanager`**) **`XATxConnectionManager`** service configuration. **`XATxConnectionManager`** implements a **`ConnectionEventListener`** that obtains the **`XAResource`** to manage transactions through the transaction manager from the adaptor **`ManagedConnection`**. To ensure that all work in a local transaction occurs over the same **`ManagedConnection`**, it includes a xid to **`ManagedConnection`** map. When a **`Connection`** is requested or a transaction started with a connection handle in use, it checks to see if a **`ManagedConnection`** already exists enrolled in the global transaction and uses it if found. Otherwise, a free **`ManagedConnection`** has its **`LocalTransaction`** started and is used. The **`xa-datasource`** child element schema is given in *Figure 11.4, "The XA DataSource configuration schema"*.
- **ha-local-tx-datasource**: This element is identical to **`local-tx-datasource`**, with the addition of the experimental datasource failover capability allowing JBoss to failover to an alternate database in the event of a database failure.
- **ha-xa-datasource**: This element is identical to **`xa-datasource`**, with the addition of the experimental datasource failover capability allowing JBoss to failover to an alternate database in the event of a database failure.

Figure 11.2. The non-transactional DataSource configuration schema

Figure 11.3. The non-XA DataSource configuration schema

Figure 11.4. The XA DataSource configuration schema

Figure 11.5. The schema for the experimental non-XA DataSource with failover

Figure 11.6. The schema for the experimental XA Datasource with failover

Elements that are common to all datasources include:

- **jndi-name**: The JNDI name under which the **`DataSource`** wrapper will be bound. Note that this name is relative to the **`java:/`** context, unless **`use-java-context`** is set to false. **`DataSource`** wrappers are not usable outside of the server VM, so they are normally bound under the **`java:/`**, which isn't shared outside the local VM.
- **use-java-context**: If this is set to false the the datasource will be bound in the global JNDI context rather than the **`java:`** context.
- **user-name**: This element specifies the default username used when creating a new connection. The actual username may be overridden by the application code **`getConnection`** parameters or the connection creation context JAAS Subject.
- **password**: This element specifies the default password used when creating a new connection. The actual password may be overridden by the application code **`getConnection`** parameters or the connection creation context JAAS Subject.
- **application-managed-security**: Specifying this element indicates that connections in the pool should be distinguished by application code supplied parameters, such as from **`getConnection(user, pw)`**.
- **security-domain**: Specifying this element indicates that connections in the pool should be distinguished by JAAS Subject based information. The content of the **`security-domain`** is the name of the JAAS security manager that will handle authentication. This name correlates to the JAAS **`login-config.xml`** descriptor **`application-policy/name`** attribute.
- **security-domain-and-application**: Specifying this element indicates that connections in the pool should be distinguished both by application code supplied parameters and JAAS Subject based information. The content of the **`security-domain`** is the name of the JAAS security manager that will handle authentication. This name correlates to the JAAS **`login-config.xml`** descriptor **`application-policy/name`** attribute.
- **min-pool-size**: This element specifies the minimum number of connections a pool should hold. These pool instances are not created until an initial request for a connection is made. This default to 0.
- **max-pool-size**: This element specifies the maximum number of connections for a pool. No more than the **`max-pool-size`** number of connections will be created in a pool. This defaults to 20.
- **blocking-timeout-millis**: This element specifies the maximum time in milliseconds to block while waiting for a connection before throwing an exception. Note that this blocks only while waiting for

a permit for a connection, and will never throw an exception if creating a new connection takes an inordinately long time. The default is 5000.

- **idle-timeout-minutes**: This element specifies the maximum time in minutes a connection may be idle before being closed. The actual maximum time depends also on the **`IdleRemover`** scan time, which is 1/2 the smallest idle-timeout-minutes of any pool.
- **new-connection-sql**: This is a SQL statement that should be executed when a new connection is created. This can be used to configure a connection with database specific settings not configurable via connection properties.
- **check-valid-connection-sql**: This is a SQL statement that should be run on a connection before it is returned from the pool to test its validity to test for stale pool connections. An example statement could be: **`select count(*) from x`**.
- **exception-sorter-class-name**: This specifies a class that implements the **`org.jboss.resource.adapter.jdbc.ExceptionSorter`** interface to examine database exceptions to determine whether or not the exception indicates a connection error. Current implementations include:
  - org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter
  - org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
  - org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter
  - org.jboss.resource.adapter.jdbc.vendor.InformixExceptionSorte
- **valid-connection-checker-class-name**: This specifies a class that implements the **`org.jboss.resource.adapter.jdbc.ValidConnectionChecker`** interface to provide a **`SQLException isValidConnection(Connection e)`** method that is called with a connection that is to be returned from the pool to test its validity. This overrides the **`check-valid-connection-sql`** when present. The only provided implementation is **`org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker`**.
- **track-statements**: This boolean element specifies whether to check for unclosed statements when a connection is returned to the pool. If true, a warning message is issued for each unclosed statement. If the log4j category **`org.jboss.resource.adapter.jdbc.WrappedConnection`** has trace level enabled, a stack trace of the connection close call is logged as well. This is a debug feature that can be turned off in production.
- **prepared-statement-cache-size**: This element specifies the number of prepared statements per connection in an LRU cache, which is keyed by the SQL query. Setting this to zero disables the cache.
- **depends**: The **`depends`** element specifies the JMX **`ObjectName`** string of a service that the connection manager services depend on. The connection manager service will not be started until the dependent services have been started.
- **type-mapping**: This element declares a default type mapping for this datasource. The type mapping should match a **`type-mapping/name`** element from **`standardjbosscmp-jdbc.xml`**.

Additional common child elements for both **`no-tx-datasource`** and **`local-tx-datasource`** include:

- **connection-url**: This is the JDBC driver connection URL string, for example, **`jdbc:hsqldb:hsql://localhost:1701`**.

- **driver-class**: This is the fully qualified name of the JDBC driver class, for example, **`org.hsqldb.jdbcDriver`**.
- **connection-property**: The **`connection-property`** element allows you to pass in arbitrary connection properties to the **`java.sql.Driver.connect(url, props)`** method. Each **`connection-property`** specifies a string name/value pair with the property name coming from the name attribute and the value coming from the element content.

Elements in common to the **`local-tx-datasource`** and **`xa-datasource`** are:

- **transaction-isolation**: This element specifies the **`java.sql.Connection`** transaction isolation level to use. The constants defined in the Connection interface are the possible element content values and include:
  - TRANSACTION_READ_UNCOMMITTED
  - TRANSACTION_READ_COMMITTED
  - TRANSACTION_REPEATABLE_READ
  - TRANSACTION_SERIALIZABLE
  - TRANSACTION_NONE
- **no-tx-separate-pools**: The presence of this element indicates that two connection pools are required to isolate connections used with JTA transaction from those used without a JTA transaction. The pools are lazily constructed on first use. Its use case is for Oracle (and possibly other vendors) XA implementations that don't like using an XA connection with and without a JTA transaction.

The unique **`xa-datasource`** child elements are:

- **track-connection-by-tx**: Specifying a true value for this element makes the connection manager keep an xid to connection map and only put the connection back in the pool when the transaction completes and all the connection handles are closed or disassociated (by the method calls returning). As a side effect, we never suspend and resume the xid on the connection's **`XAResource`**. This is the same connection tracking behavior used for local transactions.

  The XA spec implies that any connection may be enrolled in any transaction using any xid for that transaction at any time from any thread (suspending other transactions if necessary). The original JCA implementation assumed this and aggressively delisted connections and put them back in the pool as soon as control left the EJB they were used in or handles were closed. Since some other transaction could be using the connection the next time work needed to be done on the original transaction, there is no way to get the original connection back. It turns out that most **`XADataSource`** driver vendors do not support this, and require that all work done under a particular xid go through the same connection.
- **xa-datasource-class**: The fully qualified name of the **`javax.sql.XADataSource`** implementation class, for example, **`com.informix.jdbcx.IfxXADataSource`**.
- **xa-datasource-property**: The **`xa-datasource-property`** element allows for specification of the properties to assign to the **`XADataSource`** implementation class. Each property is identified by the name attribute and the property value is given by the **`xa-datasource-property`** element content. The property is mapped onto the **`XADataSource`** implementation by looking for a JavaBeans style getter method for the property name. If found, the value of the property is set using the JavaBeans setter with the element text translated to the true property type using the **`java.beans.PropertyEditor`** for the type.

- **isSameRM-override-value**: A boolean flag that allows one to override the behavior of the **`javax.transaction.xa.XAResource.isSameRM(XAResource xaRes)`** method behavior on the XA managed connection. If specified, this value is used unconditionally as the **`isSameRM(xaRes)`** return value regardless of the **`xaRes`** parameter.

The failover options common to **`ha-xa-datasource`** and **`ha-local-tx-datasource`** are:

- **url-delimeter**: This element specifies a character used to separate multiple JDBC URLs.
- **url-property**: In the case of XA datasources, this property specifies the name of the **`xa-datasource-property`** that contains the list of JDBC URLs to use.

11.3. Creating a DataSource for the External Database

JBoss Enterprise Application Platform connects to relational databases via datasources. These datasource definitions can be found in the **`<JBoss_Home>/server/all/deploy`** directory. The datasource definitions are deployable just like WAR and EAR files. The datasource files can be recognized by looking for the XML files that end in **`*-ds.xml`**.

Datasource definition files

The datasource definition files for all supported external databases can be found in the **`<JBoss_Home>/docs/examples/jca`** directory.

- MySQL: **`mysql-ds.xml`**
- PostgreSQL: **`postgres-ds.xml`**
- Oracle: **`oracle-ds.xml`**
- DB2: **`db2-ds.xml`**
- Sybase: **`sybase-ds.xml`**
- MS SQL Server: **`mssql-ds.xml`**

The following code snippet shows the **`mysql-ds.xml`** file as an example. All the other **`*-ds.xml`** files are very similiar. You will need to change the **`connection-url`**, as well as the **`user-name`** / **`password`**, to fit your own database server installation.

```
<datasources>
<local-tx-datasource>
<jndi-name>MySqlDS</jndi-name>
<connection-url>jdbc:mysql://localhost:3306/jboss</connection-url>
<driver-class>com.mysql.jdbc.Driver</driver-class>
<user-name>jbossuser</user-name>
<password>jbosspass</password>
<exception-sorter-class-name>
org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
</exception-sorter-class-name>
<!-- should only be used on drivers after 3.22.1 with "ping" support
<valid-connection-checker-class-name>
org.jboss.resource.adapter.jdbc.vendor.MySQLValidConnectionChecker
```

```
</valid-connection-checker-class-name>
-->
<!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->
<!-- sql to call on an existing pooled connection when it is obtained from
 pool -
 MySQLValidConnectionChecker is preferred for newer drivers
<check-valid-connection-sql>some arbitrary sql</check-valid-connection-sql>
 -->

<!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional)
 -->
   <metadata>
 <type-mapping>mySQL</type-mapping>
 </metadata>
 </local-tx-datasource>

</datasources>
```

Once you customized the **`*-ds.xml`** file to connect to your external database, you need to copy it to the **`<JBoss_Home>/server/all/deploy`** directory. The database connection is now available through the JNDI name specified in the **`*-ds.xml`** file.

11.4. Common configuration for DataSources and ConnectionFactorys

11.4.1. General

- <*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 where it is bound. This is prefixed with java by default:
- <*use-java-context*> - set this to false to drop the java: context from the jndi name

11.4.2. XA

<*xa-resource-timeout*> - the number of seconds passed to

```
XAResource.setTranasctionTimeout()
```

when not zero. This feature is available on JBoss Enterprise Application Platform 4.0.3 and above.

11.4.3. Security parameters

JCA Login Modules - are used to inject security configuration into the connection when configured

- *nothing* - uses the user/password specified in **`-ds.xml`** for DataSources or the **`getConnection/createConnection`** method without a **`user/password`** (the default).
- <*application-managed-security*> - uses the user/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, e.g. queue or topic in JMS.

11.4.3.1. Pooling parameters

- <*no-tx-separate-pools*> - whether separate subpools should be created for connections inside and outside JTA transactions (default false).
- <*min-pool-size*> - the minimum number of connections in the pool (default 0 - zero)
- <*max-pool-size*> - the maximum number of connections in the pool (default 20)
- <*blocking-timeout-millis*> - the length of time to wait for a connection to become available when all the connections are checked out (default 5000 == 5 seconds, from 3.2.4 it is 30000 == 30 seconds)
- <*idle-timeout-minutes*> - the number of minutes after which unused connections are closed (default 15 minutes)
- <*track-connection-by-tx*> - whether the connection should be *"locked"* to the transaction, returning it to the pool at the end of the transaction; in pre-JBoss-5.x releases the default value for Local connection factories is true and false for XA; since JBoss-5.x the default value is true for both Local and XA and the element is deprecated.
- <*interleaving/*> - enables interleaving for XA connection factories (this feature was added in JBoss-5.x)
- <*prefill*> - whether to attempt to prefill the connection pool to the minimum number of connections. NOTE: only supporting pools (OnePool) support this feature. A warning can be found in the logs if the pool does not support this. This feature is available in JBoss 4.0.5 and above.
- <*background-validation*> - In JBoss 4.0.5, background connection validation was added to reduce the overall load on the RDBMS system when validating a connection. When using this feature, JBoss will attempt to validate the current connections in the pool as a seperate thread (ConnectionValidator).
- <*background-validation-minutes*> - The interval, in minutes, that the ConnectionValidator will run. NOTE: It is prudent to set this value to something greater or less than the <*idle-timeout-minutes*>
- <*use-fast-fail*> - Whether or not to continue to attempt to acquire a connection from the pool even if the nth attempt has failed. False by default. This is to address performance issues where SQL validation may take significant time and resources to execute.

11.4.3.2. 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.

11.5. Change Database for the JMS Services

The JMS service in the JBoss Enterprise Application 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.dist}/server/${server}/deploy/messaging/${database}-persistence-service.xml`** with the file **`${jboss.dist}/docs/examples/jms/${database}-persistence-service.xml`** depending on your external database. Notice that if you are using the **`default`** server profile, the file path is **`{jboss.dist}/server/default/deploy/messaging/${database}-persistence-service.xml`**.

- MySQL: **`mysql-jdbc2-service.xml`**
- PostgreSQL: **`postgres-jdbc2-service.xml`**
- Oracle: **`oracle-jdbc2-service.xml`**
- DB2: **`db2-jdbc2-service.xml`**
- Sybase: **`sybase-jdbc2-service.xml`**
- MS SQL Server: **`mssql-jdbc2-service.xml`**

What about the hsqldb-jdbc-state-service.xml file?

Despite its name, the **`hsqldb-jdbc-state-service.xml`** file applies to all databases. So, there is no need to use a special **`jdbc-state-service.xml`** for each database.

11.6. Support Foreign Keys in CMP Services

Next, we need to go change the **`<JBoss_Home>/server/all/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 Application Server. This file configures the database connection settings for the EJB2 CMP beans deployed in the JBoss Enterprise Application Platform.

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

11.7. 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 Application 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 **`${jboss.dist}/server/${server}/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
- PostgresSQL 8.2.3: org.hibernate.dialect.PostgreSQLDialect
- PostgresSQL 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.SybaseDialect

11.8. Change Other JBoss Enterprise Application 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.

11.8.1. The Easy Way

The easy way is just to change the JNDI name for the external database to **DefaultDS**. Most JBoss services are hard-wired to use the **DefaultDS** by default. So, by changing the datasource name, we do not need to change the configuration for each service individually.

To change the JNDI name, just open the ***-ds.xml** file for your external database, and change the value of the **jndi-name** property to **DefaultDS**. For instance, in **mysql-ds.xml**, you'd change MySqlDS to DefaultDS and so on. You will need to remove the **<JBoss_Home>/server/all/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-jdbc2-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>
```

11.8.2. 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 Application 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 (e.g., the **MySqlDS** in **mysql-ds.xml** etc.). Below is a complete list of files that contain **DefaultDS**. You can update them all to use the external database on all JBoss services or update some of them to use different combination of datasources for different services.

- **${jboss.dist}/server/${server}/conf/login-config.xml**: This file is used in Java EE container managed security services.
- **${jboss.dist}/server/${server}/conf/standardjbosscmp-jdbc.xml**: This file configures the CMP beans in the EJB container.
- **<JBoss_Home>/server/all/deploy/ejb-deployer.xml**: This file configures the JBoss EJB deployer.
- **${jboss.dist}/server/${server}/deploy/ejb2-timer-service.xml**: This file configures the EJB timer services.
- **${jboss.dist}/server/${server}/deploy/snmp-adaptor.sar/attributes.xml**: This file is used by the SNMP service.
- **${jboss.dist}/server/${server}/deploy/juddi-service.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- **${jboss.dist}/server/${server}/deploy/juddi-service.sar/juddi.war/WEB-INF/jboss-web.xml**: This file configures the UUDI service.
- **<JBoss_Home>/server/all/deploy/juddi-service.sar/juddi.war/WEB-INF/juddi.properties**: This file configures the UUDI service.
- **${jboss.dist}/server/${server}/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- **${jboss.dist}/server/${server}/deploy/messaging/messaging-jboss-beans.xml** and **${jboss.dist}/server/${server}/deploy/messaging/persistence-service.xml**: Those files configure the JMS persistence service as we discussed earlier.

11.9. A Special Note About Oracle DataBases

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

The Oracle database creates tables of the form **schemaname.tablename**. The **TIMERS** and **HILOSEQUENCES** tables needed by JBoss Enterprise Application Platform would not get created on a schema if the table already exists on a different schema. To work around this issue, you need to edit

the **${jboss.dist}/server/${server}/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/all/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>
```

11.10. DataSource configuration

DataSources are defined inside a <datasources> element.

- <no-tx-datasource> - a DataSource that does not take part in JTA transactions using a java.sql.Driver
- <local-tx-datasource> - a DataSource that does not support two phase commit using a java.sql.Driver
- <xa-datasource> - a DataSource that does support two phase commit using a javax.sql.XADataSource

11.11. Parameters specific for java.sql.Driver usage

- <connection-url> - the JDBC driver connection url string
- <driver-class> - the JDBC driver class implementing java.sql.Driver
- <connection-property> - used to configure the connections retrieved from the java.sql.Driver. For example:

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

11.12. Parameters specific for javax.sql.XADataSource usage

- <xa-datasource-class> - This is the class that implements the XADataSource
- <xa-datasource-property> - This contains that properties that are used to configure the XADataSource. For example:

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

- <isSameRM-override-value> - In order to fix issues with Oracle this property should be set to false
- <track-connection-by-tx/> - This property is deprecated and enabled by default in order to correct issues with Oracle
- <no-tx-separate-pools/> - This property will pool Transactional and non-Transactional connections separately and cause your total pool size to be twice the **`max-pool-size`**, as two pools will be created. This is used to fix issues with Oracle.

11.13. Common DataSource parameters

- <jndi-name> - the JNDI name under which the DataSource should be bound.
- <use-java-context> - A boolean indicating if the jndi-name should be prefixed with java: which causes the DataSource to only be accessible from within the jboss server vm. The default is true.
- <user-name> - the user name used when creating the connection (not used when security is configured)
- <password> - the password used when creating the connection (not used when security is configured)
- <transaction-isolation> - the default transaction isolation of the connection (unspecified means use the default provided by the database):
  - 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 the connection schema, etc.
- <check-valid-connection-sql> - an sql statement that is executed before it is checked out from the pool to make sure it is still valid. If the sql fails, the connection is closed and new ones created.
- <valid-connection-checker-class-name> - a class that can check whether a connection is valid using a vendor specific mechanism
- <exception-sorter-class-name> - a class that looks at vendor specific messages to determine whether sql errors are fatal and thus the connection should be destroyed. If none specified, no errors will be treated as fatal.
- <track-statements> - (a) whether to monitor for unclosed Statements and ResultSets and issue warnings when the user forgets to close them (default 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 LRU cache. The default is 0 (zero), meaning no cache.
- <share-prepared-statements> - (b) with prepared statement cache enabled whether two requests in the same transaction should return the same statement (from jboss-4.0.2 - default false).
- <set-tx-query-timeout> - whether to enable query timeout based on the length of time remaining until the transaction times out (default false - NOTE: This was NOT ported to 4.0.x until 4.0.3)
- <query-timeout> - a static configuration of the maximum of seconds before a query times out (since 4.0.3)
- <metadata/typemapping> - a pointer to the type mapping in conf/standardjbosscmp.xml (available from JBoss 4 and above)
- <validate-on-match> - Prior to JBoss 4.0.5, connection validation occurred when the JCA layer attempted to match a managed connection. With the addition of <background-validation> this is no longer required. Specifying <validate-on-match> forces the old behavior. NOTE: this is typically NOT used in conjunction with <background-validation>
- <prefill> - whether to attempt to prefill the connection pool to the minimum number of connections. NOTE: only supporting pools (OnePool) support this feature. A warning can be found in the logs if the pool does not support this. This feature will appear in JBoss 4.0.5.
- <background-validation> - In JBoss 4.0.5, background connection validation as been added to reduce the overall load on the RDBMS system when validating a connection. When using this feature, JBoss will attempt to validate the current connections in the pool is a seperate thread (ConnectionValidator). Default is False.

- <idle-timeout-minutes> - indicates the maximum time a connection may be idle before being closed. Default is 15 minutes.
- <background-validation-minutes> - The interval, in minutes, that the ConnectionValidator will run. Default is 10 minutes. NOTE: It is prudent to set this value to something greater or less than the <idle-timeout-minutes>
- <url-delimiter> - From JBoss5 database failover is part of the main datasource config
- <url-property> - From JBoss5 database failover is part of the main datasource config
- <url-selector-strategy-class-name> - From JBoss5 ONLY database failover is part of the main datasource config
- <stale-connection-checker-class-name> - An implementation of org.jboss.resource.adapter.jdbc.StateConnectionChecker that will decide whether SQLExceptions that notify of bad connections throw org.jboss.resource.adapter.jdbc.StateConnectionException (from JBoss5)

From JBoss Enterprise Application Platform 3.2.6 and above, **`track-statements`** has a new option:

```
<track-statements>nowarn</track-statements
```

This option closes Statements and ResultSets without a warning. It is also the new default value.

The purpose is to workaround 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();
```

Assuming the prepared statements are the same. For some drivers, ps2.executeQuery() will automatically close rs1 so we actually need two real prepared statements behind the scenes. This *should* only be for the auto-commit semantic where re-running the query starts a new transaction automatically. For drivers that follow the spec, you can set it to true to share the same real prepared statement.

11.14. Generic Datasource Sample

```
<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/>
<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/>

<!--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/>

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

</xa-datasource>

</datasources>
```

11.15. Configuring a DataSource for remote usage

From JBoss-4.0.0 and above, there is support for accessing a DataSource from a remote client. The one change that is necessary for the client to be able to lookup the DataSource from JNDI is to specify use-java-context=false as shown here:

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

This results in the DataSource being bound under the JNDI name "GenericDS" instead of the default of "java:/GenericDS" which restricts the lookup to the same VM as the jboss server.

Note

JBoss does not recommend using this feature on a production environment. It requires accessing a connection pool remotely and this is an anti-pattern as connections are not serializable. Besides, transaction propagation is not supported and it could lead to connection leaks if the remote clients are unreliable (i.e crashes, network failure). If you do need to access a datasource remotely, JBoss recommends accessing it via a remote session bean facade.

11.16. Configuring a DataSource to use login modules

Add the security-domain parameter to the *-ds.xml file.

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

Add an application-policy to the login-config.xml file. The authentication section should include the configuration for your login-module. For example, if you want to encrypt the database password, use the SecureIdentityLoginModule login module.

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

In case you plan to fetch the data source connection from a web application, make sure authentication is turned on for the web application. This is in order for the Subject to be populated. If you wish for users to be able to connect anonymously, an additional login module needs to be added to the application-policy, in order to populate the security credentials. Add the UsersRolesLoginModule as the first login module in 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>
```

Pooling

12.1. Strategy

JBossJCA[1] 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, e.g. preconfigured or EJB/Web login subjects	
<security-domain-and-applicaton/>	ByContainerAndApplication	PoolBySubjectAndCRI	A per Subject and connection property combination	

Note

The xml names imply this is just about security. This is misleading.

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

```
(
ConnectionRequestInfo
)
```

12.2. Transaction stickness

You can force the same connection from a (sub-)pool to get reused throughout a transaction with the <track-connection-by-tx/> flag

[1] http://www.jboss.org/wiki/JBossJCA

Note

This is the only supported behaviour for *"local"* transactions. This element is deprecated in JBoss Enterprise Application Platform 5 where transaction stickiness is enabled by default. XA users can explicitly enable interleaving with <interleaving/> element.

12.3. Workaround for Oracle

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

12.4. Pool Access

The pool is designed for concurrent usage.

Upto <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 the <blocking-timeout-seconds/> to use the pool before throwing a *No Managed Connections Available*[2]

12.5. Pool Filling

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

- <min-pool-size/> - When the number of connections falls below this size, new connections are created
- <max-pool-size/> - No more than this number of connections are created
- <prefill/> - Feature Request has been implemented for 4.0.5. Note: the only pooling strategy that supports this feature is OnePool?, or ByNothing? pooling criteria.

The pool filling is done by a separate "Pool Filler" thread rather than blocking application threads.

12.6. Idle Connections

You can configure connections to be closed when they are idle. e.g. If you just had a peak period and now want to reap the unused ones. This is done via the <idle-timeout-minutes/>.

Idle checking is done on a separate "Idle Remover" thread on an LRU (least recently used) basis. The check is done every idle-timeout-minutes divided by 2 for connections unused 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/fresh connections are created.

[2] http://www.jboss.org/wiki/WhatDoesTheMessageNoManagedConnectionsAvailableMean

Note

If you have long running transactions and you use interleaving (i.e. don't 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 however nobody does use it, it would be a candidate for removal before the transaction is committed.

12.7. Dead connections

The JDBC protocol does not provide a natural **`connectionErrorOccured()`** event when a connection is broken. To support dead/broken connection checking there are a number of plugins.

12.7.1. Valid connection checking

The simplest format is to just run a "quick" sql statement:

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

before handing the connection to the application. 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, e.g. Oracle's or MySQL's pingDatabase() via the

```
<valid-connection-checker-class-name/>
```

12.7.2. Errors during SQL queries

You can check if a connection broke during a query by the looking the error codes or messages of the SQLException for FATAL errors rather than normal SQLExceptions. These codes/messages can be vendor specific, e.g.

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

For

```
FATAL
```

errors the connection will be closed.

12.7.3. Changing/Closing/Flushing the pool

- *change or flush()*[3] the pool
- closing/undeploying the pool will do a flush first

12.7.4. Other pooling

Thirdparty Pools[4] - only if you know what you are doing

[4] http://www.jboss.org/wiki/IWantToPluginACustomThirdpartyDataSource

Frequently Asked Questions

13.1. I have problems with Oracle XA?

Check that you:

1. You have pad=true for the XidFactory? in conf/jboss-service.xml.
2. You have <track-connection-by-tx/> in your oracle-xa-ds.xml (not necessarily for JBoss Enterprise Application Platform 5.x where it is enabled by default and the element is deprecated).
3. You have <isSameRM-override-value>false</isSameRM-override-value> in your oracle-xa-ds.xml.
4. You have <no-tx-separate-pools/> in your oracle-xa-ds.xml.
5. That your jbosscmp-jdbc.xml is specifying the same version of oracle as the one you use.
6. That the oracle server you connect to has XA.

Configuring Oracle Database for XA Support 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:

Make sure that Oracle JServer is installed with your database. If it is not installed, you must add it using Oracle Database Configuration Assistant. Choose "Change an Existing DB" and then select the database to which you want to add Oracle JServer. Choose "Next", then "Oracle JServer" and then "Finish". If the settings you have made to your database previously, are not suitable or insufficient for the Oracle JServer installation, the system prompts you to enter additional parameters. The database configuration file (init.ora) is located in **`\oracle\admin\<your_db_name>\pfile`** directory. Execute initxa.sql over your database. By default, this script file is located in **`\oracle\ora81\javavm\install`**. If errors occur during the execution of the file, you must execute the SQL statements from the file manually. Use DBA Studio to create a package and package body named JAVA_XA in SYS schema, and a synonym of this package (also named JAVA_XA) in PUBLIC schema.

A slightly more detailed set of instructions can be found at *Configuring and using XA distributed transactions in WebSphere Studio - Oracle Exception section*[1].

[1] http://www.ibm.com/developerworks/websphere/library/techarticles/0407_woolf/0407_woolf.html?ca=dnp-327#oracle_exception

Part III. Clustering Guide

Introduction and Quick Start

Clustering allows you to run an application on several parallel servers (a.k.a cluster nodes) while providing a single view to application clients. Load is distributed across different servers, and even if one or more of the servers fails, the application is still accessible via the surviving cluster nodes. Clustering is crucial for scalable enterprise applications, as you can improve performance by 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 Application Platform comes with clustering support out of the box, as part of the **`all`** configuration. The **`all`** 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.
- EJB session bean clustering, for both stateful and stateless beans, and for both EJB3 and EJB2.
- A distributed cache for JPA/Hibernate entities.
- A framework for keeping local EJB2 entity caches consistent across a cluster by invalidating cache entries across the cluster when a bean is changed on any node.
- Distributed JMS queues and topics via JBoss Messaging.
- 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*.

In this *Clustering Guide* we aim to provide you with an in depth understanding of how to use JBoss Enterprise Application 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 Application Platform Clustering, and then to provide some background information that will allow you to understand how JBoss Enterprise Application 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 Application Platform Clustering.

14.1. 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 Application Platform Clustering. Most of the areas touched on in this section are covered in much greater detail later in this guide.

14.1.1. Initial Preparation

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

- **Install JBoss Enterprise Application 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.

  If you want to run multiple JBoss Enterprise Application 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 **`all`** configuration. For example, assuming the root of the JBoss distribution was unzipped to **`/var/jboss`**, you would:

```
$ cd /var/jboss/server
$ cp -r all node1
$ cp -r all 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 Application 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 Application Platform clustering does not require the use of UDP multicast; the Enterprise Application Platform can also be reconfigured to use TCP unicast for intra-cluster communication.

- **Determine a unique integer "ServerPeerID" for each node.** This is needed for JBoss Messaging clustering, and can be skipped if you will not be running JBoss Messaging (i.e. you will remove JBM from your server configuration's **`deploy`** directory). JBM requires that each node in a cluster has a unique integer id, known as a "ServerPeerID", that should remain consistent across server restarts. A simple 1, 2, 3, ..., x naming scheme is fine. We'll cover how to use these integer ids in the next section.

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 Application Platform clusters that may be running on your network:

- **Pick a unique name for your cluster.** The default name for a JBoss Enterprise Application Platform cluster is "DefaultPartition". Come up with a different name for each cluster in your environment, e.g. "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 Application 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* [1] for a good discussion on multicast address assignment. We'll cover how to use the address you pick in the next section.

See *Section 23.2.2, "Isolating JGroups Channels"* for more on isolating clusters.

14.1.2. Launching a JBoss Enterprise Application 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 all`** 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`** (see *??? [134]*). 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. Assume the machines are named "node1" and "node2", while node1 has an IP address of **`192.168.0.101`** and node2 has an address of **`192.168.0.102`**. Assume the "ServerPeerID" for node1 is **`1`** and for node2 it's **`2`**. Assume on each machine JBoss is installed in **`/var/jboss`**.

  On node1, to launch JBoss:

```
$ cd /var/jboss/bin
$ ./run.sh -c all -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 all -g DocsPartition -u 239.255.100.100 \
    -b 192.168.0.102 -Djboss.messaging.ServerPeerID=2
```

  The **`-c`** switch says to use the **`all`** 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. The **`-D`** switch sets system property **`jboss.messaging.ServerPeerId`**, from which JBoss Messaging gets its unique id.

- **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, i.e. 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 we need to be sure each Enterprise Application Platform instance has its own work area. So, instead of using the **all** config, we are going to use the **node1** and **node2** configs we copied from **all** in *??? [134]*.

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.messaging.ServerPeerID=1
```

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 -Djboss.messaging.ServerPeerID=2
```

- **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.messaging.ServerPeerID=1 \
    -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.messaging.ServerPeerID=2 \
    -Djboss.service.binding.set=ports-01
```

  This tells the ServiceBindingManager on the first node to use the standard set of ports (e.g. JNDI on 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 (e.g. JNDI on 1199). See the **conf/bootstrap/bindings.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. 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 Application Platform servers up and running.

14.1.3. Web Application Clustering Quick Start

JBoss Enterprise Application 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 20, 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 Application Platform instances (see *Section 15.2.2, "External Load Balancer Architecture"* for more on why that is). JBoss Enterprise Application 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 20.1, "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 Application Platform web session clustering behavior, which is appropriate for most applications. See *Section 20.2, "Configuring HTTP session state replication"* for more advanced configuration options.

14.1.4. EJB Session Bean Clustering Quick Start

JBoss Enterprise Application Platform supports clustered EJB session beans, whereby requests for a bean are balanced across the cluster. For stateful beans a backup copy of bean state is maintained on one or more cluster nodes, providing high availability in case the node handling a particular session fails or is shut down. Clustering of both EJB2 and EJB3 beans is supported.

For EJB3 session beans, simply add the **`org.jboss.ejb3.annotation.Clustered`** annotation to the bean class for your stateful or stateless bean:

```
@javax.ejb.Stateless
@org.jboss.ejb3.annotation.Clustered
public class MyBean implements MySessionInt {

   public void test() {
      // Do something cool
   }
}
```

For EJB2 session beans, or for EJB3 beans where you prefer XML configuration over annotations, simply add a **`clustered`** element to the bean's section in the JBoss-specific deployment descriptor, **`jboss.xml`**:

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>example.StatelessSession</ejb-name>
            <jndi-name>example.StatelessSession</jndi-name>
            <clustered>true</clustered>
        </session>
    </enterprise-beans>
</jboss>
```

See *Chapter 18, Clustered Session EJBs* for more advanced configuration options.

14.1.5. Entity Clustering Quick Start

One of the big improvements in the clustering area in JBoss Enterprise Application Platform 5 is the use of the new Hibernate/JBoss Cache integration for second level entity caching that was introduced in Hibernate 3.3. In the JPA/Hibernate context, a second level cache refers to a cache whose contents are retained beyond the scope of a transaction. A second level cache *may* improve performance by reducing the number of database reads. You should always load test your application with second level caching enabled and disabled to see whether it has a beneficial impact on your particular application.

If you use more than one JBoss Enterprise Application Platform instance to run your JPA/Hibernate application and you use second level caching, you must use a cluster-aware cache. Otherwise a cache on server A will still hold out-of-date data after activity on server B updates some entities.

JBoss Enterprise Application Platform provides a cluster-aware second level cache based on JBoss Cache. To tell JBoss Enterprise Application Platform's standard Hibernate-based JPA provider to enable second level caching with JBoss Cache, configure your **`persistence.xml`** as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
   <persistence-unit name="somename" transaction-type="JTA">
      <jta-data-source>java:/SomeDS</jta-data-source>
      <properties>
         <property name="hibernate.cache.use_second_level_cache"
 value="true"/>
         <property name="hibernate.cache.region.factory_class"

 value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
         <property name="hibernate.cache.region.jbc2.cachefactory"
 value="java:CacheManager"/>
         <!-- Other configuration options ... -->
      </properties>
   </persistence-unit>
</persistence>
```

That tells Hibernate to use the JBoss Cache-based second level cache, but it doesn't tell it what entities to cache. That can be done by adding the **`org.hibernate.annotations.Cache`** annotation to your entity class:

```
package org.example.entities;

import java.io.Serializable;
import javax.persistence.Entity;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Cache (usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Account implements Serializable
```

See *Chapter 19, Clustered Entity EJBs* for more advanced configuration options and details on how to configure the same thing for a non-JPA Hibernate application.

Note

Clustering can add significant overhead to a JPA/Hibernate second level cache, so don't assume that just because second level caching adds a benefit to a non-clustered application that it will be beneficial to a clustered application. Even if clustered second level caching is beneficial overall, caching of more frequently modified entity types may be beneficial in a non-clustered scenario but not in a clustered one. *Always* load test your application.

Clustering Concepts

In the next section, we discuss basic concepts behind JBoss's clustering services. It is helpful that you understand these concepts before reading the rest of the Clustering Guide. Clustering configurations for specific types of applications are covered beginning with the next chapter.

15.1. Cluster Definition

A cluster is a set of nodes that communicate with each other and work toward a common goal. In a JBoss Enterprise Application Platform cluster (also known as a “partition”), a node is an JBoss Enterprise Application 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 all” on two Enterprise Application Platform instances on the same network is enough for them to form a cluster – each Enterprise Application Platform starts a Channel (actually, several) with the same default configuration, so they dynamically discover each other and form a cluster. Nodes can be dynamically added to or removed from clusters at any time, simply by starting or stopping a Channel with a configuration and name that matches the other cluster members. In summary, a JBoss cluster is a set of Enterprise Application Platform server instances each of which is running an identically configured and named JGroups Channel.

On the same Enterprise Application Platform instance, different services can create their own Channel. In a default 5.0.x Enterprise Application Platform, four different services create channels – the web session replication service, the EJB3 SFSB replication service, the EJB3 entity caching service, and a core general purpose clustering service known as HAPartition. In order to differentiate these channels, each must have a unique name, and its configuration must match its peers yet differ from the other channels.

So, if you go to two Enterprise Application Platform 5.0.x instances and execute **`run -c all`**, the channels will discover each other and you'll have a conceptual **`cluster`**. It's easy to think of this as a two node cluster, but it's important to understand that you really have 4 channels, and hence 4 two node clusters.

On the same network, even for the same service, we may have different clusters. *Figure 15.1, “Clusters and server nodes”* shows an example network of JBoss server instances divided into three clusters, with the third cluster only having one node. This sort of topology can be set up simply by configuring the Enterprise Application 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 channels on the same network.

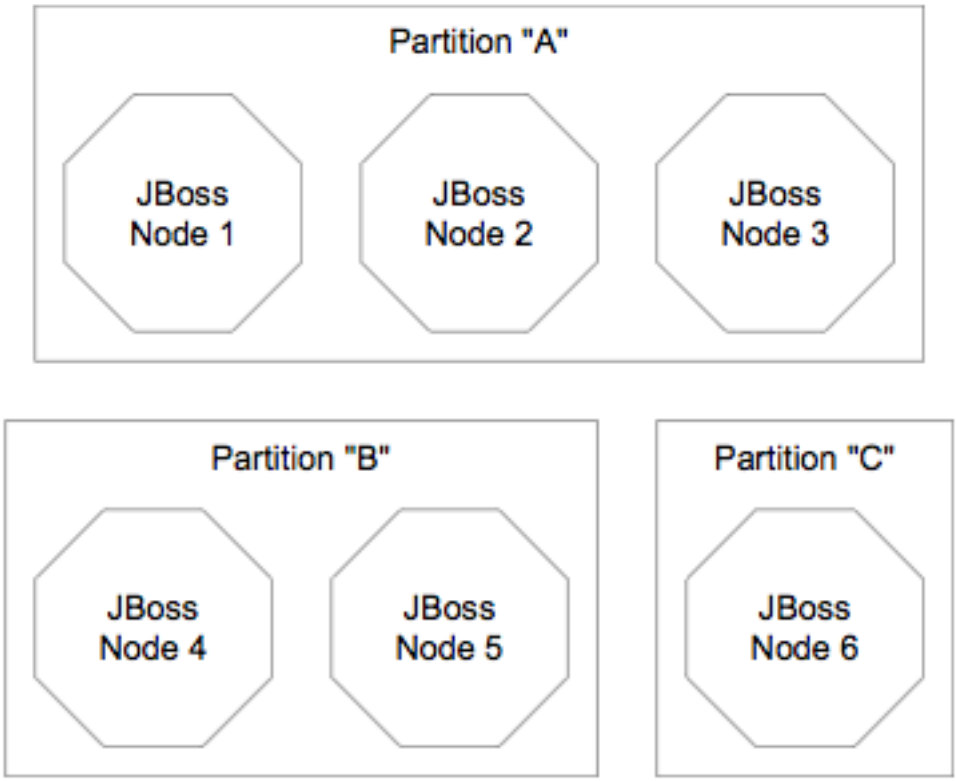


Figure 15.1. Clusters and server nodes

The section on “JGroups Configuration” and on “Isolating JGroups Channels” covers in detail how to configure Channels such that desired peers find each other and unwanted peers do not. As mentioned above, by default JBoss Enterprise Application Platform uses four separate JGroups Channels. These can be divided into two broad categories: the Channel used by the general purpose HAPartition service, and three Channels created by JBoss Cache for special purpose caching and cluster wide state replication.

15.2. Service Architectures

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

15.2.1. Client-side interceptor architecture

Most remote services provided by the JBoss application server, including JNDI, EJB, JMS, RMI and JBoss Remoting, require the client to obtain (e.g., to look up and download) a 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, un-marshall 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 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 15.2, "The client-side interceptor (proxy) architecture for clustering"*.

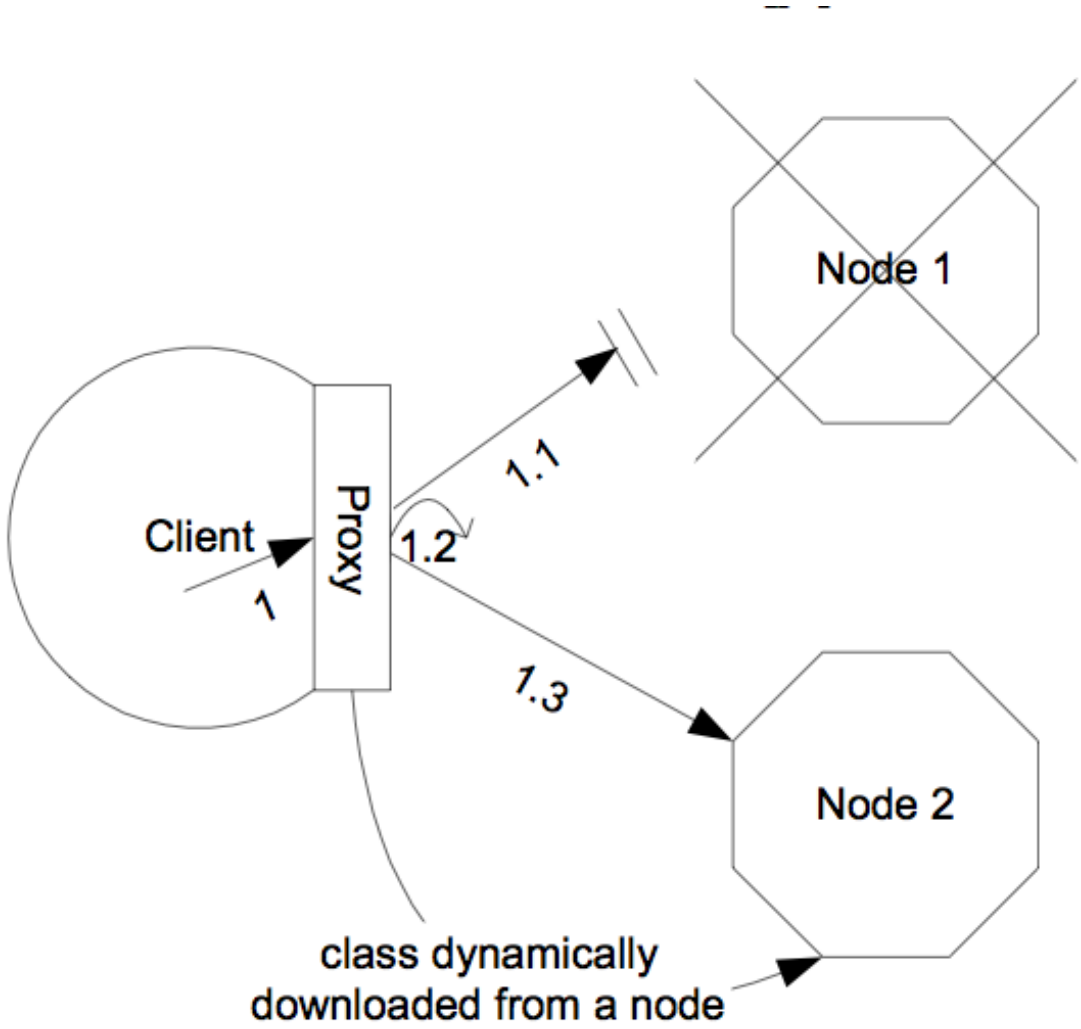


Figure 15.2. The client-side interceptor (proxy) architecture for clustering

Note

Section 18.3, "Stateless Session Bean in EJB 2.x" describes how to enable the client proxy to handle the entire cluster restart.

15.2.2. External Load Balancer Architecture

Other JBoss services, in particular the HTTP-based services, do not require the client to download anything. The client (e.g., a web browser) sends in requests and receives responses directly over the wire using to certain communication protocols (e.g., the HTTP protocol). In this case, an external load balancer is required to process all requests and dispatch them to server nodes in the cluster. The client only needs to know how to contact the load balancer; it has no knowledge of the JBoss Enterprise Application 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 Application Platform instances. It can be implemented either in software or hardware. There are many vendors of hardware load balancers; the mod_jk Apache module is an excellent example of a software load balancer. An external load balancer implements its own mechanism for understanding the cluster configuration and provides its own load balancing and failover policies. The external load balancer clustering architecture is illustrated in *Figure 15.3, “The external load balancer architecture for clustering”*.

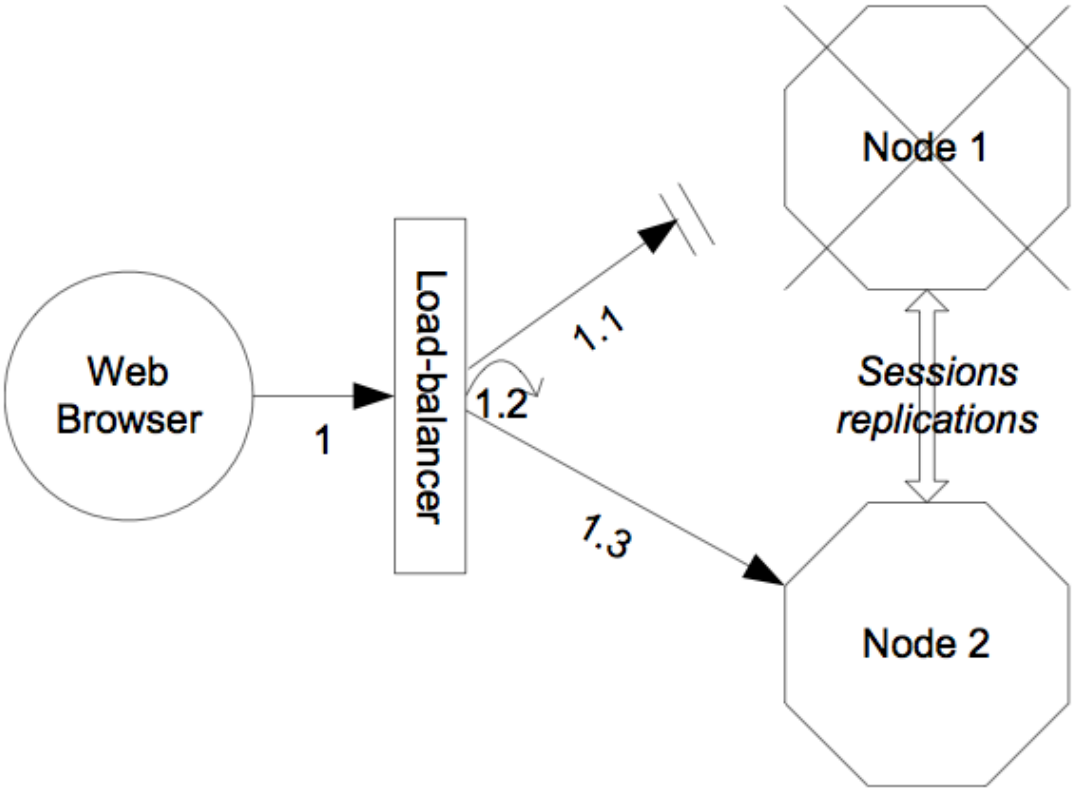


Figure 15.3. The external load balancer architecture for clustering

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.

15.3. Load-Balancing Policies

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

15.3.1. Client-side interceptor architecture

In JBoss 5.0.0, the following load balancing options are available when the client-side interceptor architecture is used. The client-side stub maintains a list of all nodes providing the target service; the job of the load balance policy is to pick a node from this list for each request.

- Round-Robin (**`org.jboss.ha.framework.interfaces.RoundRobin`**): each call is dispatched to a new node, proceeding sequentially through the list of nodes. The first target node is randomly selected from the list.
- Random-Robin (**`org.jboss.ha.framework.interfaces.RandomRobin`**): for each call the target node is randomly selected from the list.
- First Available (**`org.jboss.ha.framework.interfaces.FirstAvailable`**): one of the available target nodes is elected as the main target and is thereafter used for every call; this elected member is randomly chosen from the list of members in the cluster. When the list of target nodes changes (because a node starts or dies), the policy will choose a new target node unless the currently elected node is still available. Each client-side stub elects its own target node independently of the other stubs, so if a particular client downloads two stubs for the same target service (e.g., an EJB), each stub will independently pick its target. This is an example of a policy that provides “session affinity” or “sticky sessions”, since the target node does not change once established.
- First Available Identical All Proxies (**`org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies`**): has the same behaviour as the "First Available" policy but the elected target node is shared by all stubs in the same client-side VM that are associated with the same target service. So if a particular client downloads two stubs for the same target service (e.g. an EJB), each stub will use the same target.

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

15.3.2. External load balancer architecture

As noted above, an external load balancer provides its own load balancing capabilities. What capabilities are supported depends on the provider of the load balancer. The only JBoss requirement is that the load balancer support “session affinity” (a.k.a. “sticky sessions”). With session affinitiy enabled, once the load balancer routes a request from a client to node A and the server initiates a session, all future requests associated with that session must be routed to node A, so long as node A is available.

Clustering Building Blocks

The clustering features in JBoss Enterprise Application Platform are built on top of lower level libraries that provide much of the core functionality. *Figure 16.1, "The JBoss Enterprise Application Platform clustering architecture"* shows the main pieces:

Figure 16.1. The JBoss Enterprise Application 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 Application Platform cluster.

JBoss Cache is a highly flexible clustered transactional caching library. Many Enterprise Application Platform clustering services need to cache some state in memory while 1) ensuring for high availability purposes that a backup copy of that state is available on another node if it can't otherwise be recreated (e.g. the contents of a web session) and 2) ensuring that the data cached on each node in the cluster is consistent. JBoss Cache handles these concerns for most JBoss Enterprise Application 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 Application Platform uses to support fine-grained replication of clustered web session state. See *Section 16.2, "Distributed Caching with JBoss Cache"* for more on how JBoss Enterprise Application 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 16.1, "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. The exception to this is JBoss Messaging's clustering features, which interact with JGroups directly.

16.1. The HAPartition Service

HAPartition is a general purpose service used for a variety of tasks in Enterprise Application Platform clustering. At its core, it is an abstraction built on top of a JGroups Channel that provides support for making/receiving RPC invocations on/from one or more cluster members. HAPartition also supports a distributed registry of which clustering services are running on which cluster members. It provides notifications to interested listeners when the cluster membership changes or the clustered service registry changes. HAPartition forms the core of many of the clustering services we'll be discussing in the rest of this guide, including smart client-side clustered proxies, EJB 2 SFSB replication and entity cache management, farming, HA-JNDI and HA singletons. Custom services can also make use of HAPartition.

The following example shows the **`HAPartition`** MBean definition packaged with the standard JBoss Enterprise Application Platform distribution. So, if you simply start JBoss servers with their default clustering settings on a local network, you would get a default cluster named **`DefaultPartition`** that includes all server instances as its nodes.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
```

```
    name="jboss:service=DefaultPartition">

    <! -- Name of the partition being built -->
    <attribute name="PartitionName">
        ${jboss.partition.name:DefaultPartition}
    </attribute>

    <! -- The address used to determine the node name -->
    <attribute name="NodeAddress">${jboss.bind.address}</attribute>

    <! -- Determine if deadlock detection is enabled -->
    <attribute name="DeadlockDetection">False</attribute>

    <! -- Max time (in ms) to wait for state transfer to complete.
        Increase for large states -->
    <attribute name="StateTransferTimeout">30000</attribute>

    <! -- The JGroups protocol configuration -->
    <attribute name="PartitionConfig">
        ... ...
    </attribute>
</mbean>
```

Here, we omitted the detailed JGroups protocol configuration for this channel. JGroups handles the underlying peer-to-peer communication between nodes, and its configuration is discussed in *Chapter 23, JGroups Services*. The following list shows the available configuration attributes in the **`HAPartition`** MBean.

- **PartitionName** is an optional attribute to specify the name of the cluster. Its default value is **`DefaultPartition`**. Use the **`-g`** (a.k.a. --partition) command line switch to set this value at JBoss startup.
- **NodeAddress** is an optional attribute used to help generate a unique name for this node.
- **DeadlockDetection** is an optional boolean attribute that tells JGroups to run message deadlock detection algorithms with every request. Its default value is **`false`**.
- **StateTransferTimeout** is an optional attribute to specify the timeout for state replication across the cluster (in milliseconds). State replication refers to the process of obtaining initial application state from other already-running cluster members at service startup. Its default value is **`30000`**.
- **PartitionConfig** is an element to specify JGroup configuration options for this cluster (see *Section 23.1, "Configuring a JGroups Channel's Protocol Stack"*).

In order for nodes to form a cluster, they must have the exact same **`PartitionName`** and the **`ParitionConfig`** elements. Changes in either element on some but not all nodes would cause the cluster to split.

You can view the current cluster information by pointing your browser to the JMX console of any JBoss instance in the cluster (i.e., **`http://hostname:8080/jmx-console/`**) and then clicking on the **`jboss:service=HAPartition,partition=DefaultPartition`** MBean (change the MBean name to reflect your partitionr 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.

16.1.1. 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 Application 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 Application Platform clustering internals work.

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

There are two main usages for the DRM in JBoss Enterprise Application Platform:

- **Clustered Smart Proxies**

  Here the keys are the names of the various services that need a clustered smart proxy (see *Section 15.2.1, "Client-side interceptor architecture"*, e.g. the name of a clustered EJB. The value object each node stores in the DRM is known as a "target". It's something a smart proxy's transport layer can use to contact the node (e.g. 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 Availablity Singletons (see the HASingleton chapter). 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 HA singleton 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 doesn't have to 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.

16.1.2. DistributedState Service

The **`DistributedState`** service is a legacy component of the HAPartition service made available to HAPartition users via the **`HAPartition.getDistributedState()`** method. This service provides coordinated management of arbitary application state around the cluster. It is supported for

backwards compatibility reasons, but new applications should not use it; they should use the much more sophisticated JBoss Cache instead.

In JBoss 5 the **`DistributedState`** service actually delegates to an underlying JBoss Cache instance.

16.1.3. 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.jxm.HAServiceMBeanSupport`** class if JMX registration and notification support are desired.

16.2. 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 Application Platform cluster, including:

- Replication of clustered webapp sessions.
- Replication of clustered EJB3 Stateful Session beans.
- 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.

16.2.1. The JBoss Enterprise Application Platform CacheManager Service

Many of the standard clustered services in JBoss Enterprise Application Platform use JBoss Cache to maintain consistent state across the cluster. Different services (e.g. 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 Enterprise Application Platform 4, each of these caches was independently deployed in the **`deploy/`** directory, which had a number of disadvantages:

- Caches that end user applications didn't need were deployed anyway, with each creating an expensive JGroups channel. For example, even if there were no clustered EJB3 SFSBs, a cache to store them was started.
- Caches are internal details of the services that use them. They shouldn't be first-class deployments.
- Services would find their cache via JMX lookups. Using JMX for purposes other exposing management interfaces is just not the JBoss 5 way.

In JBoss 5, the scattered cache deployments have been replaced with a new **CacheManager** service, deployed via the **`JBOSS_HOME/server/all/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 need a cache ask the cache manager for the cache by name; the cache manager creates the cache (if not already created) 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.

16.2.1.1. Standard Cache Configurations

The following standard JBoss Cache configurations ship with JBoss Enterprise Application 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.

- **sfsb-cache**

  Standard cache used for EJB3 SFSB caching.

- **ha-partition**

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

- **mvcc-entity**

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

- **optimistic-entity**

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

- **pessimistic-entity**

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

- **mvcc-entity-repeatable**

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

- **pessimistic-entity-repeatable**

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

- **local-query**

  A config 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 config 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 config 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 config 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 JBC's MVCC locking.

- **optimistic-shared**

  A config 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 JBC's optimistic locking.

- **pessimistic-shared**

  A config 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 JBC's pessimistic locking.

- **mvcc-shared-repeatable**

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

- **pessimistic-shared-repeatable**

  Same as "pessimistic-shared" but uses JBC'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/clears entities from the EntityManager/Hibernate Session and then expects to repeatably re-read them in the same transaction. Otherwise, the Session's internal cache provides a repeatable-read semantic.

16.2.1.2. Cache Configuration Aliases

The CacheManager also supports aliasing of caches; i.e. 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 config names. It's also useful for supporting legacy EJB3 application configurations ported over from Enterprise Application Platform 4.

Aliases can be configured by editing the "CacheManager" bean in the **`jboss-cache-manager-jboss-beans.xml`** file. The following redacted config shows the standard aliases in Enterprise Application Server 5.0.0:

```
<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 config 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>
         <!-- Handle the legacy name for the EJB3 SFSB cache -->
         <entry>
            <key>jboss.cache:service=EJB3SFSBClusteredCache</key>
            <value>sfsb-cache</value>
         </entry>
         <!-- Handle the legacy name for the EJB3 Entity cache -->
         <entry>
            <key>jboss.cache:service=EJB3EntityTreeCache</key>
            <value>mvcc-shared</value>
         </entry>
      </map>
   </property>
```

```
    . . .

</bean>
```

Clustered JNDI Services

JNDI is one of the most important services provided by the application server. The JBoss HA-JNDI (High Availability JNDI) service brings the following features to JNDI:

- Transparent failover of naming operations. If an HA-JNDI naming Context is connected to the HA-JNDI service on a particular JBoss Enterprise Application Platform instance, and that service fails or is shut down, the HA-JNDI client can transparently fail over to another Enterprise Application 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. To illustrate:

- If an EJB is not configured as clustered, looking up the EJB via HA-JNDI does not somehow result in the addition of clustering capabilities (load balancing of EJB calls, transparent failover, state replication) to the EJB.
- If an EJB is configured as clustered, looking up the EJB via regular JNDI instead of HA-JNDI does not somehow result in the removal of the bean proxy's clustering capabilities.

17.1. How it works

The JBoss client-side HA-JNDI naming Context is based on the client-side interceptor architecture (see the Introduction and Quick Start chapter). 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 17.2, “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, all EJB homes and such will not be bound to the cluster-wide JNDI Context, but rather, each home will be bound into the local JNDI.

When a remote client does a lookup through HA-JNDI, HA-JNDI will delegate to the local JNDI 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; rather they are bound in the local JNDI tree. For example, when EJBs are deployed, their proxies are always bound in local JNDI, not HA-JNDI. So, an EJB home lookup done through HA-JNDI will always be delegated to the local JNDI instance.

Note

If different beans (even of the same type, but participating in different clusters) use the same JNDI name, this means that each JNDI server will have a logically different "target" bound 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.

17.2. 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 Application Platform itself or is in another VM.

17.2.1. For clients running inside the application server

If you want to access HA-JNDI from inside the application server, 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/hajndi-jboss-beans.xml`** file (see *Section 17.3, "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 Application 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 Application 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 Application 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 Application Platform's **conf/jndi.properties** file. Doing either will almost certainly break things for your application and quite possibly across the application server. If you want to externalize your client configuration, one approach is to deploy a properties file not named **jndi.properties**, and then programatically create a **Properties** object that loads that file's contents.

17.2.1.1. Accessing HA-JNDI Resources from EJBs and WARs -- Environment Naming Context

If your HA-JNDI client is an EJB or 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. Following is an example of doing this for a JMS connection factory and queue (the most common use case for this kind of thing).

Within the bean definition in the ejb-jar.xml or in the war's web.xml you will need to define two resource-ref mappings, one for the connection factory and one for the destination.

```
<resource-ref>
 <res-ref-name>jms/ConnectionFactory</res-ref-name>
 <res-type>javax.jms.QueueConnectionFactory</res-type>
 <res-auth>Container</res-auth>
</resource-ref>

<resource-ref>
 <res-ref-name>jms/Queue</res-ref-name>
 <res-type>javax.jms.Queue</res-type>
 <res-auth>Container</res-auth>
</resource-ref>
```

Using these examples the bean performing the lookup can obtain the connection factory by looking up 'java:comp/env/jms/ConnectionFactory' and can obtain the queue by looking up 'java:comp/env/jms/Queue'.

Within the JBoss-specific deployment descriptor (jboss.xml for EJBs, jboss-web.xml for a WAR) these references need to be mapped to a URL that makes use of HA-JNDI.

```
<resource-ref>
 <res-ref-name>jms/ConnectionFactory</res-ref-name>
 <jndi-name>jnp://${jboss.bind.address}:1100/ConnectionFactory</jndi-name>
```

```
</resource-ref>

<resource-ref>
 <res-ref-name>jms/Queue</res-ref-name>
 <jndi-name>jnp://${jboss.bind.address}:1100/queue/A</jndi-name>
 </resource-ref>
```

The URL should be the URL to the HA-JNDI server running on the same node as the bean; if the bean is available the local HA-JNDI server should also be available. The lookup will then automatically query all of the nodes in the cluster to identify which node has the JMS resources available.

The **`${jboss.bind.address}`** syntax used above tells JBoss to use the value of the **`jboss.bind.address`** system property when determining the URL. That system property is itself set to whatever value you assign to the **`-b`** command line option when you start JBoss Enterprise Application Platform.

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

The JBoss application server'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 application server because of the possibility of its being found on the classpath when it shouldn't and thus disrupting the internal operation of the server. For example, if an EJB deployment included a jndi.properties configured for HA-JNDI, when the server binds the EJB proxies into JNDI it will likely bind them into the replicated HA-JNDI tree and not into the local JNDI tree where they belong.

17.2.1.3. 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. Please visit the following link for an example: *Problem with removing a Node from Cluster*[1]

17.2.2. For clients running outside the application server

The JNDI client needs to be aware of the HA-JNDI cluster. You can pass a list of JNDI servers (i.e., the nodes in the HA-JNDI cluster) to the **`java.naming.provider.url`** JNDI setting in the **`jndi.properties`** file. Each server node is identified by its IP address and the JNDI port number. The server nodes are separated by commas (see *Section 17.3, "JBoss configuration"* for how to configure the servers and ports).

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

When initialising, the JNP client code will try to get in touch with each server node from the list, one after the other, stopping as soon as one server has been reached. It will then download the HA-JNDI stub from this node.

[1] http://www.jboss.com/index.html?module=bb&op=viewtopic&t=104715

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 17.3, "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`**: Provides a list of IP addresses and port numbers for HA-JNDI provider nodes in the cluster. The client tries those providers one by one and uses the first one that responds.
- **`jnp.disableDiscovery`**: When set to **`true`**, this property disables the automatic discovery feature. Default is **`false`**.
- **`jnp.partitionName`**: In an environment where multiple HA-JNDI services bound to distinct clusters (a.k.a. partitions), are running, this property allows you to ensure that your client only accepts automatic-discovery responses from servers in the desired partition. If you do not use the automatic discovery feature (i.e. jnp.disableDiscovery is true), this property is not used. By default, this property is not set and the automatic discovery selects the first HA-JNDI server that responds, regardless of the cluster partition name.
- **`jnp.discoveryTimeout`**: Determines how many milliseconds the context will wait for a response to its automatic discovery packet. Default is 5000 ms.
- **`jnp.discoveryGroup`**: Determines which multicast group address is used for the automatic discovery. Default is 230.0.0.4. Must match the value of the AutoDiscoveryAddress configured on the server side HA-JNDI service. Note that the server side HA-JNDI service by default listens on the

address specified via the **-u** startup switch, so if **-u** is used on the server side (as is recommended), jnp.discoveryGroup will need to be configured on the client side.

- **jnp.discoveryPort**: Determines which multicast port is used for the automatic discovery. Default is 1102. Must match the value of the AutoDiscoveryPort configured on the server side HA-JNDI service.
- **jnp.discoveryTTL**: specifies the TTL (time-to-live) f or autodiscovery IP multicast packets. This value represents the number of network hops a multicast packet can be allowed to propagate before networking equipment should drop the packet. Despite its name, it does not represent a unit of time.

17.3. JBoss configuration

The **hajndi-jboss-beans.xml** file in the **JBOSS_HOME/server/all/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** accepts the core clustering service used manage HA-JNDI's clustered proxies and to make the group RPCs that find locally bound objects on other nodes. See *Section 16.1, "The HAPartition Service"* for more.
- **distributedTreeManager** accepts a handler for the replicated tree. The standard handler uses JBoss Cache to manage the replicated tree. The JBoss Cache instance is retrieved using the injected **`HAPartitionCacheHandler`** bean. See *Section 16.1, "The HAPartition Service"* for more details.
- **localNamingInstance** accepts the reference to the local JNDI service.
- **lookupPool** accepts the thread pool used to provide threads to handle the bootstrap and auto discovery lookups.

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

- **bindAddress** specifies the address to which the HA-JNDI server will bind to listen for naming proxy download requests from JNP clients. The default value is the value of the **`jboss.bind.address`** system property, or **`localhost`** if that property is not set. The **`jboss.bind.address`** system property is set if the **`-b`** command line switch is used when JBoss is started.
- **port** specifies the port to which the HA-JNDI server will bind to listen for naming proxy download requests from JNP clients. The value is obtained from the ServiceBindingManager bean configured in **`conf/bootstrap/bindings.xml`**. The default value is **`1100`**.
- **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 will bind to listen for RMI requests (e.g. for JNDI lookups) from naming proxies. The default value is the value of the **`jboss.bind.address`** system property, or **`localhost`** if that property is not set. The **`jboss.bind.address`** system property is set if the **`-b`** command line switch is used when JBoss is started.
- **rmiPort** specifies the port to which the server will bind 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** is a boolean flag that disables configuration of the auto discovery multicast listener. The default is **`false`**.
- **autoDiscoveryAddress** specifies the multicast address to listen to for JNDI automatic discovery. The default value is the value of the **`jboss.partition.udpGroup`** system property, or 230.0.0.4 if that is not set. The **`jboss.partition.udpGroup`** system property is set if the **`-u`** command line switch is used when JBoss is started.

- **autoDiscoveryGroup** 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 should listen for auto-discovery request packets. If this attribute is not specified and a **`bindAddress`** is specified, the **`bindAddress`** will be used.
- **autoDiscoveryTTL** specifies the TTL (time-to-live) for autodiscovery IP multicast packets. This value represents the number of network hops a multicast packet can be allowed to propagate before networking equipment should drop the packet. Despite its name, it does not represent a unit of time.
- **loadBalancePolicy** specifies the class name of the LoadBalancePolicyimplementation that should be included in the client proxy. See the Introduction and Quick Start chapter for details.
- **clientSocketFactory** is an optional attribute that specifies the fully qualified classname of the **`java.rmi.server.RMIClientSocketFactory`** that should be used to create client sockets. The default is **`null`**.
- **serverSocketFactory** is an optional attribute that specifies the fully qualified classname of the **`java.rmi.server.RMIServerSocketFactory`** that should be used to create server sockets. The default is **`null`**.

17.3.1. 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

 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
         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>
```

Clustered Session EJBs

Session EJBs provide remote invocation services. They are clustered based on the client-side interceptor architecture. The client application for a clustered session bean is 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. Now, let's check out how to configure clustered session beans in EJB 3.0 and EJB 2.x server applications respectively.

18.1. Stateless Session Bean in EJB 3.0

Clustering stateless session beans is most probably the easiest case: as no state is involved, calls can be load-balanced on any participating node (i.e. any node that has this specific bean deployed) of the cluster.

To cluster a stateless session bean in EJB 3.0, all you need to do is to annotate the bean class with the **`@Clustered`** annotation. You can pass in the load balance policy and cluster partition as parameters to the annotation. The default load balance policy is **`org.jboss.ha.framework.interfaces.RandomRobin`** and the default cluster is **`DefaultPartition`**. Below is the definition of the **`@Cluster`** annotation.

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

Here is an example of a clustered EJB 3.0 stateless session bean implementation.

```
@Stateless
@Clustered
public class MyBean implements MySessionInt
{
   public void test()
   {
      // Do something cool
   }
}
```

The **`@Clustered`** annotation can also be omitted and the clustering configuration applied in jboss.xml:

```
<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>NonAnnotationStateful</ejb-name>
         <clustered>true</clustered>
         <cluster-config>
            <partition-name>FooPartition</partition-name>
```

```
            <load-balance-
policy>org.jboss.ha.framework.interfaces.RandomRobin</load-balance-policy>
         </cluster-config>
      </session>
   </enterprise-beans>
</jboss>
```

Note

The **`<clustered>true</clustered>`** element is really just an alias for the **`<configuration-name>Clustered Stateless SessionBean</configuration-name>`** element in the conf/standard-jboss.xml file.

In the bean configuration, only the <clustered> element is mandatory. It indicates that the bean needs to support clustering features. The <cluster-config> element is optional and the default values of its attributes are indicated in the sample configuration above. Below is a description of the attributes in the <cluster-config> element.

- **partition-name** specifies the name of the cluster the bean participates in. The default value is **`DefaultPartition`**. The default partition name can also be set system-wide using the **`jboss.partition.name`** system property.
- **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. You can also implement your own load-balance policy class or use the class **`FirstAvailable`** that persists to use the first node available that it meets until it fails.

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

18.2.1. The EJB application configuration

To cluster stateful session beans in EJB 3.0, you need to tag the bean implementation class with the **`@Cluster`** annotation, just as we did with the EJB 3.0 stateless session bean earlier. The @org.jboss.ejb3.annotation.CacheConfig annotation can also be applied to the bean to specify caching behavior. Below is the definition of the @CacheConfig annotation:

```
public @interface CacheConfig
{
   String name() default "";
   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 16.2.1, "The JBoss Enterprise Application Platform CacheManager Service"*. By default, the **sfsb-cache** configuration will be used.
- **maxSize** specifies the maximum number of beans that can cached before the cache should start passivating beans, using an LRU algorithm.
- **idleTimeoutSeconds** specifies the max period of time a bean can go unused before the cache should passivate it (irregardless of whether maxSize beans are cached.)
- **removalTimeoutSeconds** specifies the max period of time a bean can go unused before the cache should remove it altogether.
- **replicationIsPassivation** specifies whether the cache should consider a replication as being equivalent to a passivation, and invoke any @PrePassivate and @PostActivate callbacks on the bean. By default true, since replication involves serializing the bean, and preparing for and recovering from serialization is a common reason for implementing the callback methods.

Here is an example of a clustered EJB 3.0 stateful session bean implementation.

```
@Stateful
@Clustered
@CacheConfig(maxSize=5000, removalTimeoutSeconds=18000)
public class MyBean implements MySessionInt
{
   private int state = 0;

   public void increment()
   {
      System.out.println("counter: " + (state++));
   }
}
```

As with stateless beans, the @Clustered annotation can alternatively be omitted and the clustering configuration instead applied to jboss.xml:

```
<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>
```

18.2.2. 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 if 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 the replication would not occur.

18.2.3. CacheManager service configuration

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The **`CacheManager`** service, described in *Section 16.2.1, "The JBoss Enterprise Application 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="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>
     <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 CacheLoaderConfiguration, be sure that you do not use a shared store, e.g. a single schema in a shared database. Each node in the cluster must have its own persistent store, otherwise as nodes independently passivate and activate clustered beans, they will corrupt each other's data.

Buddy Replication

Using buddy replication, state is replicated to a configurable number of backup servers in the cluster (aka 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.

18.3. Stateless Session Bean in EJB 2.x

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.

18.4. Stateful Session Bean in EJB 2.x

Clustering stateful session beans is more complex than clustering their stateless counterparts since JBoss needs to manage the state information. The state of all stateful session beans are replicated and synchronized across the cluster each time the state of a bean changes. The JBoss Enterprise Application Platform uses the **`HASessionState`** MBean to manage distributed session states for clustered EJB 2.x stateful session beans. In this section, we cover both the session bean configuration and the **`HASessionState`** MBean configuration.

18.4.1. 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-nam>
        <home-load-balance-
policy>org.jboss.ha.framework.interfaces.RoundRobin</home-load-balance-
policy>
        <bean-load-balance-
policy>org.jboss.ha.framework.interfaces.FirstAvailable</bean-load-balance-
policy>
        <session-state-manager-jndi-name>/HASessionState/Default</session-
state-manager-jndi-name>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

In the bean configuration, only the **<clustered>** tag is mandatory to indicate that the bean works in a cluster. The **<cluster-config>** element is optional and its default attribute values are indicated in the sample configuration above.

The **<session-state-manager-jndi-name>** tag is used to give the JNDI name of the **HASessionState** service to be used by this bean.

The description of the remaining tags is identical to the one for stateless session bean. Actions on the clustered stateful session bean's home interface are by default load-balanced, round-robin. Once the bean's remote stub is available to the client, calls will not be load-balanced round-robin any more and will stay "sticky" to the first node in the list.

18.4.2. 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. This feature is available on JBoss Enterprise Application Platform 3.0.1+ only.

18.4.3. The HASessionState service configuration

The **HASessionState** service MBean is defined in the `all/deploy/cluster-service.xml` file.

```
<mbean code="org.jboss.ha.hasessionstate.server.HASessionStateService"
 name="jboss:service=HASessionState">
  <depends>jboss:service=Naming</depends>
  <!--
    We now inject the partition into the HAJNDI service instead
    of requiring that the partition name be passed
  -->
  <depends optional-attribute-name="ClusterPartition" proxy-
type="attribute">
    jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
  <!-- JNDI name under which the service is bound -->
  <attribute name="JndiName">/HASessionState/Default</attribute>
  <!--
    Max delay before cleaning unreclaimed state.
    Defaults to 30*60*1000 => 30 minutes
  -->
  <attribute name="BeanCleaningDelay">0</attribute>
</mbean>
```

The configuration attributes in the **HASessionState** MBean are listed below.

- **ClusterPartition** is a required attribute to inject the HAPartition service that HA-JNDI uses for intra-cluster communication.
- **JndiName** is an optional attribute to specify the JNDI name under which this **`HASessionState`** service is bound. The default value is **`/HAPartition/Default`**.
- **BeanCleaningDelay** is an optional attribute to specify the number of miliseconds after which the **`HASessionState`** service can clean a state that has not been modified. If a node, owning a bean, crashes, its brother node will take ownership of this bean. Nevertheless, the container cache of the brother node will not know about it (because it has never seen it before) and will never delete according to the cleaning settings of the bean. That is why the **`HASessionState`** service needs to do this cleanup sometimes. The default value is **`30*60*1000`** milliseconds (i.e., 30 minutes).

18.4.4. Handling Cluster Restart

We have covered the HA smart client architecture in the section called “Client-side interceptor architecture”. The default HA smart proxy client can only failover as long as one node in the cluster exists. If there is a complete cluster shutdown, the proxy becomes orphaned and loses knowledge of the available nodes in the cluster. There is no way for the proxy to recover from this. The proxy needs to look up a fresh set of targets out of JNDI/HAJNDI when the nodes are restarted.

The 3.2.7+/4.0.2+ releases contain a RetryInterceptor that can be added to the proxy client side interceptor stack to allow for a transparent recovery from such a restart failure. To enable it for an EJB, setup an invoker-proxy-binding that includes the RetryInterceptor. Below is an example jboss.xml configuration.

```
<jboss>
  <session>
    <ejb-name>nextgen_RetryInterceptorStatelessSession</ejb-name>
    <invoker-bindings>
      <invoker>
        <invoker-proxy-binding-name>clustered-retry-stateless-rmi-invoker</
invoker-proxy-binding-name>
        <jndi-name>nextgen_RetryInterceptorStatelessSession</jndi-name>
      </invoker>
    </invoker-bindings>
    <clustered>true</clustered>
  </session>
  <invoker-proxy-binding>
    <name>clustered-retry-stateless-rmi-invoker</name>
    <invoker-mbean>jboss:service=invoker,type=jrmpha</invoker-mbean>
    <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
    <proxy-factory-config>
      <client-interceptors>
        <home>
          <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
          <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>
        <bean>
          <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</
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>
        </bean>
      </client-interceptors>
    </proxy-factory-config>
  </invoker-proxy-binding>
</jboss>
```

18.4.5. 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. But, for that lookup to succeed, the InitialContext needs to be configured properly to find your naming server. The RetryInterceptor will go through the following steps in attempting to determine the proper naming environment properties:

1. It will check its own static retryEnv field. This field can be set by client code via a call to RetryInterceptor.setRetryEnv(Properties). This approach to configuration has two downsides: first, it reduces portability by introducing JBoss-specific calls to the client code; and second, since a static field is used only a single configuration per JVM is possible.

2. If the retryEnv field is null, it will check for any environment properties bound to a ThreadLocal by the org.jboss.naming.NamingContextFactory class. To use this class as your naming context factory, in your jndi.properties set property java.naming.factory.initial=org.jboss.naming.NamingContextFactory. The advantage of this approach is use of org.jboss.naming.NamingContextFactory is simply a configuration option in your 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 the section on “ClusteredJNDI Services” for more on multicast discovery of HA-JNDI.

18.4.6. 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 doesn't make the RetryInterceptor part of its default client interceptor stacks for clustered EJBs.

In the 4.0.4.RC1 release, a new flavor of retry interceptor was introduced, the org.jboss.proxy.ejb.SingleRetryInterceptor. This version works like the RetryInterceptor, but only makes a single attempt to re-lookup the HA proxy in JNDI. If this attempt fails, the EJB call will fail just

as if no retry interceptor was used. Beginning with 4.0.4.CR2, the SingleRetryInterceptor is part of the default client interceptor stacks for clustered EJBs.

The downside of the SingleRetryInterceptor is that if the retry attempt is made during a portion of a cluster restart where no servers are available, the retry will fail and no further attempts will be made.

Clustered Entity EJBs

In a JBoss Enterprise Application 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.

19.1. 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.

19.1.1. 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`**.

e.g.

```
<?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.

Property	Default	Description
hibernate.cache.region.jbc2.cfg.shared	shared-jbc.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.

Table 19.1. Additional properties for SharedJBossCacheRegionFactory

  org.hibernate.cache.jbc2.JndiSharedJBossCacheRegionFactory
  : Uses a single JBoss Cache configuration, from an existing CacheManager bound to JNDI, for all cache regions.

Property	Default	Description
hibernate.cache.region.jbc2.cfg.shared	*Required*	JNDI name to which the shared **`Cache`** instance is bound.

Table 19.2. Additional properties for JndiSharedJBossCacheRegionFactory

A multiplexed region factory uses separate Cache instances, using optimized configurations for each cache region.

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.

Table 19.3. Common properties for multiplexed region factory implementations

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.

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	jgroups-stacks.xml,org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	The classpath or filesystem resource containing the JGroups protocol stack configurations.

Table 19.4. Additional properties for MultiplexedJBossCacheRegionFactory

org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory
: Uses separate JBoss Cache configurations, from a JNDI-bound CacheManager, see *Section 16.2.1, "The JBoss Enterprise Application Platform CacheManager Service"*, per cache region.

Property	Default	Description
hibernate.cache.region.jbc2.cachefactory	*Required*	JNDI name to which the **`CacheManager`** instance is bound.

Table 19.5. Additional properties for JndiMultiplexedJBossCacheRegionFactory

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.

19.1.2. 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 could evict oldest objects (or least used objects, depending on configuration) to make room for new objects. Assuming the region_prefix specified in **`persistence.xml`** was **`myprefix`**, the default name of the cache region for the **`com.mycompany.entities.Account`** entity bean 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.
                   Set this to a value greater than your max expected
 transaction length. -->
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
      </property>
      <property name="evictionRegionConfigs">
        <list>
          <bean class="org.jboss.cache.config.EvictionRegionConfig">
            <property name="regionName">/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 in the **/_default** region as defined above. The @Cache annotation exposes an optional attribute “region” that lets you specify the cache region where an entity is to be stored, rather than having it be automatically be created from the fully-qualified class name of the entity class.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = ”Account”)
public class Account implements Serializable
{
  // ... ...
}
```

The eviction configuration would then become:

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
 name="jboss.cache:service=EJB3EntityTreeCache">
     ... ...
    <attribute name="EvictionPolicyConfig">
      <config>
        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <region name="/_default_">
          <attribute name="maxNodes">5000</attribute>
          <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
        <!-- Separate eviction rules for Account entities -->
        <region name="/myprefix/Account">
          <attribute name="maxNodes">10000</attribute>
          <attribute name="timeToLiveSeconds">5000</attribute>
        </region>
         ... ...
      </config>
    </attribute>
  </mbean>
</server>
```

19.1.3. Query result caching

The EJB3 Query API also provides means for you to save in the second-level cache the results (i.e., collections of primary keys of entity beans, or collections of scalar values) of specified queries. Here we show a simple example of annotating a bean with a named query, also providing the Hibernate-specific hints that tells Hibernate to cache the query.

First, in persistence.xml you need to tell Hibernate to enable query caching:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

Next, you create a named query associated with an entity, and tell Hibernate you want to cache the results of that query:

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = ”Account”)
@NamedQueries(
{
  @NamedQuery(
    name = "account.bybranch",
    query = "select acct from Account as acct where acct.branch = ?1",
    hints = { @QueryHint(name = "org.hibernate.cacheable", value =
 "true") }
  )
```

```
})
public class Account implements Serializable
{
  // ... ...
}
```

The @NamedQueries, @NamedQuery and @QueryHint annotations are all in the javax.persistence package. See the Hibernate and EJB3 documentation for more on how to use EJB3 queries and on how to instruct EJB3 to cache queries.

By default, Hibernate stores query results in JBoss Cache in a region named {region_prefix}/org/hibernate/cache/StandardQueryCache. Based on this, you can set up separate eviction handling for your query results. So, if the region prefix were set to myprefix in **`persistence.xml`**, you could, for example, create this sort of eviction handling:

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
 name="jboss.cache:service=EJB3EntityTreeCache">
     ... ...
    <attribute name="EvictionPolicyConfig">
      <config>
        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <region name="/_default_">
          <attribute name="maxNodes">5000</attribute>
          <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
        <!-- Separate eviction rules for Account entities -->
        <region name="/myprefix/Account">
          <attribute name="maxNodes">10000</attribute>
          <attribute name="timeToLiveSeconds">5000</attribute>
        </region>
        <!-- Cache queries for 10 minutes -->
        <region name="/myprefix/org/hibernate/cache/StandardQueryCache">
          <attribute name="maxNodes">100</attribute>
          <attribute name="timeToLiveSeconds">600</attribute>
        </region>
          ... ...
      </config>
    </attribute>
  </mbean>
</server>
```

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 ofthe default /org/hibernate/cache/StandardQueryCache. For example:

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = ”Account”)
@NamedQueries(
```

```
{
  @NamedQuery(
    name = "account.bybranch",
    query = "select acct from Account as acct where acct.branch = ?1",
    hints =
    {
      @QueryHint(name = "org.hibernate.cacheable", value = "true"),
      @QueryHint(name = ”org.hibernate.cacheRegion, value = ”Queries”)
    }
  )
})
public class Account implements Serializable
{
  // ... ...
}
```

The related eviction configuration:

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
 name="jboss.cache:service=EJB3EntityTreeCache">
     ... ...
    <attribute name="EvictionPolicyConfig">
      <config>
        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <region name="/_default_">
          <attribute name="maxNodes">5000</attribute>
          <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
        <!-- Separate eviction rules for Account entities -->
        <region name="/myprefix/Account">
          <attribute name="maxNodes">10000</attribute>
          <attribute name="timeToLiveSeconds">5000</attribute>
        </region>
        <!-- Cache queries for 10 minutes -->
        <region name="/myprefix/Queries">
          <attribute name="maxNodes">100</attribute>
          <attribute name="timeToLiveSeconds">600</attribute>
        </region>
         ... ...
      </config>
    </attribute>
  </mbean>
</server>
```

19.2. Entity Bean in EJB 2.x

First of all, it is worth noting that clustering 2.x entity beans is a bad thing to do. Its 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 sepecial case situation of read-only, or one read-write node with read-only nodes synched with the cache invalidation services.

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. (There are some design patterns that allow you to use Commit Option "A" with read-mostly beans. You can also take a look at the Seppuku pattern *http://dima.dhs.org/misc/readOnlyUpdates.html*. JBoss may incorporate this pattern into later versions.)

Note

If you are using Bean Managed Persistence (BMP), you are going to have to implement synchronization on your own. The MVCSoft CMP 2.0 persistence engine (see *http://www.jboss.org/jbossgroup/partners.jsp*) provides different kinds of optimistic locking strategies that can work in a JBoss cluster.

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 of incoming invocations

State replication is directly handled by JBoss. When you run JBoss in the **`all`** configuration, session state replication is enabled by default. Just configure your web application as **`<distributable>`** in its **`web.xml`** (see below), deploy it, and its session state is automtically 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. In other situations, losing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

20.1. 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.

20.1.1. Download the software

First of all, make sure that you have Apache installed. You can download Apache directly from Apache web site at **`http://httpd.apache.org/`**. Its installation is pretty straightforward and requires no specific configuration. As several versions of Apache exist, we advise you to use version 2.0.x. We will consider, for the next sections, that you have installed Apache in the **`APACHE_HOME`** directory.

Next, download mod_jk binaries. Several versions of mod_jk exist as well. We strongly advise you to use mod_jk 1.2.x, as both mod_jk and mod_jk2 are deprecated, unsupported and no further

developments are going on in the community. The mod_jk 1.2.x binary can be downloaded from **`http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/`**. Rename the downloaded file to **`mod_jk.so`** and copy it under **`APACHE_HOME/modules/`**.

20.1.2. 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 +ForwardURICompat -ForwardDirectories

# JkRequestLogFormat
JkRequestLogFormat "%w %V %T"

# Mount your applications
JkMount /application/* loadbalancer

# You can use external file for mount points.
# It will be checked for updates each 60 seconds.
# The format of the file is: /url=worker
# /examples/*=loadbalancer
JkMountFile conf/uriworkermap.properties

# Add shared memory.
# This directive is present with 1.2.10 and
# later versions of mod_jk, and is needed for
# for load balancing to work properly
JkShmFile logs/jk.shm
```

```
# Add jkstatus for managing runtime data
<Location /jkstatus/>
    JkMount status
    Order deny,allow
    Deny from all
    Allow from 127.0.0.1
</Location>
```

Please note that two settings are very important:

- The **LoadModule** directive must reference the mod_jk library you have downloaded in the previous section. You must indicate the exact same name with the "modules" file path prefix.
- The **JkMount** directive tells Apache which URLs it should forward to the mod_jk module (and, in turn, to the Servlet containers). In the above file, all requests with URL path **/application/*** are sent to the mod_jk load-balancer. This way, you can configure Apache to server static contents (or PHP contents) directly and only use the loadbalancer for Java applications. If you only use mod_jk as a loadbalancer, you can also forward all URLs (i.e., **/***) to mod_jk.

In addition to the **JkMount** directive, you can also use the **JkMountFile** directive to specify a mount points configuration file, which contains multiple Tomcat forwarding URL mappings. You just need to create a **uriworkermap.properties** file in the **APACHE_HOME/conf** directory. The format of the file is **/url=worker_name**. To get things started, paste the following example into the file you created:

```
# Simple worker configuration file

# Mount the Servlet context to the ajp13 worker
/jmx-console=loadbalancer
/jmx-console/*=loadbalancer
/web-console=loadbalancer
/web-console/*=loadbalancer
```

This will configure mod_jk to forward requests to **/jmx-console** and **/web-console** to Tomcat.

You will most probably not change the other settings in **mod_jk.conf**. They are used to tell mod_jk where to put its logging file, which logging level to use and so on.

20.1.3. Configure worker nodes in mod_jk

Next, you need to configure mod_jk workers file **conf/workers.properties**. This file specifies where the different Servlet containers are located and how calls should be load-balanced across them. The configuration file contains one section for each target servlet container and one global section. For a two nodes setup, the file could look like this:

```
# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status
```

```
# Define Node1
# modify the host as your host IP or DNS name.
worker.node1.port=8009
worker.node1.host=node1.mydomain.com
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.cachesize=10

# Define Node2
# modify the host as your host IP or DNS name.
worker.node2.port=8009
worker.node2.host= node2.mydomain.com
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.cachesize=10

# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node1,node2
worker.loadbalancer.sticky_session=1
#worker.list=loadbalancer

# Status worker for managing load balancer
worker.status.type=status
```

Basically, the above file configures mod_jk to perform weighted round-robin load balancing with sticky sessions between two servlet containers (JBoss Tomcat) node1 and node2 listening on port 8009.

In the **works.properties** file, each node is defined using the **worker.XXX** naming convention where **XXX** represents an arbitrary name you choose for each of the target Servlet containers. For each worker, you must specify the host name (or IP address) and the port number of the AJP13 connector running in the Servlet container.

The **lbfactor** attribute is the load-balancing factor for this specific worker. It is used to define the priority (or weight) a node should have over other nodes. The higher this number is for a given worker relative to the other workers, the more HTTP requests the worker will receive. This setting can be used to differentiate servers with different processing power.

The **cachesize** attribute defines the size of the thread pools associated to the Servlet container (i.e. the number of concurrent requests it will forward to the Servlet container). Make sure this number does not outnumber the number of threads configured on the AJP13 connector of the Servlet container. Please review **http://jakarta.apache.org/tomcat/connectors-doc/config/workers.html** for comments on **cachesize** for Apache 1.3.x.

The last part of the **conf/workers.properties** file defines the loadbalancer worker. The only thing you must change is the **worker.loadbalancer.balanced_workers** line: it must list all workers previously defined in the same file: load-balancing will happen over these workers.

The **sticky_session** property specifies the cluster behavior for HTTP sessions. If you specify **worker.loadbalancer.sticky_session=0**, each request will be load balanced between node1 and node2; i.e., different requests for the same session will go to different servers. But when a user opens a session on one server, it is always necessary to always forward this user's requests to the same server, as long as that server is available. This is called a "sticky session", as the client is always

using the same server he reached on his first request. To enable session stickiness, you need to set **worker.loadbalancer.sticky_session** to 1.

Note

A non-loadbalanced setup with a single node requires a **worker.list=node1** entry.

20.1.4. Configuring JBoss to work with mod_jk

Finally, we must configure the JBoss Tomcat instances on all clustered nodes so that they can expect requests forwarded from the mod_jk loadbalancer.

On each clustered JBoss node, we have to name the node according to the name specified in **workers.properties**. For instance, on JBoss instance node1, edit the **JBOSS_HOME/server/all/deploy/jboss-web.deployer/server.xml** file (replace **/all** with your own server name if necessary). Locate the **<Engine>** element and add an attribute **jvmRoute**:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
... ...
</Engine>
```

You also need to be sure the AJP connector in server.xml is enabled (i.e., uncommented). It is enabled by default.

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

Then, for each JBoss Tomcat instance in the cluster, we need to tell it that mod_jk is in use, so it can properly manage the **jvmRoute** appended to its session cookies so that mod_jk can properly route incoming requests. Edit the **JBOSS_HOME/server/all/deploy/jbossweb-tomcat50.sar/META-INF/jboss-service.xml** file (replace **/all** with your own server name). Locate the **<attribute>** element with a name of **UseJK**, and set its value to **true**:

```
<attribute name="UseJK">true</attribute>
```

At this point, you have a fully working Apache+mod_jk load-balancer setup that will balance call to the Servlet containers of your cluster while taking care of session stickiness (clients will always use the same Servlet container).

Note

For more updated information on using mod_jk 1.2 with JBoss Tomcat, please refer to the JBoss wiki page at **http://wiki.jboss.org/wiki/Wiki.jsp?page=UsingMod_jk1.2WithJBoss**.

20.2. Configuring HTTP session state replication

The preceding discussion has been focused on using mod_jk as a load balancer. The content of the remainder our discussion of clustering HTTP services in JBoss Enterprise Application Platform applies no matter what load balancer is used.

In *Section 20.1.3, "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 better and more reliable solution is to replicate session data across the nodes in the cluster. This way, the client can hit any server node and obtain the same session state.

The **`jboss.cache:service=TomcatClusteringCache`** MBean makes use of JBoss Cache to provide HTTP session replication services to the JBoss Tomcat cluster. This MBean is defined in the **`deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml file`**.

Note

Before JBoss Enterprise Application Platform 4.2.0, the location of the HTTP session cache configuration file was **`deploy/tc5-cluster.sar/META-INF/jboss-service.xml`**.

Below is a typical **`deploy/jbossweb-cluster.sar/META-INF/jboss-service.xml`** file. The configuration attributes in the **`TomcatClusteringCache`** MBean are very similar to those in the JBoss Enterprise Application Platform cache configuration.

```
<mbean code="org.jboss.cache.aop.TreeCacheAop"
    name="jboss.cache:service=TomcatClusteringCache">

    <depends>jboss:service=Naming</depends>
    <depends>jboss:service=TransactionManager</depends>
    <depends>jboss.aop:service=AspectDeployer</depends>

    <attribute name="TransactionManagerLookupClass">
        org.jboss.cache.BatchModeTransactionManagerLookup
    </attribute>

    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>

    <attribute name="CacheMode">REPL_ASYNC</attribute>

    <attribute name="ClusterName">
      Tomcat-${jboss.partition.name:Cluster}
    </attribute>

    <attribute name="UseMarshalling">false</attribute>

    <attribute name="InactiveOnStartup">false</attribute>

    <attribute name="ClusterConfig">
        ... ...
```

```
    </attribute>


    <attribute name="LockAcquisitionTimeout">15000</attribute>
    <attribute name="SyncReplTimeout">20000</attribute>
</mbean>
```

Note that the value of the mbean element's code attribute is org.jboss.cache.aop.TreeCacheAop, which is different from the other JBoss Cache Mbeans used in JBoss Enterprise Application Platform. This is because FIELD granularity HTTP session replication (covered below) needs the added features of the **`TreeCacheAop`** (a.k.a. **`PojoCache`**) class.

The details of all the configuration options for a TreeCache MBean are covered in the JBoss Cache documentation. Below, we will just discuss several attributes that are most relevant to the HTTP cluster session replication.

- **TransactionManagerLookupClass** sets the transaction manager factory. The default value is **`org.jboss.cache.BatchModeTransactionManagerLookup`**. It tells the cache NOT to participate in JTA-specific transactions. Instead, the cache manages its own transactions. Please do not change this.
- **CacheMode** controls how the cache is replicated. The valid values are **`REPL_SYNC`** and **`REPL_ASYNC`**. With either setting the client request thread updates the local cache with the current sesssion contents and then sends a message to the caches on the other members of the cluster, telling them to make the same change. With REPL_ASYNC (the default) the request thread returns as soon as the update message has been put on the network. With REPL_SYNC, the request thread blocks until it gets a reply message from all cluster members, informing it that the update was successfully applied. Using synchronous replication makes sure changes are applied aroundthe cluster before the web request completes. However, synchronous replication is much slower.
- **ClusterName** specifies the name of the cluster that the cache works within. The default cluster name is the the word "Tomcat-" appended by the current JBoss partition name. All the nodes must use the same cluster name.
- The **UseMarshalling** and **InactiveOnStartup** attributes must have the same value. They must be **`true`** if **`FIELD`** level session replication is needed (see later). Otherwise, they are default to **`false`**.
- **ClusterConfig** configures the underlying JGroups stack. Please refer to *Section 23.1, "Configuring a JGroups Channel's Protocol Stack"* for more information.
- **LockAcquisitionTimeout** sets the maximum number of milliseconds to wait for a lock acquisition when trying to lock a cache node. The default value is 15000.
- **SyncReplTimeout** sets the maximum number of milliseconds to wait for a response from all nodes in the cluster when a synchronous replication message is sent out. The default value is 20000; should be a few seconds longer than LockAcquisitionTimeout.

20.2.1. Enabling session replication in your application

To enable clustering of your web application you must tag it as distributable in the **`web.xml`** descriptor. Here's an example:

```
<?xml version="1.0"?>
```

```
<web-app  xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                              http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd"
          version="2.4">
    <distributable/>
    <!-- ... -->
</web-app>
```

You can futher configure session replication using the **`replication-config`** element in the **`jboss-web.xml`** file. Here is an example:

```
<jboss-web>
    <replication-config>
        <replication-trigger>SET_AND_NON_PRIMITIVE_GET</replication-
trigger>
        <replication-granularity>SESSION</replication-granularity>
        <replication-field-batch-mode>true</replication-field-batch-mode>
    </replication-config>
</jboss-web>
```

The **`replication-trigger`** element determines what triggers a session replication (i.e. when is a session is considered **`dirty`** and in need of replication). It has 4 options:

- **SET**: With this policy, the session is considered dirty only when an attribute is set in the session (i.e., HttpSession.setAttribute() is invoked.) If your application always writes changed values back into the session, this option will be most optimal in terms of performance. The downside of SET is that if an object is retrieved from the session and modified without being written back into the session, the session manager will not know the attribute is dirty and the change to that object may not be replicated.

- **SET_AND_GET**: With this policy, any attribute that is get or set will be marked as dirty. If an object is retrieved from the session and modified without being written back into the session, the change to that object will be replicated. The downside of SET_AND_GET is that it can have significant performance implications, since even reading immutable objects from the session (e.g., strings, numbers) will mark the read attributes as needing to be replicated.

- **SET_AND_NON_PRIMITIVE_GET**: This policy is similar to the SET_AND_GET policy except that get operationsthat return attribute values with primitive types do not mark the attribute as dirty. Primitive system types (i.e., String, Integer, Long, etc.) are immutable, so there is no reason to mark an attribute with such a type as dirty just because it has been read. If a get operation returns a value of a non-primitive type, the session manager has no simple way to know whether the object is mutable, so it assumes it is an marks the attribute as dirty. This setting avoids the downside of SET while reducing the performance impact of SET_AND_GET. It is the default setting.

- **ACCESS**: This option causes the session to be marked as dirty whenever it is accessed. Since a the session is accessed during each HTTP request, it will be replicated with each request. The purpose of ACCESS is to ensure session last-access timestamps are kept in sync around the cluster.. Since with the other replication-trigger options the time stamp may not be updated in other clustering nodes because of no replication, the session in other nodes may expire before the active node if the HTTP request does not retrieve or modify any session attributes. When this option is

set, the session timestamps will be synchronized throughout the cluster nodes. Note that use of this option can have a significant performance impact, so use it with caution. With the other replication-trigger options, if a session has gone 80% of its expiration interval without being replicated, as a safeguard its timestamp will be replicated no matter what. So, ACCESS is only useful in special circumstances where the above safeguard is considered inadequate.

The **`replication-granularity`** element controls the size of the replication units. The supported values are:

- **ATTRIBUTE**: Replication is only for the dirty attributes in the session plus some session data, like the last-accessed timestamp. For sessions that carry large amounts of data, this option can increase replication performance. However, attributes will be separately serialized, so if there are any shared references between objects stored in the attributes, those shared references may be broken on remote nodes. For example, say a Person object stored under key “husband” has a reference to an Address, while another Person object stored under key “wife” has a reference to that same Address object. When the “husband” and “wife” attributes are separately deserialized on the remote nodes, each Person object will now have a reference to its own Address object; the Address object will no longer be shared.
- **SESSION**: The entire session object is replicated if any attribute is dirty. The entire session is serialized in one unit, so shared object references are maintained on remote nodes. This is the default setting.
- **FIELD**: Replication is only for individual changed data fields inside session attribute objects. Shared object references will be preserved across the cluster. Potentially most performant, but requires changes to your application (this will be discussed later).

The **`replication-field-batch-mode`** element indicates whether you want all replication messages associated with a request to be batched into one message. Only applicable if replication-granularity is FIELD. Default is **`true`**.

If your sessions are generally small, SESSION is the better policy. If your session is larger and some parts are infrequently accessed, ATTRIBUTE replication will be more effective. If your application has very big data objects in session attributes and only fields in those objects are frequently modified, the FIELD policy would be the best. In the next section, we will discuss exactly how the FIELD level replication works.

20.2.2. Using FIELD level replication

FIELD-level replication only replicates modified data fields inside objects stored in the session. Its use could potentially drastically reduce the data traffic between clustered nodes, and hence improve the performance of the whole cluster. To use FIELD-level replication, you have to first prepare (i.e., bytecode enhance) your Java class to allow the session cache to detect when fields in cached objects have been changed and need to be replicated.

The first step in doing this is to identify the classes that need to be prepared. This is done via annotations. For example:

```
@org.jboss.cache.aop.AopMarker
public class Address
{
...
```

```
}
```

If you annotate a class with InstanceAopMarker instead, then all of its subclasses will be automatically annotated as well. Similarly, you can annotate an interface with InstanceofAopMarker and all of its implementing classes will be annotated. For example:

```
@org.jboss.cache.aop.InstanceOfAopMarker
public class Person
{
...
}
then when you have a sub-class like
public class Student extends Person
{
...
}
```

There will be no need to annotate **`Student`**. It will be annotated automatically because it is a sub-class of **`Person`**. Jboss Enterprise Application Platform 4.2 requires JDK 5 at runtime, but some users may still need to build their projects using JDK 1.4. In this case, annotating classes can be done via JDK 1.4 style annotations embedded in JavaDocs. For example:

```
/*
 * My usual comments here first.
 * @@org.jboss.web.tomcat.tc5.session.AopMarker
 */
public class Address
{
...
}
```

The anologue for **`@InstanceAopMarker`** is:

```
/*
 *
 * @@org.jboss.web.tomcat.tc5.session.InstanceOfAopMarker
 */
public class Person
{
...
}
```

Once you have annotated your classes, you will need to perform a pre-processing step to bytecode enhance your classes for use by TreeCacheAop. You need to use the JBoss AOP pre-compiler **`annotationc`** and post-compiler **`aopc`** to process the above source code before and after they are compiled by the Java compiler. The annotationc step is only need if the JDK 1.4 style annotations are used; if JDK 5 annotations are used it is not necessary. Here is an example on how to invoke those commands from command line.

```
$ annotationc [classpath] [source files or directories]
$ javac -cp [classpath] [source files or directories]
$ aopc [classpath] [class files or directories]
```

Please see the JBoss AOP documentation for the usage of the pre- and post-compiler. The JBoss AOP project also provides easy to use ANT tasks to help integrate those steps into your application build process.

Note

You can see a complete example on how to build, deploy, and validate a FIELD-level replicated web application from this page: *http://wiki.jboss.org/wiki/Wiki.jsp?page=Http_session_field_level_example*. The example bundles the pre- and post-compile tools so you do not need to download JBoss AOP separately.

When you deploy the web application into JBoss Enterprise Application Platform, make sure that the following configurations are correct:

- In the server's **`deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml`** file, the **`inactiveOnStartup`** and **`useMarshalling`** attributes must both be **`true`**.
- In the application's **`jboss-web.xml`** file, the **`replication-granularity`** attribute must be **`FIELD`**.

Finally, let's see an example on how to use FIELD-level replication on those data classes. Notice that there is no need to call **`session.setAttribute()`** after you make changes to the data object, and all changes to the fields are automatically replicated across the cluster.

```
// Do this only once. So this can be in init(), e.g.
if(firstTime)
{
  Person joe = new Person("Joe", 40);
  Person mary = new Person("Mary", 30);
  Address addr = new Address();
  addr.setZip(94086);

  joe.setAddress(addr);
  mary.setAddress(addr); // joe and mary share the same address!

  session.setAttribute("joe", joe); // that's it.
  session.setAttribute("mary", mary); // that's it.
}

Person mary = (Person)session.getAttribute("mary");
mary.getAddress().setZip(95123); // this will update and replicate the zip
 code.
```

Besides plain objects, you can also use regular Java collections of those objects as session attributes. JBoss cache automatically figures out how to handle those collections and replicate field changes in their member objects.

20.3. Monitoring session replication

If you have deployed and accessed your application, go to the **`jboss.cache:service=TomcatClusteringCache`** MBean and invoke the **`printDetails`** operation. You should see output resembling the following.

```
/JSESSION

/localhost

/quote

/FB04767C454BAB3B2E462A27CB571330
VERSION: 6
FB04767C454BAB3B2E462A27CB571330:
 org.jboss.invocation.MarshalledValue@1f13a81c

/AxCI8Ovt5VQTfNyYy9Bomw**
VERSION: 4
AxCI8Ovt5VQTfNyYy9Bomw**: org.jboss.invocation.MarshalledValue@e076e4c8
```

This output shows two separate web sessions, in one application named *quote*, that are being shared via JBossCache. This example uses a **`replication-granularity`** of **`session`**. Had **`ATTRIBUTE`** level replication been used, there would be additional entries showing each replicated session attribute. In either case, the replicated values are stored in an opaque **`MarshelledValue`** container. There aren't currently any tools that allow you to inspect the contents of the replicated session values. If you do not see any output, either the application was not correctly marked as **`distributable`** or you haven't accessed a part of application that places values in the HTTP session. The **`org.jboss.cache`** and **`org.jboss.web`** logging categories provide additional insight into session replication useful for debugging purposes.

20.4. Using Clustered Single Sign On

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 apps 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 webapps to different servers, while maintaining the SSO.

20.4.1. Configuration

To enable clustered single sign-on, you must add the **`ClusteredSingleSignOn`** valve to the appropriate **`Host`** elements of the **`JBOSS_HOME/server/all/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** is a required attribute to set the Java class name of the valve implementation to use. This must be set to **`org.jboss.web.tomcat.service.sso.ClusteredSingleSign`**.
- **cacheConfig** is the name of the cache configuration (see the Editing the CacheManager Configuration section) to use for the clustered SSO cache. Default is **`clustered-sso`**.
- **treeCacheName** is 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`** (see see the Editing the CacheManager Configuration section), an attempt to locate an mbean registered in JMX under this ObjectName will be made. Default value is **`jboss.cache:service=TomcatClusteringCache`**.
- **cookieDomain** is used to set the host domain to be used for sso cookies. See *Section 20.4.4, "Configuring the Cookie Domain"* for more. Default is **`"/"`**.
- **maxEmptyLife** is 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`**, i.e. 30 minutes.
- **processExpiresInterval** is 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** is 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`**.

20.4.2. SSO Behavior

The user will not be challenged as long as he accesses only unprotected resources in any of the web applications on the virtual host.

Upon access to a protected resource in any web app, the user will be challenged to authenticate, using the login method defined for the web app.

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.

20.4.3. Limitations

There are a number of known limitations to this Tomcat valve-based SSO implementation:

- Only useful within a cluster of JBoss servers; SSO does not propagate to other resources.
- Requires use of container managed authentication (via **`<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 webapps, the varying security integrations must all accept the same credentials (e.g. username and password).

20.4.4. 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 (i.e. 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 apps 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" />
```

JBoss Messaging Clustering Notes

21.1. Unique server peer id

JBoss Messaging clustering should work out of the box in the *all* configuration with no configuration changes. It is however crucial that every node is assigned a unique server id.

Every node deployed must have a unique id, including those in a particular LAN cluster, and also those only linked by message bridges.

21.2. Clustered destinations

JBoss Messaging clusters JMS queues and topics transparently across the cluster. Messages sent to a distributed queue or topic on one node are consumable on other nodes. To designate that a particular destination is clustered simply set the clustered attribute in the destination deployment descriptor to true.

JBoss Messaging balances messages between nodes, catering for faster or slower consumers to efficiently balance processing load across the cluster.

If you do not want message redistribution between nodes, but still want to retain the other characteristics of clustered destinations, you can specify the attribute **`ClusterPullConnectionFactoryName`** on the server peer.

21.3. Clustered durable subs

JBoss Messaging durable subscriptions can also be clustered. This means multiple subscribers can consume from the same durable subscription from different nodes of the cluster. A durable subscription will be clustered if it's topic is clustered.

21.4. Clustered temporary destinations

JBoss Messaging also supports clustered temporary topics and queues. All temporary topics and queues will be clustered if the post office is clustered.

21.5. Non clustered servers

If you don't want your nodes to participate in a cluster, or only have one non clustered server you can set the clustered attribute on the postoffice to **`false`**.

21.6. Message ordering in the cluster

If you wish to apply strict JMS ordering to messages, such that a particular JMS consumer consumes messages in the same order as they were produced by a particular producer, you can set the **`DefaultPreserveOrdering`** attribute in the server peer to **`true`**. By default this is false.

Note

The side effect of setting this to true is that messages cannot be distributed as freely around the cluster.

21.7. Idempotent operations

If the call to send a persistent message to a persistent destination returns successfully with no exception, then you can be sure that the message was persisted. However if the call doesn't return successfully e.g. if an exception is thrown, then you *can't be sure the message wasn't persisted*. This is because the failure might have occurred after persisting the message but before writing the response to the caller. This is a common attribute of any RPC type call: You can't tell by the call not returning that the call didn't actually succeed. Whether it's a web services call, a HTTP get request, an EJB invocation the same applies. The trick is to code your application so your operations are *idempotent* i.e. they can be repeated without getting the system into an inconsistent state. With a message system you can do this on the application level, by checking for duplicate messages, and discarding them if they arrive. Duplicate checking is a very powerful technique that can remove the need for XA transactions in many cases.

21.7.1. Clustered connection factories

If the supportsLoadBalancing attribute of the connection factory is set to true then consecutive create connection attempts will round robin between available servers. The first node to try is chosen randomly.

If the supportsFailover attribute of the connection factory is set to true then automatic failover is enabled. This will automatically failover from one server to another, transparently to the user, in case of failure.

If automatic failover is not required or you wish to do manual failover (JBoss MQ style) this can be set to false, and you can supply a standard JMS ExceptionListener on the connection which will be called in case of connection failure. You would then need to manually close the connection, lookup a new connection factory from HA JNDI and recreate the connection.

Clustered Deployment Options

22.1. Clustered Singleton Services

A clustered singleton service (also known as an HA singleton) is a service that is deployed on multiple nodes in a cluster, but is providing its service on only one of the nodes. The node running the singleton service is typically called the master node. When the master fails or is shut down, another master is selected from the remaining nodes and the service is restarted on the new master. Thus, other than a brief interval when one master has stopped and another has yet to take over, the service is always being provided by one but only one node.

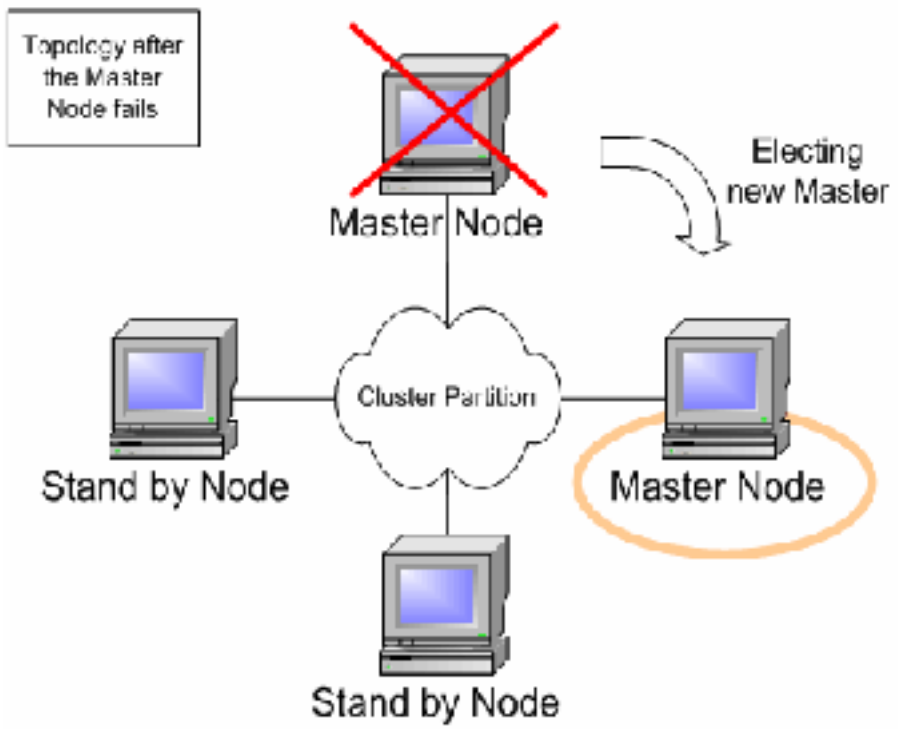


Figure 22.1. Topology after the Master Node fails

22.1.1. HASingleton Deployment Options

The JBoss Enterprise Application 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/all/deploy-hasingleton`** directory instead of in **`deploy`**. The **`deploy-hasingleton`** directory does not lie under deploy or farm, so its contents are not automatically deployed when an Enterprise Application Platform instance starts.

Instead, deploying the contents of this directory is the responsibility of a special service, the **`jboss.ha:service=HASingletonDeployer`** MBean (which itself is deployed via the deploy/deploy-hasingleton-service.xml file.) The HASingletonDeployer service is itself an HA Singleton, one whose provided service when it becomes master is to deploy the contents of deploy-hasingleton and whose service when it stops being the master (typically at server shutdown) is to undeploy the contents of **`deploy-hasingleton`**.

So, by placing your deployments in **`deploy-hasingleton`** you know that they will be deployed only on the master node in the cluster. If the master node cleanly shuts down, they will be cleanly undeployed as part of shutdown. If the master node fails or is shut down, they will be deployed on whatever node takes over as master.

Using deploy-hasingleton is very simple, but it does have two drawbacks:

- There is no hot-deployment feature for services in **`deploy-hasingleton`** . Redeploying a service that has been deployed to **`deploy-hasingleton`** requires a server restart.
- If the master node fails and another node takes over as master, your singleton service needs to go through the entire deployment process before it will be providing services. Depending on how complex the deployment of your service is and what sorts of startup activities it engages in, this could take a while, during which time the service is not being provided.

22.1.1.2. Mbean deployments using HASingletonController

If your service is an Mbean (i.e., not a J2EE deployment like an ear or war or jar), you can deploy it along with a service called an HASingletonController in order to turn it into an HA singleton. It is the job of the HASingletonController to work with the HAPartition service to monitor the cluster and determine if it is now the master node for its service. If it determines it has become the master node, it invokes a method on your service telling it to begin providing service. If it determines it is no longer the master node, it invokes a method on your service telling it to stop providing service. Let's walk through an illustration.

First, we have an MBean service that we want to make an HA singleton. The only thing special about it is it needs to expose in its MBean interface a method that can be called when it should begin providing service, and another that can be called when it should stop providing service:

```
public class HASingletonExample implements HASingletonExampleMBean
{
   private boolean isMasterNode = false;

   public void startSingleton()
   {
     isMasterNode = true;
   }

   public boolean isMasterNode()
   {
      return isMasterNode;
   }

   public void stopSingleton()
   {
```

```
      isMasterNode = false;
   }
}
```

We used **`startSingleton`** and **`stopSingleton`** in the above example, but you could name the methods anything.

Next, we deploy our service, along with an HASingletonController to control it, most likely packaged in a .sar file, with the following **`META-INF/jboss-service.xml`**:

```
<server>
  <!-- This MBean is an example of a clustered singleton -->
  <mbean code="org.jboss.ha.examples.HASingletonExample"
         name="jboss:service=HASingletonExample"/>

  <!-- This HASingletonController manages the cluster Singleton -->
  <mbean code="org.jboss.ha.singleton.HASingletonController"
         name="jboss:service=ExampleHASingletonController">

    <!-- Inject a ref to the HAPartition -->
    <depends optional-attribute-name="ClusterPartition" proxy-
type="attribute">
      jboss:service=${jboss.partition.name:DefaultPartition}
    </depends>
    <!-- Inject a ref to the service being controlled -->
    <depends optional-attribute-
name="TargetName">jboss:service=HASingletonExample</depends>

    <!-- Methods to invoke when become master / stop being master -->
    <attribute name="TargetStartMethod">startSingleton</attribute>
    <attribute name="TargetStopMethod">stopSingleton</attribute>
  </mbean>
</server>
```

Voila! A clustered singleton service.

The obvious downside to this approach is it only works for MBeans. Upsides are that the above example can be placed in **`deploy`** or **`farm`** and thus can be hot deployed and farmed deployed. Also, if our example service had complex, time-consuming startup requirements, those could potentially be implemented in create() or start() methods. JBoss will invoke create() and start() as soon as the service is deployed; it doesn't wait until the node becomes the master node. So, the service could be primed and ready to go, just waiting for the controller to implement startSingleton() at which point it can immediately provide service.

The jboss.ha:service=HASingletonDeployer service discussed above is itself an interesting example of using an HASingletonController. Here is its deployment descriptor (extracted from the **`deploy/deploy-hasingleton-service.xml`** file):

```
<mbean code="org.jboss.ha.singleton.HASingletonController"
       name="jboss.ha:service=HASingletonDeployer">
```

```
  <depends optional-attribute-name="ClusterPartition" proxy-
type="attribute">
    jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
  <depends optional-attributeame="TargetName">
    jboss.system:service=MainDeployer
  </depends>
  <attribute name="TargetStartMethod">deploy</attribute>
  <attribute name="TargetStartMethodArgument">
    ${jboss.server.home.url}/deploy-hasingleton
  </attribute>
  <attribute name="TargetStopMethod">undeploy</attribute>
  <attribute name="TargetStopMethodArgument">
    ${jboss.server.home.url}/deploy-hasingleton
 </attribute>
</mbean>
```

A few interesting things here. First the service being controlled is the **`MainDeployer`** service, which is the core deployment service in JBoss. That is, it's a service that wasn't written with an intent that it be controlled by an **`HASingletonController`**. But it still works! Second, the target start and stop methods are **`deploy`** and **`undeploy`**. No requirement that they have particular names, or even that they logically have *start* and *stop* functionality. Here the functionality of the invoked methods is more like *do* and *undo*. Finally, note the **`TargetStart(Stop)MethodArgument`** attributes. Your singleton service's start/stop methods can take an argument, in this case the location of the directory the **`MainDeployer`** should deploy/undeploy.

22.1.1.3. HASingleton deployments using a Barrier

Services deployed normally inside deploy or farm that want to be started/stopped whenever the content of deploy-hasingleton gets deployed/undeployed, (i.e., whenever the current node becomes the master), need only specify a dependency on the Barrier mbean:

```
<depends>jboss.ha:service=HASingletonDeployer,type=Barrier</depends>
```

The way it works is that a BarrierController is deployed along with the jboss.ha:service=HASingletonDeployer MBean and listens for JMX notifications from it. A BarrierController is a relatively simple Mbean that can subscribe to receive any JMX notification in the system. It uses the received notifications to control the lifecycle of a dynamically created Mbean called the Barrier.The Barrier is instantiated, registered and brought to the CREATE state when the BarrierController is deployed. After that, the BarrierController starts and stops the Barrier when matching JMX notifications are received. Thus, other services need only depend on the Barrier MBean using the usual <depends> tag, and they will be started and stopped in tandem with the Barrier. When the BarrierController is undeployed the Barrier is destroyed too.

This provides an alternative to the deploy-hasingleton approach in that we can use farming to distribute the service, while content in deploy-hasingleton must be copied manually on all nodes.

On the other hand, the barrier-dependent service will be instantiated/created (i.e., any create() method invoked) on all nodes, but only started on the master node. This is different with the deploy-hasingleton approach that will only deploy (instantiate/create/start) the contents of the deploy-hasingleton directory on one of the nodes.

So services depending on the barrier will need to make sure they do minimal or no work inside their create() step, rather they should use start() to do the work.

Note

The Barrier controls the start/stop of dependent services, but not their destruction, which happens only when the **`BarrierController`** is itself destroyed/undeployed. Thus using the **`Barrier`** to control services that need to be "destroyed" as part of their normal “undeploy” operation (like, for example, an **`EJBContainer`**) will not have the desired effect.

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 decide whether it is now the “master node”. How is this done?

For each member of the cluster, the HAPartition mbean maintains an attribute called the CurrentView, which is basically an ordered list of the current members of the cluster. As nodes join and leave the cluster, JGroups ensures that each surviving member of the cluster gets an updated view. You can see the current view by going into the JMX console, and looking at the CurrentView attribute in the **`jboss:service=DefaultPartition`** mbean. Every member of the cluster will have the same view, with the members in the same order.

Let's say, for example, that we have a 4 node cluster, nodes A through D, and the current view can be expressed as {A, B, C, D}. Generally speaking, the order of nodes in the view will reflect the order in which they joined the cluster (although this is not always the case, and should not be assumed to be the case).

To further our example, let's say there is a singleton service (i.e. an **`HASingletonController`**) named Foo that's deployed around the cluster, except, for whatever reason, on B. The **`HAPartition`** service maintains across the cluster a registry of what services are deployed where, in view order. So, on every node in the cluster, the **`HAPartition`** service knows that the view with respect to the Foo service is {A, C, D} (no B).

Whenever there is a change in the cluster topology of the Foo service, the **`HAPartition`** service invokes a callback on Foo notifying it of the new topology. So, for example, when Foo started on D, the Foo service running on A, C and D all got callbacks telling them the new view for Foo was {A, C, D}. That callback gives each node enough information to independently decide if it is now the master. The Foo service on each node does this by checking if they are the first member of the view – if they are, they are the master; if not, they're not. Simple as that.

If A were to fail or shutdown, Foo on C and D would get a callback with a new view for Foo of {C, D}. C would then become the master. If A restarted, A, C and D would get a callback with a new view for Foo of {C, D, A}. C would remain the master – there's nothing magic about A that would cause it to become the master again just because it was before.

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 HA singleton, following a change in cluster topology.

```
public interface HASingletonElectionPolicy
{
   ClusterNode elect(List<ClusterNode> nodes);
}
```

JBoss ships with 2 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 2nd oldest; etc. **`position`** can also be negative to indicate youngness; imagine the list of available nodes as a circular linked list. **`position = -1`**, refers to the youngest node; **`position = -2`**, refers to the 2nd youngest node; 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 Farm Service previously available in JBoss 4.x is not available in JBoss 5.0 as it was incompatible with the new Profile Service at the core of the Enterprise Application Platform. A new Profile Service-based replacement for the Farm Service will be added in a future release.

JGroups Services

JGroups provides the underlying group communication support for JBoss Enterprise Application Platform clusters. JBoss Enterprise Application Platform ships with a reasonable set of default JGroups configurations. Most applications 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.

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. It is built on top a stack of network communication protocols that provide transport, discovery, reliability and failure detection, and cluster membership management services. *Figure 23.1, "Protocol stack in JGroups"* shows the protocol stack in JGroups.

Figure 23.1. Protocol stack in JGroups

JGroups configurations often appear as a nested attribute in cluster related MBean services, such as the **`PartitionConfig`** attribute in the **`ClusterPartition`** MBean or the **`ClusterConfig`** attribute in the **`TreeCache`** MBean. You can configure the behavior and properties of each protocol in JGroups via those MBean attributes. Below is an example JGroups configuration in the **`ClusterPartition`** MBean.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
 name="jboss:service=${jboss.partition.name:DefaultPartition}">

  ... ...

  <attribute name="PartitionConfig">
   <Config>

    <UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
         mcast_port="${jboss.hapartition.mcast_port:45566}"
         tos="8"
         ucast_recv_buf_size="20000000"
         ucast_send_buf_size="640000"
         mcast_recv_buf_size="25000000"
         mcast_send_buf_size="640000"
         loopback="false"
         discard_incompatible_packets="true"
         enable_bundling="false"
         max_bundle_size="64000"
         max_bundle_timeout="30"
         use_incoming_packet_handler="true"
         use_outgoing_packet_handler="false"
         ip_ttl="${jgroups.udp.ip_ttl:2}"
         down_thread="false" up_thread="false"/>

    <PING timeout="2000"
```

```
          down_thread="false" up_thread="false" num_initial_members="3"/>

    <MERGE2 max_interval="100000"
     down_thread="false" up_thread="false" min_interval="20000"/>
    <FD_SOCK down_thread="false" up_thread="false"/>

    <FD timeout="10000" max_tries="5"
        down_thread="false" up_thread="false" shun="true"/>
    <VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
    <pbcast.NAKACK max_xmit_size="60000"
     use_mcast_xmit="false" gc_lag="0"
     retransmit_timeout="300,600,1200,2400,4800"
     down_thread="false" up_thread="false"
     discard_delivered_msgs="true"/>
    <UNICAST timeout="300,600,1200,2400,3600"
      down_thread="false" up_thread="false"/>
    <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
     down_thread="false" up_thread="false"
     max_bytes="400000"/>
    <pbcast.GMS print_local_addr="true" join_timeout="3000"
         down_thread="false" up_thread="false"
         join_retry_timeout="2000" shun="true"
         view_bundling="true"/>
    <FRAG2 frag_size="60000" down_thread="false" up_thread="false"/>
    <pbcast.STATE_TRANSFER down_thread="false"
      up_thread="false" use_flush="false"/>
   </Config>
  </attribute>
</mbean>
```

All the JGroups configuration data is contained in the <Config> element under the JGroups config MBean attribute. This information is used to configure a JGroups Channel; the Channel is conceptually similar to a socket, and manages communication between peers in a cluster. Each element inside the <Config> element defines a particular JGroups Protocol; each Protocol performs one function, and the combination of those functions is what defines the characteristics of the overall Channel. In the next several sections, we will dig into the commonly used protocols and their options and explain exactly what they mean.

23.1.1. Common Configuration Properties

The following common properties are exposed by all of the JGroups protocols discussed below:

- **`down_thread`** whether the protocol should create an internal queue and a queue processing thread (aka the down_thread) for messages passed down from higher layers. The higher layer could be another protocol higher in the stack, or the application itself, if the protocol is the top one on the stack. If true (the default), when a message is passed down from a higher layer, the calling thread places the message in the protocol's queue, and then returns immediately. The protocol's down_thread is responsible for reading messages off the queue, doing whatever protocol-specific processing is required, and passing the message on to the next protocol in the stack.
- **`up_thread`** is conceptually similar to down_thread, but here the queue and thread are for messages received from lower layers in the protocol stack.

Generally speaking, **up_thread** and **down_thread** should be set to false.

23.1.2. Transport Protocols

The transport protocols send messages from one cluster node to another (unicast) or from cluster node to all other nodes in the cluster (mcast). JGroups supports UDP, TCP, and TUNNEL as transport protocols.

Note

The **UDP**, **TCP**, and **TUNNEL** elements are mutually exclusive. You can only have one transport protocol in each JGroups **Config** element

23.1.2.1. UDP configuration

UDP is the preferred protocol for JGroups. UDP uses multicast or multiple unicasts to send and receive messages. If you choose UDP as the transport protocol for your cluster service, you need to configure it in the **UDP** sub-element in the JGroups **Config** element. Here is an example.

```
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
     mcast_port="${jboss.hapartition.mcast_port:45566}"
     tos="8"
     ucast_recv_buf_size="20000000"
     ucast_send_buf_size="640000"
     mcast_recv_buf_size="25000000"
     mcast_send_buf_size="640000"
     loopback="false"
     discard_incompatible_packets="true"
     enable_bundling="false"
     max_bundle_size="64000"
     max_bundle_timeout="30"
     use_incoming_packet_handler="true"
     use_outgoing_packet_handler="false"
     ip_ttl="${jgroups.udp.ip_ttl:2}"
 down_thread="false" up_thread="false"/>
```

The available attributes in the above JGroups configuration are listed below.

- **ip_mcast** specifies whether or not to use IP multicasting. The default is **true**. If set to false, it will send n unicast packets rather than 1 multicast packet. Either way, packets are UDP datagrams.
- **mcast_addr** specifies the multicast address (class D) for joining a group (i.e., the cluster). If omitted, the default is **228.8.8.8** .
- **mcast_port** specifies the multicast port number. If omitted, the default is **45566**.
- **bind_addr** specifies the interface on which to receive and send multicasts (uses the **-Djgroups.bind_address** system property, if present). If you have a multihomed machine, set the **bind_addr** attribute or system property to the appropriate NIC IP address. By default, system property setting takes priority over XML attribute unless -Djgroups.ignore.bind_addr system property is set.

- **receive_on_all_interfaces** specifies whether this node should listen on all interfaces for multicasts. The default is **`false`**. It overrides the **`bind_addr`** property for receiving multicasts. However, **`bind_addr`** (if set) is still used to send multicasts.
- **send_on_all_interfaces** specifies whether this node send UDP packets via all the NICs if you have a multi NIC machine. This means that the same multicast message is sent N times, so use with care.
- **receive_interfaces** specifies a list of of interfaces to receive multicasts on. The multicast receive socket will listen on all of these interfaces. This is a comma-separated list of IP addresses or interface names. E.g. "**`192.168.5.1,eth1,127.0.0.1`**".
- **ip_ttl** specifies time-to-live for IP Multicast packets. TTL is the commonly used term in multicast networking, but is actually something of a misnomer, since the value here refers to how many network hops a packet will be allowed to travel before networking equipment will drop it.
- **use_incoming_packet_handler** specifies whether to use a separate thread to process incoming messages. Sometimes receivers are overloaded (they have to handle de-serialization etc). Packet handler is a separate thread taking care of de-serialization, receiver thread(s) simply put packet in queue and return immediately. Setting this to true adds one more thread. The default is **`true`**.
- **use_outgoing_packet_handler** specifies whether to use a separate thread to process outgoing messages. The default is false.
- **enable_bundling** specifies whether to enable message bundling. If it is **`true`**, the node would queue outgoing messages until **`max_bundle_size`** bytes have accumulated, or **`max_bundle_time`** milliseconds have elapsed, whichever occurs first. Then bundle queued messages into a large message and send it. The messages are unbundled at the receiver. The default is **`false`**.
- **loopback** specifies whether to loop outgoing message back up the stack. In **`unicast`** mode, the messages are sent to self. In **`mcast`** mode, a copy of the mcast message is sent. The default is **`false`**
- **discard_incompatibe_packets** specifies whether to discard packets from different JGroups versions. Each message in the cluster is tagged with a JGroups version. When a message from a different version of JGroups is received, it will be discarded if set to true, otherwise a warning will be logged. The default is **`false`**
- **mcast_send_buf_size, mcast_recv_buf_size, ucast_send_buf_size, ucast_recv_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.
- **`tos`** specifies traffic class for sending unicast and multicast datagrams.

Note

On Windows 2000 machines, because of the media sense feature being broken with multicast (even after disabling media sense), you need to set the UDP protocol's **`loopback`** attribute to **`true`**.

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. Here is an example of the **`TCP`** element.

```
<TCP start_port="7800"
    bind_addr="192.168.5.1"
    loopback="true"
    down_thread="false" up_thread="false"/>
```

Below are the attributes available in the **`TCP`** element.

- **bind_addr** specifies the binding address. It can also be set with the **`-Djgroups.bind_address`** command line option at server startup.
- **start_port, end_port** define the range of TCP ports the server should bind to. The server socket is bound to the first available port from **`start_port`**. If no available port is found (e.g., because of a firewall) before the **`end_port`**, the server throws an exception. If no **`end_port`** is provided or **`end_port < start_port`** then there is no upper limit on the port range. If **`start_port == end_port`**, then we force JGroups to use the given port (start fails if port is not available). The default is 7800. If set to 0, then the operating system will pick a port. Please, bear in mind that setting it to 0 will work only if we use MPING or TCPGOSSIP as discovery protocol because **`TCCPING`** requires listing the nodes and their corresponding ports.
- **loopback** specifies whether to loop outgoing message back up the stack. In **`unicast`** mode, the messages are sent to self. In **`mcast`** mode, a copy of the mcast message is sent. The default is false.
- **recv_buf_size, send_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.
- **conn_expire_time** specifies the time (in milliseconds) after which a connection can be closed by the reaper if no traffic has been received.
- **reaper_interval** specifies interval (in milliseconds) to run the reaper. If both values are 0, no reaping will be done. If either value is > 0, reaping will be enabled. By default, reaper_interval is 0, which means no reaper.
- **sock_conn_timeout** specifies max time in millis for a socket creation. When doing the initial discovery, and a peer hangs, don't wait forever but go on after the timeout to ping other members. Reduces chances of *not* finding any members at all. The default is 2000.
- **use_send_queues** specifies whether to use separate send queues for each connection. This prevents blocking on write if the peer hangs. The default is true.
- **external_addr** specifies external IP address to broadcast to other group members (if different to local address). This is useful when you have use (Network Address Translation) NAT, e.g. a node on a private network, behind a firewall, but you can only route to it via an externally visible address, which is different from the local address it is bound to. Therefore, the node can be configured to broadcast its external address, while still able to bind to the local one. This avoids having to use the

TUNNEL protocol, (and hence a requirement for a central gossip router) because nodes outside the firewall can still route to the node inside the firewall, but only on its external address. Without setting the external_addr, the node behind the firewall will broadcast its private address to the other nodes which will not be able to route to it.

- **skip_suspected_members** specifies whether unicast messages should not be sent to suspected members. The default is true.
- **tcp_nodelay** specifies TCP_NODELAY. TCP by default nagles messages, that is, conceptually, smaller messages are bundled into larger ones. If we want to invoke synchronous cluster method calls, then we need to disable nagling in addition to disabling message bundling (by setting **`enable_bundling`** to false). Nagling is disabled by setting **`tcp_nodelay`** to true. The default is false.

23.1.2.3. TUNNEL configuration

The TUNNEL protocol uses an external router to send messages. The external router is known as a **`GossipRouter`**. Each node has to register with the router. All messages are sent to the router and forwarded on to their destinations. The TUNNEL approach can be used to setup communication with nodes behind firewalls. A node can establish a TCP connection to the GossipRouter through the firewall (you can use port 80). The same connection is used by the router to send messages to nodes behind the firewall as most firewalls do not permit outside hosts to initiate a TCP connection to a host inside the firewall. The TUNNEL configuration is defined in the TUNNEL element in the JGroups Config element. Here is an example..

```
<TUNNEL router_port="12001"
    router_host="192.168.5.1"
    down_thread="false" up_thread="false/>
```

The available attributes in the **`TUNNEL`** element are listed below.

- **router_host** specifies the host on which the GossipRouter is running.
- **router_port** specifies the port on which the GossipRouter is listening.
- **loopback** specifies whether to loop messages back up the stack. The default is **`true`**.

23.1.3. Discovery Protocols

The cluster needs to maintain a list of current member nodes at all times so that the load balancer and client interceptor know how to route their requests. Discovery protocols are used to discover active nodes in the cluster and detect the oldest member of the cluster, which is the coordinator. All initial nodes are discovered when the cluster starts up. When a new node joins the cluster later, it is only discovered after the group membership protocol (GMS, see *Section 23.1.6, “Group Membership (GMS)”*) admits it into the group.

Since the discovery protocols sit on top of the transport protocol, you can choose to use different discovery protocols based on your transport protocol. These are also configured as sub-elements in the JGroups MBean **`Config`** element.

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=coordinator's address and A=own address. After timeout milliseconds or num_initial_members replies, the joiner determines the coordinator from the responses, and sends a JOIN request to it (handled by). If nobody responds, we assume we are the first member of a group.

Here is an example PING configuration for IP multicast.

```
<PING timeout="2000"
    num_initial_members="2"
    down_thread="false" up_thread="false"/>
```

Here is another example PING configuration for contacting a Gossip Router.

```
<PING gossip_host="localhost"
      gossip_port="1234"
       timeout="3000"
       num_initial_members="3"
       down_thread="false" up_thread="false"/>
```

The available attributes in the **`PING`** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **gossip_host** specifies the host on which the GossipRouter is running.
- **gossip_port** specifies the port on which the GossipRouter is listening on.
- **gossip_refresh** specifies the interval (in milliseconds) for the lease from the GossipRouter. The default is 20000.
- **initial_hosts** is a comma-seperated list of addresses (e.g., **`host1[12345],host2[23456]`**), which are pinged for discovery.

If both **`gossip_host`** and **`gossip_port`** are defined, the cluster uses the GossipRouter for the initial discovery. If the **`initial_hosts`** is specified, the cluster pings that static list of addresses for discovery. Otherwise, the cluster uses IP multicasting for discovery.

Note

The discovery phase returns when the **`timeout`** ms have elapsed or the **`num_initial_members`** responses have been received.

23.1.3.2. TCPGOSSIP

The TCPGOSSIP protocol only works with a GossipRouter. It works essentially the same way as the PING protocol configuration with valid **`gossip_host`** and **`gossip_port`** attributes. It works on top of both UDP and TCP transport protocols. Here is an example.

```
<TCPGOSSIP timeout="2000"
    initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"
    num_initial_members="3"
 down_thread="false" up_thread="false"/>
```

The available attributes in the **`TCPGOSSIP`** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-seperated list of addresses (e.g., **`host1[12345],host2[23456]`**) for GossipRouters to register with.

23.1.3.3. TCPPING

The TCPPING protocol takes a set of known members and ping them for discovery. This is essentially a static configuration. It works on top of TCP. Here is an example of the **`TCPPING`** configuration element in the JGroups **`Config`** element.

```
<TCPPING timeout="2000"
 initial_hosts="hosta[2300],hostb[3400],hostc[4500]"
 port_range="3"
 num_initial_members="3"
        down_thread="false" up_thread="false"/>
```

The available attributes in the **`TCPPING`** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-seperated list of addresses (e.g., **`host1[12345],host2[23456]`**) for pinging.
- **port_range** specifies the number of consecutive ports to be probed when getting the initial membership, starting with the port specified in the initial_hosts parameter. Given the current values of port_range and initial_hosts above, the TCPPING layer will try to connect to hosta:2300, hosta:2301, hosta:2302, hostb:3400, hostb:3401, hostb:3402, hostc:4500, hostc:4501, hostc:4502. The configuration options allows for multiple nodes on the same host to be pinged.

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 TCPPING, which has to list all group members explicitly, but MPING doesn't have this requirement. The typical use case for this is when we want TCP as transport, but multicasting for discovery so we don't have to define a static list of initial hosts in TCPPING or require external Gossip Router.

```
<MPING timeout="2000"
    bind_to_all_interfaces="true"
    mcast_addr="228.8.8.8"
    mcast_port="7500"
    ip_ttl="8"
    num_initial_members="3"
    down_thread="false" up_thread="false"/>
```

The available attributes in the **`MPING`** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2..
- **bind_addr** specifies the interface on which to send and receive multicast packets.
- **bind_to_all_interfaces** overrides the **`bind_addr`** and uses all interfaces in multihome nodes.
- **mcast_addr, mcast_port, ip_ttl** attributes are the same as related attributes in the UDP protocol configuration.

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 after which, if the node is still considered dead, the cluster updates its view so that the load balancer and client interceptors know to avoid the dead node. The failure detection protocols are configured as sub-elements in the JGroups MBean **`Config`** element.

23.1.4.1. FD

FD is a failure detection protocol based on heartbeat messages. This protocol requires each node to periodically send are-you-alive messages to its neighbour. If the neighbour fails to respond, the calling node sends a SUSPECT message to the cluster. The current group coordinator can optionally double check whether the suspected node is indeed dead after which, if the node is still considered dead, updates the cluster's view. Here is an example FD configuration.

```
<FD timeout="2000"
    max_tries="3"
    shun="true"
    down_thread="false" up_thread="false"/>
```

The available attributes in the **FD** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for the responses to the are-you-alive messages. The default is 3000.
- **max_tries** specifies the number of missed are-you-alive messages from a node before the node is suspected. The default is 2.
- **shun** specifies whether a failed node will be shunned. Once shunned, the node will be expelled from the cluster even if it comes back later. The shunned node would have to re-join the cluster through the discovery process. JGroups allows to configure itself such that shunning leads to automatic rejoins and state transfer, which is the default behaivour within JBoss Application Server.

Note

Regular traffic from a node counts as if it is a live. So, the are-you-alive messages are only sent when there is no regular traffic to the node for sometime.

23.1.4.2. FD_SOCK

FD_SOCK is a failure detection protocol based on a ring of TCP sockets created between group members. Each member in a group connects to its neighbor (last member connects to first) thus forming a ring. Member B is suspected when its neighbor A detects abnormally closed TCP socket (presumably due to a node B crash). However, if a member B is about to leave gracefully, it lets its neighbor A know, so that it does not become suspected. The simplest FD_SOCK configuration does not take any attribute. You can just declare an empty **FD_SOCK** element in JGroups's **Config** element.

```
<FD_SOCK_down_thread="false" up_thread="false"/>
```

There available attributes in the **FD_SOCK** element are listed below.

- **bind_addr** specifies the interface to which the server socket should bind to. If -Djgroups.bind_address system property is defined, XML value will be ignore. This behaivour can be reversed setting -Djgroups.ignore.bind_addr=true system property.

23.1.4.3. VERIFY_SUSPECT

This protocol verifies whether a suspected member is really dead by pinging that member once again. This verification is performed by the coordinator of the cluster. The suspected member is dropped from the cluster group if confirmed to be dead. The aim of this protocol is to minimize false suspicions. Here's an example.

```
<VERIFY_SUSPECT timeout="1500"
 down_thread="false" up_thread="false"/>
```

The available attributes in the FD_SOCK 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_SOCK

FD and FD_SOCK, each taken individually, do not provide a solid failure detection layer. Let's look at the the differences between these failure detection protocols to understand how they complement each other:

- *FD*
- An overloaded machine might be slow in sending are-you-alive responses.
- A member will be suspected when suspended in a debugger/profiler.
- Low timeouts lead to higher probability of false suspicions and higher network traffic.
- High timeouts will not detect and remove crashed members for some time.
- *FD_SOCK*:
- Suspended in a debugger is no problem because the TCP connection is still open.
- High load no problem either for the same reason.
- Members will only be suspected when TCP connection breaks
- So hung members will not be detected.
- Also, a crashed switch will not be detected until the connection runs into the TCP timeout (between 2-20 minutes, depending on TCP/IP stack implementation).

The aim of a failure detection layer is to report real failures and therefore avoid false suspicions. There are two solutions:

1. By default, JGroups configures the FD_SOCK socket with KEEP_ALIVE, which means that TCP sends a heartbeat on socket on which no traffic has been received in 2 hours. If a host crashed (or an intermediate switch or router crashed) without closing the TCP connection properly, we would detect this after 2 hours (plus a few minutes). This is of course better than never closing the connection (if KEEP_ALIVE is off), but may not be of much help. So, the first solution would be to lower the timeout value for KEEP_ALIVE. This can only be done for the entire kernel in most operating systems, so if this is lowered to 15 minutes, this will affect all TCP sockets.
2. The second solution is to combine FD_SOCK and FD; the timeout in FD can be set such that it is much lower than the TCP timeout, and this can be configured individually per process. FD_SOCK will already generate a suspect message if the socket was closed abnormally. However, in the case of a crashed switch or host, FD will make sure the socket is eventually closed and the suspect message generated. Example:

```
<FD_SOCK down_thread="false" up_thread="false"/>
<FD timeout="10000" max_tries="5" shun="true"
down_thread="false" up_thread="false" />
```

This suspects a member when the socket to the neighbor has been closed abonormally (e.g. process crash, because the OS closes all sockets). However, f a host or switch crashes, then the sockets won't be closed, therefore, as a seond line of defense, FD will suspect the neighbor after 50 seconds. Note

that with this example, if you have your system stopped in a breakpoint in the debugger, the node you're debugging will be suspected after ca 50 seconds.

A combination of FD and FD_SOCK provides a solid failure detection layer and for this reason, such technique is used accross JGroups configurations included within JBoss Application Server.

23.1.5. Reliable Delivery Protocols

Reliable delivery protocols within the JGroups stack ensure that data pockets are actually delivered in the right order (FIFO) to the destination node. The basis for reliable message delivery is positive and negative delivery acknowledgments (ACK and NAK). In the ACK mode, the sender resends the message until the acknowledgment is received from the receiver. In the NAK mode, the receiver requests retransmission when it discovers a gap.

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. If the JGroups stack is configured with TCP transport protocol, UNICAST is not necessary because TCP itself guarantees FIFO delivery of unicast messages. Here is an example configuration for the **`UNICAST`** protocol.

```
<UNICAST timeout="100,200,400,800"
down_thread="false" up_thread="false"/>
```

There is only one configurable attribute in the **`UNICAST`** element.

- **timeout** specifies the retransmission timeout (in milliseconds). For instance, if the timeout is "100,200,400,800", the sender resends the message if it hasn't received an ACK after 100 ms the first time, and the second time it waits for 200 ms before resending, and so on.

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 keeps track of the sequence numbers and deliver the messages in order. When a gap in the sequence number is detected, the receiver asks the sender to retransmit the missing message. The NAKACK protocol is configured as the **`pbcast.NAKACK`** sub-element under the JGroups **`Config`** element. Here is an example configuration.

```
<pbcast.NAKACK max_xmit_size="60000" use_mcast_xmit="false"

   retransmit_timeout="300,600,1200,2400,4800" gc_lag="0"
   discard_delivered_msgs="true"
   down_thread="false" up_thread="false"/>
```

The configurable attributes in the **`pbcast.NAKACK`** element are as follows.

- **retransmit_timeout** specifies the retransmission timeout (in milliseconds). It is the same as the **`timeout`** attribute in the UNICAST protocol.

- **use_mcast_xmit** determines whether the sender should send the retransmission to the entire cluster rather than just the node requesting it. This is useful when the sender drops the pocket -- so we do not need to retransmit for each node.
- **max_xmit_size** specifies maximum size for a bundled retransmission, if multiple packets are reported missing.
- **discard_delivered_msgs** specifies whether to discard delivery messages on the receiver nodes. By default, we save all delivered messages. However, if we only ask the sender to resend their messages, we can enable this option and discard delivered messages.
- **gc_lag specifies** the number of messages garbage collection lags behind.

23.1.6. Group Membership (GMS)

The group membership service (GMS) protocol in the JGroups stack maintains a list of active nodes. It handles the requests to join and leave the cluster. It also handles the SUSPECT messages sent by failure detection protocols. All nodes in the cluster, as well as the load balancer and client side interceptors, are notified if the group membership changes. The group membership service is configured in the **`pbcast.GMS`** sub-element under the JGroups **`Config`** element. Here is an example configuration.

```
<pbcast.GMS print_local_addr="true"
    join_timeout="3000"
    down_thread="false" up_thread="false"
    join_retry_timeout="2000"
    shun="true"
    view_bundling="true"/>
```

The configurable attributes in the **`pbcast.GMS`** element are as follows.

- **join_timeout** specifies the maximum number of milliseconds to wait for a new node JOIN request to succeed. Retry afterwards.
- **join_retry_timeout** specifies the maximum number of milliseconds to wait after a failed JOIN to re-submit it.
- **print_local_addr** specifies whether to dump the node's own address to the output when started.
- **shun** specifies whether a node should shun itself if it receives a cluster view that it is not a member node.
- **disable_initial_coord** specifies whether to prevent this node as the cluster coordinator.
- **view_bundling** specifies whether multiple JOIN or LEAVE request arriving at the same time are bundled and handled together at the same time, only sending out 1 new view / bundle. This is is more efficient than handling each request separately.

23.1.7. Flow Control (FC)

The flow control (FC) protocol tries to adapt the data sending rate to the data receipt rate 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 receivers 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 protocol is configured in the **FC** sub-element under the JGroups **Config** element. Here is an example configuration.

```
<FC max_credits="1000000"
down_thread="false" up_thread="false"
    min_threshold="0.10"/>
```

The configurable attributes in the **FC** element are as follows.

- **max_credits** specifies the maximum number of credits (in bytes). This value should be smaller than the JVM heap size.
- **min_credits** specifies the threshold credit on the sender, below which the receiver should send in more credits.
- **min_threshold** specifies percentage value of the threshold. It overrides the **min_credits** attribute.

Note

Applications that use synchronous group RPC calls primarily do not require FC protocol in their JGroups protocol stack because synchronous communication, where the hread that makes the call blocks waiting for responses from all the members of the group, already slows overall rate of calls. Even though TCP provides flow control by itself, FC is still required in TCP based JGroups stacks because of group communication, where we essentially have to send group messages at the highest speed the slowest receiver can keep up with. TCP flow control only takes into account individual node communications and has not a notion of who's the slowest in the group, which is why FC is required.

23.1.7.1. Why is FC needed on top of TCP ? TCP has its own flow control !

The reason is group communication, where we essentially have to send group messages at the highest speed the slowest receiver can keep up with. Let's say we have a cluster {A,B,C,D}. D is slow (maybe overloaded), the rest is fast. When A sends a group message, it establishes TCP connections A-A (conceptually), A-B, A-C and A-D (if they don't yet exist). So let's say A sends 100 million messages to the cluster. Because TCP's flow control only applies to A-B, A-C and A-D, but not to A-{B,C,D}, where {B,C,D} is the group, it is possible that A, B and C receive the 100M, but D only received 1M messages. (BTW: this is also the reason why we need NAKACK, although TCP does its own retransmission).

Now JGroups has to buffer all messages in memory for the case when the original sender S dies and a node asks for retransmission of a message of S. Because all members buffer all messages they received, they need to purge stable messages (= messages seen by everyone) every now and then. This is done by the STABLE protocol, which can be configured to run the stability protocol round time based (e.g. every 50s) or size based (whenever 400K data has been received).

In the above case, the slow node D will prevent the group from purging messages above 1M, so every member will buffer 99M messages ! This in most cases leads to OOM exceptions. Note that - although the sliding window protocol in TCP will cause writes to block if the window is full - we assume in the above case that this is still much faster for A-B and A-C than for A-D.

So, in summary, we need to send messages at a rate the slowest receiver (D) can handle.

23.1.7.2. So do I always need FC?

This depends on how the application uses the JGroups channel. Referring to the example above, if there was something about the application that would naturally cause A to slow down its rate of sending because D wasn't keeping up, then FC would not be needed.

A good example of such an application is one that makes synchronous group RPC calls (typically using a JGroups RpcDispatcher.) By synchronous, we mean the thread that makes the call blocks waiting for responses from all the members of the group. In that kind of application, the threads on A that are making calls would block waiting for responses from D, thus naturally slowing the overall rate of calls.

A JBoss Cache cluster configured for REPL_SYNC is a good example of an application that makes synchronous group RPC calls. If a channel is only used for a cache configured for REPL_SYNC, we recommend you remove FC from its protocol stack.

And, of course, if your cluster only consists of two nodes, including FC in a TCP-based protocol stack is unnecessary. There is no group beyond the single peer-to-peer relationship, and TCP's internal flow control will handle that just fine.

Another case where FC may not be needed is for a channel used by a JBoss Cache configured for buddy replication and a single buddy. Such a channel will in many respects act like a two node cluster, where messages are only exchanged with one other node, the buddy. (There may be other messages related to data gravitation that go to all members, but in a properly engineered buddy replication use case these should be infrequent. But if you remove FC be sure to load test your application.)

23.1.8. Fragmentation (FRAG2)

This protocol fragments messages larger than certain size. Unfragments at the receiver's side. It works for both unicast and multicast messages. It is configured in the FRAG2 sub-element under the JGroups Config element. Here is an example configuration.

```
<FRAG2 frag_size="60000" down_thread="false" up_thread="false"/>
```

The configurable attributes in the FRAG2 element are as follows.

- **frag_size** specifies the max frag size in bytes. Messages larger than that are fragmented.

Note

TCP protocol already provides fragmentation but a fragmentation JGroups protocol is still needed if FC is used. The reason for this is that if you send a message larger than FC.max_bytes, FC protocol would block. So, frag_size within FRAG2 needs to be set to always be less than FC.max_bytes.

23.1.9. State Transfer

The state transfer service transfers the state from an existing node (i.e., the cluster coordinator) to a newly joining node. It is configured in the **`pbcast.STATE_TRANSFER`** sub-element under the JGroups **`Config`** element. It does not have any configurable attribute. Here is an example configuration.

```
<pbcast.STATE_TRANSFER down_thread="false" up_thread="false"/>
```

23.1.10. Distributed Garbage Collection (STABLE)

In a JGroups cluster, all nodes have to store all messages received for potential retransmission in case of a failure. However, if we store all messages forever, we will run out of memory. So, the distributed garbage collection service in JGroups periodically purges messages that have seen by all nodes from the memory in each node. The distributed garbage collection service is configured in the **`pbcast.STABLE`** sub-element under the JGroups **`Config`** element. Here is an example configuration.

```
<pbcast.STABLE stability_delay="1000"
    desired_avg_gossip="5000"
    down_thread="false" up_thread="false"
       max_bytes="400000"/>
```

The configurable attributes in the **`pbcast.STABLE`** element are as follows.

- **desired_avg_gossip** specifies intervals (in milliseconds) of garbage collection runs. Value **`0`** disables this service.
- **max_bytes** specifies the maximum number of bytes received before the cluster triggers a garbage collection run. Value **`0`** disables this service.
- **stability_delay** specifies delay before we send STABILITY msg (give others a change to send first). If used together with max_bytes, this attribute should be set to a small number.

Note

Set the **`max_bytes`** attribute when you have a high traffic cluster.

23.1.11. Merging (MERGE2)

When a network error occurs, the cluster might be partitioned into several different partitions. JGroups has a MERGE service that allows the coordinators in partitions to communicate with each other and form a single cluster back again. The flow control service is configured in the **`MERGE2`** sub-element under the JGroups **`Config`** element. Here is an example configuration.

```
<MERGE2 max_interval="10000"
    min_interval="2000"
    down_thread="false" up_thread="false"/>
```

The configurable attributes in the **FC** element are as follows.

- **max_interval** specifies the maximum number of milliseconds to send out a MERGE message.
- **min_interval** specifies the minimum number of milliseconds to send out a MERGE message.

JGroups chooses a random value between **min_interval** and **max_interval** to send out the MERGE message.

Note

The cluster states are not merged in a merger. This has to be done by the application. If **MERGE2** is used in conjunction with TCPPING, the **initial_hosts** attribute must contain all the nodes that could potentially be merged back, in order for the merge process to work properly. Otherwise, the merge process would not merge all the nodes even though shunning is disabled. Alternatively use MPING, which is commonly used with TCP to provide multicast member discovery capabilities, instead of TCPPING to avoid having to specify all the nodes.

23.2. Other Configuration Issues

23.2.1. Binding JGroups Channels to a particular interface

In the Transport Protocols section above, we briefly touched on how the interface to which JGroups will bind sockets is configured. Let's get into this topic in more depth:

First, it's important to understand that the value set in any bind_addr element in an XML configuration file will be ignored by JGroups if it finds that system property jgroups.bind_addr (or a deprecated earlier name for the same thing, **bind.address**) has been set. The system property trumps XML. If JBoss Enterprise Application Platform is started with the -b (a.k.a. --host) switch, the Enterprise Application Platform will set **jgroups.bind_addr** to the specified value.

Beginning with Enterprise Application Platform 4.2.0, for security reasons the Enterprise Application Platform will bind most services to localhost if -b is not set. The effect of this is that in most cases users are going to be setting -b and thus jgroups.bind_addr is going to be set and any XML setting will be ignored.

So, what are *best practices* for managing how JGroups binds to interfaces?

- Binding JGroups to the same interface as other services. Simple, just use -b:

```
./run.sh -b 192.168.1.100 -c all
```

- Binding services (e.g., JBoss Web) to one interface, but use a different one for JGroups:

```
./run.sh -b 10.0.0.100 -Djgroups.bind_addr=192.168.1.100 -c all
```

  Specifically setting the system property overrides the -b value. This is a common usage pattern; put client traffic on one network, with intra-cluster traffic on another.

- Binding services (e.g., JBoss Web) to all interfaces. This can be done like this:

```
./run.sh -b 0.0.0.0 -c all
```

However, doing this will not cause JGroups to bind to all interfaces! Instead , JGroups will bind to the machine's default interface. See the Transport Protocols section for how to tell JGroups to receive or send on all interfaces, if that is what you really want.

- Binding services (e.g., JBoss Web) to all interfaces, but specify the JGroups interface:

```
./run.sh -b 0.0.0.0 -Djgroups.bind_addr=192.168.1.100 -c all
```

Again, specifically setting the system property overrides the -b value.

- Using different interfaces for different channels:

```
./run.sh -b 10.0.0.100 -Djgroups.ignore.bind_addr=true -c all
```

This setting tells JGroups to ignore the **`jgroups.bind_addr`** system property, and instead use whatever is specfied in XML. You would need to edit the various XML configuration files to set the **`bind_addr`** to the desired interfaces.

23.2.2. Isolating JGroups Channels

Within JBoss Enterprise Application Platform, there are a number of services that independently create JGroups channels -- 3 different JBoss Cache services (used for HttpSession replication, EJB3 SFSB replication and EJB3 entity replication) along with the general purpose clustering service called HAPartition that underlies most other JBossHA services.

It is critical that these channels only communicate with their intended peers; not with the channels used by other services and not with channels for the same service opened on machines not meant to be part of the group. Nodes improperly communicating with each other is one of the most common issues users have with JBoss Enterprise Application Platform clustering.

Whom a JGroups channel will communicate with is defined by its group name, multicast address, and multicast port, so isolating JGroups channels comes down to ensuring different channels use different values for the group name, multicast address and multicast port.

To isolate JGroups channels for different services on the same set of Enterprise Application Platform instances from each other, you MUST change the group name and the multicast port. In other words, each channel must have its own set of values.

For example, say we have a production cluster of 3 machines, each of which has an HAPartition deployed along with a JBoss Cache used for web session clustering. The HAPartition channels should not communicate with the JBoss Cache channels. They should use a different group name and multicast port. They can use the same multicast address, although they don't need to.

To isolate JGroups channels for the same service from other instances of the service on the network, you MUST change ALL three values. Each channel must have its own group name, multicast address, and multicast port.

For example, say we have a production cluster of 3 machines, each of which has an HAPartition deployed. On the same network there is also a QA cluster of 3 machines, which also has an

HAPartition deployed. The HAPartition group name, multicast address, and multicast port for the production machines must be different from those used on the QA machines.

23.2.2.1. Changing the Group Name

The group name for a JGroups channel is configured via the service that starts the channel. Unfortunately, different services use different attribute names for configuring this. For HAPartition and related services configured in the deploy/cluster-service.xml file, this is configured via a PartitionName attribute. For JBoss Cache services, the name of the attribute is ClusterName.

The HAPartition and all the standard JBoss Cache services, make it easy for you to create unique groups names simply by using the -g (a.k.a. –partition) switch when starting JBoss:

```
./run.sh -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the jboss.partition.name system property, which is used as a component in the configuration of the group name in all the standard clustering configuration files. For example,

```
<attribute name="ClusterName">Tomcat-${jboss.partition.name:Cluster}</
attribute>
```

23.2.2.2. Changing the multicast address and port

The -u (a.k.a. --udp) command line switch may be used to control the multicast address used by the JGroups channels opened by all standard Enterprise Application Platform services.

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the jboss.partition.udpGroup system property, which you can see referenced in all of the standard protocol stack configs in JBoss Enterprise Application Platform:

```
<Config>
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
 ....
```

Unfortunately, setting the multicast ports is not so simple. As described above, by default there are four separate JGroups channels in the standard JBoss Enterprise Application Platform all configuration, and each should be given a unique port. There are no command line switches to set these, but the standard configuration files do use system properties to set them. So, they can be configured from the command line by using -D. For example,

```
 /run.sh -u 230.1.2.3 -g QAPartition -Djboss.hapartition.mcast_port=12345 -
Djboss.webpartition.mcast_port=23456 -
Djboss.ejb3entitypartition.mcast_port=34567 -
Djboss.ejb3sfsbpartition.mcast_port=45678 -b 192.168.1.100 -c all
```

Why isn't it sufficient to change the group name?

If channels with different group names share the same multicast address and port, the lower level JGroups protocols in each channel will see, process and eventually discard messages intended for the other group. This will at a minimum hurt performance and can lead to anomalous behavior.

Why do I need to change the multicast port if I change the address?

It should be sufficient to just change the address, but there is a problem on several operating systems whereby packets addressed to a particular multicast port are delivered to all listeners on that port, regardless of the multicast address they are listening on. So the recommendation is to change both the address and the port.

23.3. JGroups Troubleshooting

23.3.1. Nodes do not form a cluster

Make sure your machine is set up correctly for IP multicast. There are 2 test programs that can be used to detect this: McastReceiverTest and McastSenderTest. Go to the **`$JBOSS_HOME/server/all/lib`** directory and start McastReceiverTest, for example:

```
java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -mcast_addr
 224.10.10.10 -port 5555
```

Then in another window start **`McastSenderTest`**:

```
java -cp jgroups.jar org.jgroups.tests.McastSenderTest -mcast_addr
 224.10.10.10 -port 5555
```

If you want to bind to a specific network interface card (NIC), use **`-bind_addr 192.168.0.2`**, where 192.168.0.2 is the IP address of the NIC to which you want to bind. Use this parameter in both the sender and the receiver.

You should be able to type in the **`McastSenderTest`** window and see the output in the **`McastReceiverTest`** window. If not, try to use -ttl 32 in the sender. If this still fails, consult a system administrator to help you setup IP multicast correctly, and ask the admin to make sure that multicast will work on the interface you have chosen or, if the machines have multiple interfaces, ask to be told the correct interface. Once you know multicast is working properly on each machine in your cluster, you can repeat the above test to test the network, putting the sender on one machine and the receiver on another.

23.3.2. Causes of missing heartbeats in FD

Sometimes a member is suspected by FD because a heartbeat ack has not been received for some time T (defined by timeout and max_tries). This can have multiple reasons, e.g. in a cluster of A,B,C,D; C can be suspected if (note that A pings B, B pings C, C pings D and D pings A):

- B or C are running at 100% CPU for more than T seconds. So even if C sends a heartbeat ack to B, B may not be able to process it because it is at 100%
- B or C are garbage collecting, same as above.
- A combination of the 2 cases above

- The network loses packets. This usually happens when there is a lot of traffic on the network, and the switch starts dropping packets (usually broadcasts first, then IP multicasts, TCP packets last).
- B or C are processing a callback. Let's say C received a remote method call over its channel and takes T+1 seconds to process it. During this time, C will not process any other messages, including heartbeats, and therefore B will not receive the heartbeat ack and will suspect C.

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 Application 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 Application Platform provides. We then discuss the different options available for deploying a custom cache in the Enterprise Application Platform.

Users considering deploying JBoss Cache for direct use by their own application are strongly encouraged to read the JBoss Cache documentation available at http://www.jboss.org/jbosscache.

See also *Section 16.2, "Distributed Caching with JBoss Cache"* for information on how the standard JBoss Enterprise Application Platform clustered services use JBoss Cache.

24.1. Key JBoss Cache Configuration Options

JBoss Enterprise Application Platform ships with a reasonable set of default JBoss Cache configurations that are suitable for the standard clustered service use cases (e.g. 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.jboss.org/jbosscache.

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 Application Platform CacheManager service uses the JBoss Microcontainer schema to be consistent with most other internal Enterprise Application 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 16.2.1, "The JBoss Enterprise Application Platform CacheManager Service"*, the standard JBoss Enterprise Application 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 (i.e. cache loading, eviction, buddy replication). Rather than delegating this task of XML parsing/Java Bean creation to JBC, we let the Enterprise Application Platform's microcontainer do it directly. This has the advantage of making the microcontainer aware of the configuration beans, which in later Enterprise Application Platform 5.x 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 Application 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>

   <!-- 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 java beans 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** means the cache instance sends a message to its peers notifying them of the change(s) and before returning waits for them to acknowledge that they have applied the same changes. If the changes are made as part of a JTA transaction, this is done as part of a 2 phase-commit process during transaction commit. Any locks are held until this acknowledgment is received. Waiting for acknowledgement from all nodes adds delays, but it ensures consistency around the cluster. Synchronous mode is needed when all the nodes in the cluster may access the cached data resulting in a high need for consistency.

- **Asynchronous** means the cache instance sends a message to its peers notifying them of the change(s) and then immediately returns, without any acknowledgement that they have applied the same changes. It *does not* mean sending the message is handled by some other thread besides the one that changed the cache content; the thread that makes the change still spends some time dealing with sending messages to the cluster, just not as much as with synchronous communication. Asynchronous mode is most useful for cases like session replication, where the cache doing the sending expects to be the only one that accesses the data and the cluster messages are used to provide backup copies in case of failure of the sending node. Asynchronous messaging adds a small risk that a later user request that fails over to another node may see out-of-date state, but for many session-type applications this risk is acceptable given the major performance benefits asynchronous mode has over synchronous mode.

- **Local** means the cache instance doesn't send a message at all. A JGroups channel isn't even used by the cache. JBoss Cache has many useful features besides its clustering capabilities and is a very useful caching library even when not used in a cluster. Also, even in a cluster, some cached data does not need to be kept consistent around the cluster, in which case Local mode will improve performance. Caching of JPA/Hibernate query result sets is an example of this; Hibernate's second level caching logic uses a separate mechanism to invalidate stale query result sets from the second level cache, so JBoss Cache doesn't need to send messages around the cluster for a query result set cache.

Replication vs. Invalidation

This aspect deals with the content of messages sent around the cluster when a cache changes its local state, i.e. what should the other caches in the cluster do to reflect the change:

- **Replication** means the other nodes should update their state to reflect the new state on the sending node. This means the sending node needs to include the changed state, increasing the cost of the message. Replication is necessary if the other nodes have no other way to obtain the state.
- **Invalidation** means the other nodes should remove the changed state from their local state. Invalidation reduces the cost of the cluster update messages, since only the cache key of the changed state needs to be transmitted, not the state itself. However, it 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 the cached state can be re-read from the database.

These two aspects combine to form 5 valid values for the **`cacheMode`** configuration attribute:

- **LOCAL** means no cluster messages are needed.
- **REPL_SYNC** means synchronous replication messages are sent.
- **REPL_ASYNC** means asynchronous replication messages are sent.
- **INVALIDATION_SYNC** means synchronous invalidation messages are sent.
- **INVALIDATION_ASYNC** means 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 Application Platform, this attribute would have one of two values:

- **org.jboss.cache.transaction.JBossTransactionManagerLookup**

  This finds 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**

This is used in the cache configurations used for web session and EJB SFSB caching. It specifies a simple mock **`TransactionManager`** that ships with JBoss Cache called the **`BatchModeTransactionManager`**. This transaction manager is not a true JTA transaction manager and should not be used for anything other than JBoss Cache. Its usage in JBoss Enterprise Application Platform is to get most of the benefits of JBoss Cache's transactional behavior for the session replication use cases, but without getting tangled up with end user transactions that may run during a request.

Note

For caches used for JPA/Hibernate caching, the **`transactionManagerLookupClass`** should not be configured. Hibernate internally configures the cache to use the same transaction manager it is using 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 choices for **`nodeLockingScheme`**:

- **MVCC** or multi-versioned concurrency control, is 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 don't block writers
  - Writers that fail fast

  It achieves this by using data versioning and copying for concurrent writers. The theory is that readers continue reading shared state, while writers copy the shared state, increment a version id, and write that shared state back after verifying that the version is still valid (i.e., another concurrent writer has not changed this state first).

  MVCC is the recommended choice for JPA/Hibernate entity caching.

- **PESSIMISTIC** locking involves threads/transactions acquiring either exclusive or non-exclusive locks on nodes before reading or writing. Which is acquired depends on the **`isolationLevel`** (see below) 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 considerably more overhead than MVCC and allows lesser concurrency, since reader threads must block until a write has completed and released its exclusive lock (potentially a long time if the write is part of a transaction). A write will also be delayed due to ongoing reads.

  Generally MVCC is a better choice than PESSIMISTIC, which is deprecated as of JBoss Cache 3.0. But, for the session caching usage in JBoss Enterprise Application Platform 5.0.0, PESSIMISTIC is still the default. This is largely because for the session use case there are generally not concurrent threads accessing the same cache location, so the benefits of MVCC are not as great.

- **OPTIMISTIC** locking seeks to improve upon the concurrency available with PESSIMISTIC by creating a "workspace" for each request/transaction that accesses the cache. Data accessed by the

request/transaction (even reads) is *copied* into the workspace, which is adds overhead. All data is versioned; on completion of non-transactional requests or commits of transactions the version of data in the workspace is compared to the main cache, and an exception is raised if there are are inconsistencies. Otherwise changes to the workspace are applied to the main cache.

OPTIMISTIC locking is deprecated but is still provided to support backward compatibility. Users are encouraged to use MVCC instead, which provides the same benefits at lower cost.

The **`isolationLevel`** attribute has two possible values **READ_COMMITTED** and **REPEATABLE_READ** which correspond in semantic to database-style isolation levels. Previous versions of JBoss Cache supported all 5 database isolation levels, and if an unsupported isolation level is configured, it is either upgraded or downgraded to the closest supported level.

REPEATABLE_READ is the default isolation level, to maintain compatibility with previous versions of JBoss Cache. READ_COMMITTED, while providing a slightly weaker isolation, has a significant performance benefit over REPEATABLE_READ.

24.1.5. JGroups Integration

Each JBoss Cache instance internally uses a JGroups **`Channel`** to handle group communications. Inside JBoss Enterprise Application Platform, we strongly recommend that you use the Enterprise Application Platform's JGroups Channel Factory service as the source for your cache's **`Channel`**. In this section we discuss how to configure your cache to get it's 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 has 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.:

```
<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 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-jbosscache-guide-3.pdf. For web session caches, eviction should not be configured; the distributable session manager handles eviction itself. For EJB 3 SFSB caches, stick with the eviction configuration in the Enterprise Application Platform's standard **sfsb-cache** configuration (see *Section 16.2.1, "The JBoss Enterprise Application 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 just 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 passivaton.

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 Application 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** config 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">
${jboss.server.data.dir}${/}field-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>
```

Some explanation:

- **passivation** property MUST be **true**
- **shared** property MUST be **false**. Do not passivate sessions to a shared persistent store, otherwise if another node activates the session, it will be gone from the persistent store and also gone from memory on other nodes that have passivated it. Backup copies will be lost.
- **individualCacheLoaderConfigs** property accepts a list of Cache Loader configurations. JBC allows you to chain cache loaders; see the JBoss Cache docs. For the session passivation use case a single cache loader is sufficient.
- **class** attribute on a cache loader config bean must refer to the configuration class for a cache loader implementation (e.g. **org.jboss.cache.loader.FileCacheLoaderConfig** or **org.jboss.cache.loader.JDBCCacheLoaderConfig**). See the JBoss Cache documentation for more on the available CacheLoader implementations. If you wish to use JDBCCacheLoader (to persist to a database rather than the filesystem used by FileCacheLoader) note the comment above about the **shared** property. Don't use a shared database, or at least not a shared table in the database. Each node in the cluster must have its own storage location.
- **location** property for FileCacheLoaderConfig defines the root node of the filesystem tree where passivated sessions should be stored. The default is to store them in your JBoss Enterprise Application 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`** as a minor performance optimization.

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 occurence; 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 (a.k.a. "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 (e.g. 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, it will if possible 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`** and **`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 Application 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 Application Platform's CacheManager service (see *Section 16.2.1, "The JBoss Enterprise Application Platform CacheManager Service"*). End user applications can do the same thing; here's how.

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>
      .....
```

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 the CacheManager to your application. 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 find look up the CacheManger is 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 Application 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 Application 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.releasePojoCache("my-cache-config");
   }
}
```

24.2.2. Deployment Via a `-service.xml` File

As in JBoss 4, you can also deploy a JBoss Cache instance as an MBean service via a **`-service.xml`** file. The primary difference from JBoss 4 is the value of the **`code`** attribute in the **`mbean`** element. In JBoss 4, this was **`org.jboss.cache.TreeCache`**; in JBoss 5 it is **`org.jboss.cache.jmx.CacheJmxWrapper`**. Here's an example:

```
<?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 (i.e. you can't store stuff in it). Rather, as it's 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 Application Platform 5 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** dir, 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 Application 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=ExampleC

 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 Application Platform registering the bean in JXM 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

JBoss Enterprise Application Platform 5 Performance Tuning

25.1. Introduction

Developing applications and deploying them to an application server does not guarantee best performance without performance tuning of the applications and server. Performance tuning involves ensuring your application does not consume resources unnecessarily while ensures best performance of the applications and application server.

Application design, hardware/network profile, operating system, application software development, testing and deployment all play a major role in performance tuning. A bottleneck in performance therefore could be caused by these factors not just your application. Recent studies show that most performance problems are the result of the applications not the middleware or the operating systems. This could be associated with the technological developments in computer software, hardware and networking which has increased their reliability.

Improvement of application design and undertaking performance review of your applications before implementation is vital to avoiding bottlenecks after implementation. To undertake a performance review you need to setup a test environment undertake and analyse the test results. To effectively undertake a review, you also need to identify peak application workload times and the difference from normal workload periods. Peak workload times could be during the day, week, certain periods of the month, quarter or year. In understanding peaks workloads it is advisable not to go by averages as the peaks may be much more than the averages calculated over a period. The system requirements are bound by the peaks in the workload not the averages. On undertaking tuning it is recommended to carry out a few more tests and tuning of your system until a satisfactory performance is achieved.

25.2. Hardware tuning

To develop a suitable hardware configuration that suits the performance of your applications on the JBoss Enterprise Application Platform, you need to understand the impact 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 which 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 be helpful in identifying your CPU specifications and how it works. For AMD CPU's please refer to *http://www.amd.com/us-en/Processors/*

ProductInformation/0,,30_118,00.html for more information. For Intel CPU's please refer to *http://www.intel.com/products/processor/index.htm?iid=subhdr+prod_proc* for more information.

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 provides a higher amount of storage than the CPU cache and can improve computer performance. Storing data or programs frequently used in RAM can highly 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 would involve storing frequently used database information in RAM for quick application access while taking caution not to affect overall performance of other applications and operating system.

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.

However 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 realtime.

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 realtime where possible or with system rollouts

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 dialup 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 obviosly 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 while 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 acknowledgement options among others should be take into account in your network design.

Deciding to turn DNS lookups 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 5 documentation on *http://java.sun.com/j2se/1.5.0/docs/* for further discussions on this.

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 startup or other factors.

The JBoss Application Server web console *http://localhost:8080/web-console/* provides you with monitoring tools starting with the JVM Hardware environment statistics on the default page and access to monitoring tools and snapshots.

Performance Monitor v/s Profiler

A performance monitor informs you on overall application performance such as requests per second. Profiling tools such as *JBoss Profiler*[1] 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 to take caution not to create bottlenecks for other applications while fixing a performance issue in one application.

25.5.1. Instrumentation

Applications should always be instrumented for performance analysis. In most cases, it is evident that performance requirements and peak workloads examined before production are incorrect compared to during production. 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.

Instrumentation in the past would have 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.

25.6. Tuning JBoss Application Server

Before tuning the JBoss Enterprise Application Platform, please ensure that you are familiar with its components outlined in the introduction section of this book. You should also be familiar with any particular services your application may use on the application 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 application server. This section discusses these among other JBoss Application Server performance tuning topics.

25.6.1. Memory usage

Memory usage of Java applications including the JBoss Enterprise Application Platform is dictated by the heap space allocated. You could therefore as an example, reduce 1GB heap space you currently have allocated to 800MB to save space.

There are several instances where the Java Virtual Machine (JVM) may report **`OutOfMemoryError`** even when it is not really out of its available memory. The JVM may report an out of memory error when it is really out of memory or when only a segment or generation of the heap space is exhausted as most modern JVM's divide the heap space into generations/segments. Another example could be inability of the operating system (occurs on Linux/Unix systems)to create new threads for the JVM.

Running out of memory generates an Error that is not likely to be masked in a catch block because it is an Error rather than an Exception. This is important since one often sees theories expressed about OutOfMemoryError being reported erroneously. That is very unlikely, although OOMEs do occur when the heap has plenty of memory or plenty of recoverable memory. An OOME is also thrown when the permanent memory is exhausted and that is not part of the heap per se. That is a JVM specific area

of memory where information on loaded classes is maintained. If you have a mountain of classes (e.g, a lot of EJBs and JSP pages) you can easily exhaust this area. Oftentimes an application will fail to deploy or fail to redeploy. Increase your permanent memory space as follows to avoid OOMEs. The default with the **`-server`** switch is 64 megabytes:

```
-XX:MaxPermSize?=128m
```

Note this is in addition to the heap. In this case we have 512M heap, 128M permanent space for a total of 640 megabytes. Don't forget the JVM itself takes up a chunk of system memory and there is also two megs per thread of stack space. That can add up with a lot of HTTP/S processors.

`-XX:MaxPermSize?=128m -Xmx512m` (total of 640 megabytes allocated from system - 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)

On Windows, you can set this in the **`<JBOSS_HOME>\bin\run.bat`** file by setting

```
JAVA_OPTS=%JAVA_OPTS% -Xms128m -Xmx512m -XX:MaxPermSize?=128m
```

The HotSpot Java Virtual Machine ships with J2SE 1.4.2 and above and 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/docs/hotspot/gc1.4.2/*.

The *jvmstat toolkit*[2] is recommended for the Hotspot JVM and can help give you a precise picture of your permanent memory space and the other segments on the heap. Please visit the link above for more information on the toolkit.

25.6.1.1. VFS Tuning

25.6.1.1.1. VFS Cache Tuning

Magic about the caching settings...

25.6.1.1.2. Annotation Scaning Tuning

Magic about controlling annotations scaning...

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 read's and writes.

Resource limits set by your operating system may also set limits on your database management system. A database administrator can analyse a database and identify performance bottlenecks through 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.

[2] http://java.sun.com/performance/jvmstat/

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 Application 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 Application Platform and utilizing the connection pool settings. Connection pooling in the JBoss Enterprise Application 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/<your_configuration>/deploy/
```

The filename should be in the following formats:

```
<yourdatabasename>-ds.xml
```

Note

Please note that the name of the file must end with **`-ds.xml`** in order for the JBoss application server to recognize it as a *data source file*. The Hypersonic database data source file for example is named **`hsqldb-ds.xml`**.

The example below is a sample Hypersonic database data source file. Please note that this file contains more comments or descriptions for the respective tags. For a full view of this file, and its comments, please refer to the **`hsqldb-ds.xml`** in the **`deploy`** directory of your configuration.

More examples

More examples of datasource definition files for supported external databases can be found in the **`<JBoss_Home>/docs/examples/jca`** directory.

```
...
<datasources>
   <local-tx-datasource>

 <!-- The jndi name of the DataSource, it is prefixed with java:/ -->
 <!-- Datasources are not available outside the virtual machine -->
 <jndi-name>DefaultDS</jndi-name>

 <!-- For server mode db, allowing other processes to use hsqldb over tcp.
 This requires the org.jboss.jdbc.HypersonicDatabase mbean.
 <connection-url>jdbc:hsqldb:hsql://${jboss.bind.address}:1701</connection-
url>
 -->
 <!-- For totally in-memory db, not saved when jboss stops.
```

```
 The org.jboss.jdbc.HypersonicDatabase mbean is required for proper db
 shutdown
 <connection-url>jdbc:hsqldb:.</connection-url>
 -->
 <!-- For in-process persistent db, saved when jboss stops.
 The org.jboss.jdbc.HypersonicDatabase mbean is required for proper db
 shutdown
 -->
 <connection-url>jdbc:hsqldb:${jboss.server.data.dir}${/}hypersonic
${/}localDB</connection-url>

 <!-- The driver class -->
 <driver-class>org.hsqldb.jdbcDriver</driver-class>

 <!-- The login and password. Do not enter plain text for production
 databases. Please see Security section for more information -->
 <user-name>sa</user-name>
 <password></password>

 <!--example of how to specify class that determines if exception means
 connection should be destroyed-->
 <!--exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.DummyExceptionSorter</
exception-sorter-class-name-->

 <!-- this will be run before a managed connection is removed from the pool
 for use by a client-->
 <!--<check-valid-connection-sql>select * from something</check-valid-
connection-sql> -->

 <!-- The minimum database connections managed in a pool/sub-pool. Pools
 are lazily constructed on first use -->
 <min-pool-size>5</min-pool-size>

 <!-- The maximum database connections managed in a pool/sub-pool -->
 <max-pool-size>20</max-pool-size>

 <!-- The time before an unused connection is destroyed -->
 <!-- NOTE: This is the check period. It will be destroyed somewhere
 between 1x and 2x this timeout after last use -->
 <!-- TEMPORARY FIX! - Disable idle connection removal, HSQLDB has a
 problem with not reaping threads on closed connections -->
 <idle-timeout-minutes>0</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>
 -->

 <!-- example of how to specify a class that determines a connection is
 valid before it is handed out from the pool
 <valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.DummyValidConnectionChecker</
valid-connection-checker-class-name>
 -->

 <!-- Whether to check all statements are closed when the connection is
 returned to the pool,
 this is a debugging feature that should be turned off in production -->
 <track-statements/>

 <!-- Use the getConnection(user, pw) for logins
 <application-managed-security/>
 -->

 <!-- Use the security domain defined in conf/login-config.xml -->
 <security-domain>HsqlDbRealm</security-domain>

 <!-- Use the security domain defined in conf/login-config.xml or the
 getConnection(user, pw) for logins. The security domain takes precedence.
 <security-domain-and-application>HsqlDbRealm</security-domain-and-
application>
 -->

 <!-- HSQL DB benefits from prepared statement caching which stores recent
 prepared statements for future use. The prepared-statement-cache-size
 indicates the number of prepared statements to store in the cache. -->
 <prepared-statement-cache-size>32</prepared-statement-cache-size>

 <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
 (optional) -->
 <metadata>
 <type-mapping>Hypersonic SQL</type-mapping>
 </metadata>

 <!-- When using in-process (standalone) mode -->
 <depends>jboss:service=Hypersonic,database=localDB</depends>
 <!-- Uncomment when using hsqldb in server mode
 <depends>jboss:service=Hypersonic</depends>
 -->
 </local-tx-datasource>

 <!-- Uncomment if you want hsqldb accessed over tcp (server mode)
 <mbean code="org.jboss.jdbc.HypersonicDatabase"
 name="jboss:service=Hypersonic">
 <attribute name="Port">1701</attribute>
```

```
 <attribute name="BindAddress">${jboss.bind.address}</attribute>
 <attribute name="Silent">true</attribute>
 <attribute name="Database">default</attribute>
 <attribute name="Trace">false</attribute>
 <attribute name="No_system_exit">true</attribute>
 </mbean>
 -->

 <!-- For hsqldb accessed from jboss only, in-process (standalone) mode --
>
   <mbean code="org.jboss.jdbc.HypersonicDatabase"
 name="jboss:service=Hypersonic,database=localDB">
 <attribute name="Database">localDB</attribute>
 <attribute name="InProcessMode">true</attribute>
</mbean>

</datasources>
```

25.6.3. Other key configurations

Other key configurations required for performance tuning of your application server include the **<JBoss_Home>/server/<your_configuration>/deployers/jbossweb.deployer/server.xml** file that sets your HTTP requests pool.

JBoss Enterprise Application 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/<your_configuration>/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 **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. Index

Index

A

AOP (see JBoss AOP)

C

Configuration
  databases, 105

D

DataSource
  deployment type, 13

E

EAR (see Enterprise Application)
Enterprise Application
  deployment type, 13
Exploded Deployment, 14

F

Frequently Asked Questions, 129

H

Hot deployment
  disabling, 9
  implementation, 9

J

JAX-WS (see Web Services)
JBoss AOP
  applying aspects, 87
  aspect oriented framework, 85
  creating aspects, 87
JBoss Enterprise Application Platform
  architecture, 5
  bootstrap, 9
  JMX Microkernel, 1
  microcontainer, 1
  performance tuning, 257
  Server interface implementation, 9
  transactions, 89
JBoss Enterprise Application Platform 5
Performance Tuning
  performance, 257
JBoss Messaging
  about, 95
JBoss Microcontainer
  *-jboss-beans.xml deployment type, 13
  beans deployment type, 13
  project modules, 16
JBoss Transactions, 89
JBoss5 Virtual Deployment Framework
  virtual deployment framework, 77
JBossWS
  Web Services, 19
JMS (see JBoss Messaging)

M

MC (see JBoss Microcontainer)

P

Performance
  JBoss Enterprise Application Platform 5
  Performance Tuning, 257
Pooling
  JBossJCA, 125
Profiles
  all, 14
  default, 14
  minimal, 14
  standard, 14
  web, 14
ProfileService
  bootstrap, 9

R

Remoting
  about, 93

S

SAR (see Service Archive)
Server Configuration (see Server Profile)
Server Profile
  definition, 14
Service Archive
  *-service.xml deployment type, 13
  deployment type, 13

V

Virtual Deployment Framework (see JBoss5 Virtual Deployment Framework)

W

WAR (see Web Application)
Web Application
  deployment type, 13
Web Services
  web services, 19