JBoss Enterprise Application Platform Common Criteria Certification 5

Administration And Configuration Guide

for JBoss Enterprise Application Platform 5 Common Criteria Certification Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5 Administration And Configuration Guide

for JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

JBoss Community

Edited by

JBoss Community

Red Hat Documentation Group

Legal Notice

Abstract

This book is a guide to the administration and configuration of JBoss Enterprise Application Platform 5.1.0.

Table of Contents

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.

CHAPTER 1. 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 8, *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.

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 included in 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 JBoss Enterprise Application Platform 4.2.
- JBoss Cache comes in two flavors: a traditional tree-structured node-based cache, and a PojoCache, an in-memory, transactional, and replicated cache system that allows users to operate on simple POJOs transparently without active user management of either replication or persistency aspects.
- JBossWS 3.x is the web services stack for JBoss Enterprise Application Platform 5 providing Java EE compatible web services, JAXWS-2.x.
- 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.

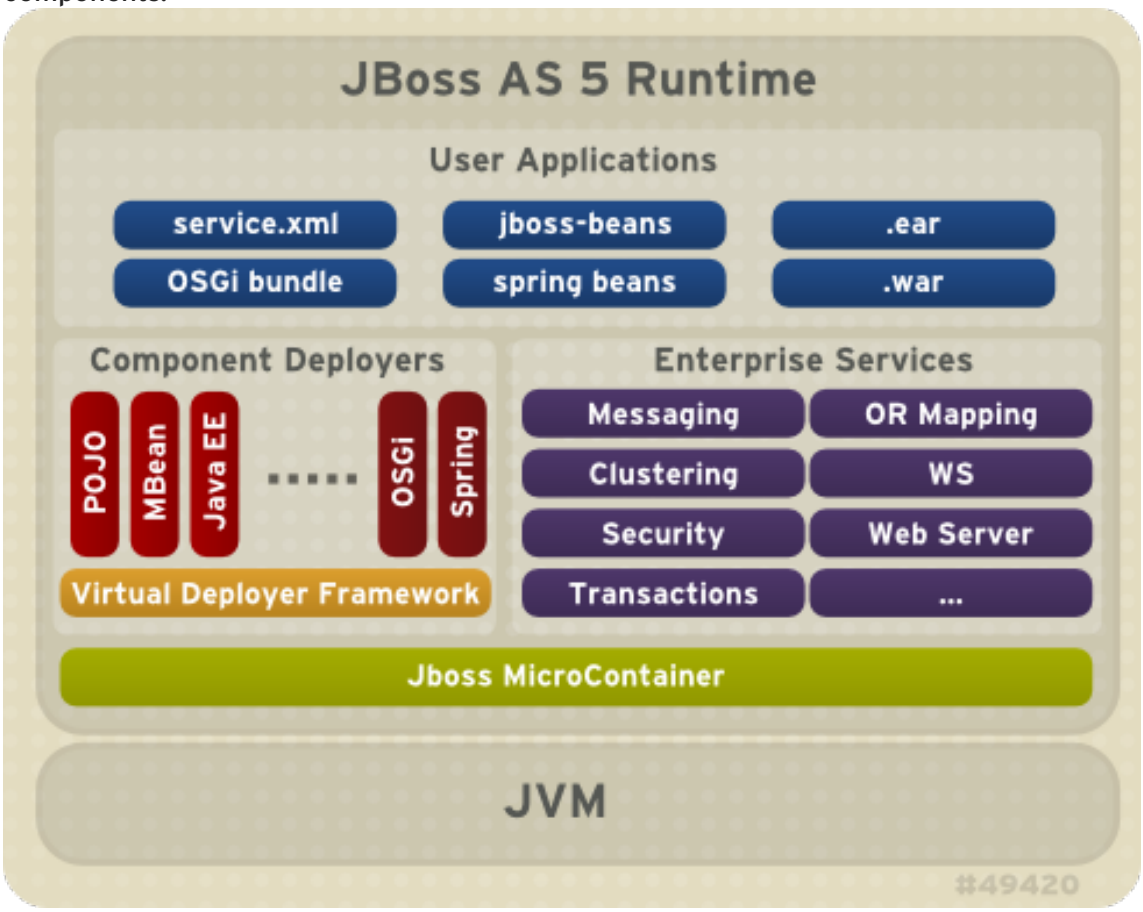
1.1. JBOSS ENTERPRISE APPLICATION PLATFORM USE CASES

- 99% of web applications involving a database
- Mission critical web applications likely to be clustered.
- Simple web applications with JSPs/Servlets upgrades to JBoss Enterprise 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

CHAPTER 2. JBOSS ENTERPRISE APPLICATION PLATFORM 5 ARCHITECTURE

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



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

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

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

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

2.1. THE JBOSS ENTERPRISE 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

CHAPTER 3. LOGGING

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

Section 3.1, "Logging Defaults" contains information about customizing the default logging behavior for the Platform. See Section 3.2, "Component-Specific Logging" for additional customization. Appendix B, *Logging Information and Recipes* provides some logging *recipes*, which you can customize to your needs.

3.1. LOGGING DEFAULTS

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

Table 3.1. Common log4j Configuration Directives

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

Example 3.1. Sample Appender

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

  <!-- Rollover at midnight each day -->
  <param name="DatePattern" value="'.'yyyy-MM-dd"/>
  <layout class="org.apache.log4j.PatternLayout">
  <!-- The default pattern: Date Priority [Category] (Thread) Message\n
-->
```

```
    <param name="ConversionPattern" value="%d %-5p [%c] (%t) %m%n"/>
    </layout>
  </appender>
```

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

3.2. COMPONENT-SPECIFIC LOGGING

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

3.2.1. SQL Logging with Hibernate

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

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

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

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

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

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

3.2.2. Transaction Service Logging

The TransactionManagerService included with the Enterprise Platform handles logging differently than the stand-alone Transaction Service. Specifically, it overrides the value of the com.arjuna.common.util.logger property given in the **jbossjta-properties.xml** file, forcing use of the **log4j_releveler** logger. All **INFO** level messages in the transaction code behave as **DEBUG** messages. Therefore, these messages are only present in log files if the filter level is **DEBUG**. All other log messages behave as normal.

CHAPTER 4. DEPLOYMENT

Deploying applications on JBoss Enterprise Application Platform is achieved by copying the application into the **`$JBOSS_HOME/server/default/deploy`** directory. You can replace *default* with different server profiles such as *all* or *minimal* (profiles are covered later in this guide). The JBoss Enterprise Application Platform constantly scans the deploy directory to pick up new applications or any changes to existing applications. This enables *hot deployment* of applications on the fly, while JBoss Enterprise Application Platform is still running.

4.1. 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:

WAR

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

EAR

The EAR application archive (e.g., myapp.ear) packages a Java EE enterprise application in a JAR file. It typically contains a WAR file for the web module, JAR files for EJB modules, as well as META-INF deployment descriptors such as application.xml and jboss-app.xml etc.

JBoss Microcontainer

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

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.

SAR

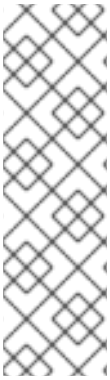
The SAR application archive (e.g., myservice.sar) packages a JBoss service in a JAR file. It is mostly used by JBoss Enterprise Application Platform internal services that have not been updated to support MC beans style deployments.

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.

DataSource

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



NOTE

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.

4.2. STANDARD SERVER PROFILES

The JBoss Enterprise Application Platform ships with six 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 starts the server in the *all* profile.

Each profile is contained in a directory named ***install_directory*/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.

all

Default profile loaded when **run.sh** is executed without the ***-c*** parameter. The profile provides clustering support and other enterprise extensions.

production

The *production* profile is based on the **all** profile and provides configuration optimized for production environments.

minimal

Starts the core server container without any of the enterprise services. Use the **minimal** profile as a base to build a customized version of JBoss Enterprise Application Platform that only contains the services you need.

default

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).

NOTE

The **`default`** profile is a misnomer; it is not loaded automatically if you do not specify a profile at start up. The **`all`** profile is loaded when you do not specify a profile at startup.

standard

The *standard* profile is the profile that has been tested for Java EE 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).

web

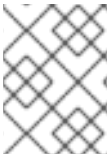
The *web* profile is an experimental, lightweight configuration created around JBoss Web that will follow the developments of the Java EE 6 web profile. Except for the **`servlet/jsp`** container, it provides support for JTA/JCA and JPA. It also limits itself to allowing access to the server only through the http port. Please note that this configuration is not Java EE 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.

CHAPTER 5. 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, you can 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.



NOTE

For detailed information regarding the Microcontainer architecture, refer to the Microcontainer User Guide hosted on docs.redhat.com.

CHAPTER 6. THE JNDI NAMING SERVICE

The naming service plays a key role in enterprise Java applications, providing the core infrastructure that is used to locate objects or services in an application server. It is also the mechanism that clients external to the application server use to locate services inside the application server. Application code, whether it is internal or external to the JBoss Enterprise Application Platform instance, needs only know that it needs to talk to the a message queue named **`queue/IncomingOrders`** and need not worry about any of the queue's configuration details.

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

6.1. AN OVERVIEW OF JNDI

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

The SPI layer is an abstraction that naming service providers must implement to enable the core JNDI classes to expose the naming service using the common JNDI client interface. An implementation of JNDI for a naming service is referred to as a *JNDI provider*. JBoss naming is an example JNDI implementation, based on the SPI classes. Note that the JNDI SPI is not needed by J2EE component developers.

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

6.1.1. Names

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

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

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

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

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

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

6.1.2. Contexts

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

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

6.1.2.1. Obtaining a Context using InitialContext

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

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

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

For each property found in both of these two sources, the property's value is determined as follows. If the property is one of the standard JNDI properties that specify a list of JNDI factories, all of the values

are concatenated into a single colon-separated list. For other properties, only the first value found is used. The preferred method of specifying the JNDI environment properties is through a **`jndi.properties`** file, which allows your code to externalize the JNDI provider specific information so that changing JNDI providers will not require changes to your code or recompilation.

The **`Context`** implementation used internally by the **`InitialContext`** class is determined at runtime. The default policy uses the environment property **`java.naming.factory.initial`**, which contains the class name of the **`javax.naming.spi.InitialContextFactory`** implementation. You obtain the name of the **`InitialContextFactory`** class from the naming service provider you are using.

Example 6.1, "A sample jndi.properties file" gives a sample **`jndi.properties`** file a client application would use to connect to a JBossNS service running on the local host at port 1099. The client application would need to have the **`jndi.properties`** file available on the application classpath. These are the properties that the JBossNS JNDI implementation requires. Other JNDI providers will have different properties and values.

Example 6.1. A sample jndi.properties file

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

6.2. THE JBOSS NAMING SERVICE ARCHITECTURE

The JBoss Naming Service (JBossNS) architecture is a Java socket/RMI based implementation of the **`javax.naming.Context`** interface. It is a client/server implementation that can be accessed remotely. The implementation is optimized so that access from within the same VM in which the JBossNS server is running does not involve sockets. Same VM access occurs through an object reference available as a global singleton. Figure 6.1, "Key components in the JBoss Naming Service architecture." illustrates some of the key classes in the JBossNS implementation and their relationships.

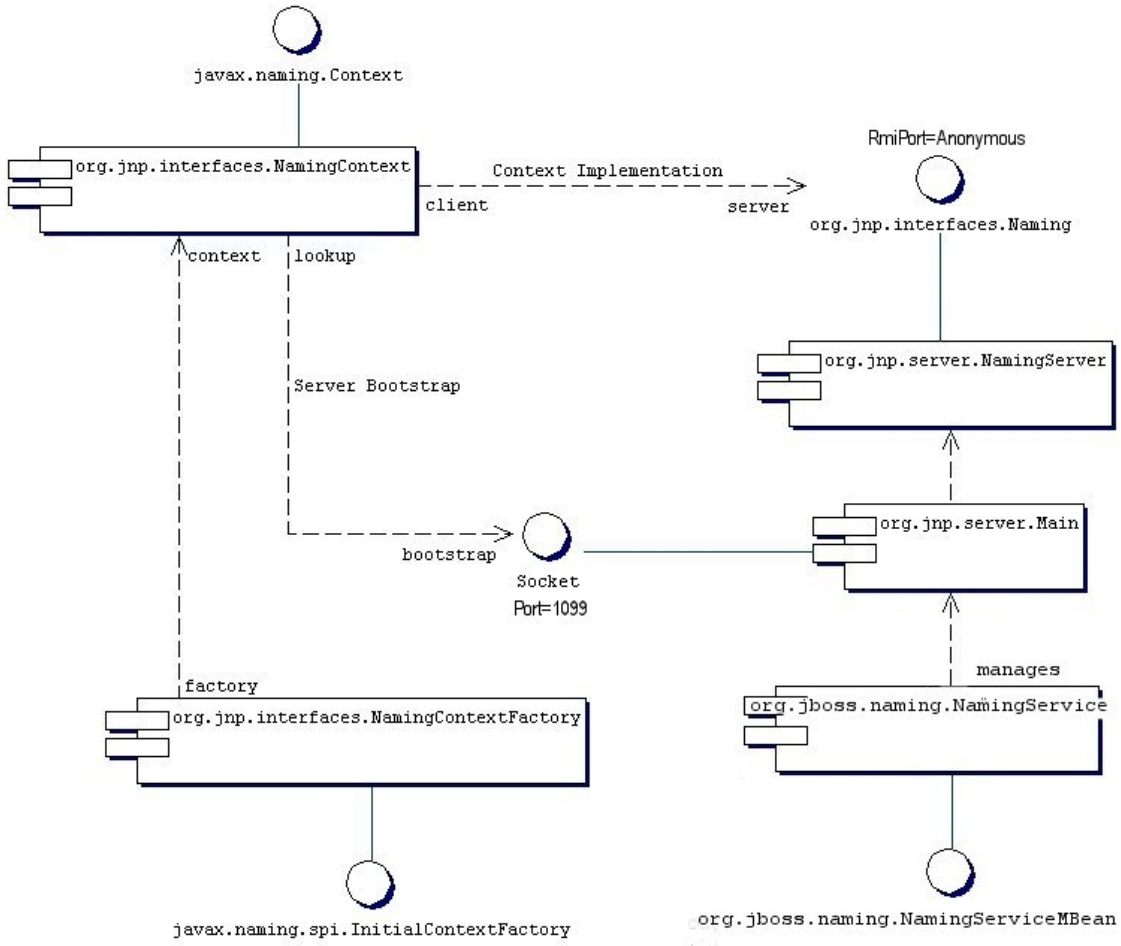


Figure 6.1. Key components in the JBoss Naming Service architecture.

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

- **Port**: The jnp protocol listening port for the **`NamingService`**. If not specified default is 1099, the same as the RMI registry default port.
- **RmiPort**: The RMI port on which the RMI Naming implementation will be exported. If not specified the default is 0 which means use any available port.
- **BindAddress**: The specific address the **`NamingService`** listens on. This can be used on a multi-homed host for a **`java.net.ServerSocket`** that will only accept connect requests on one of its addresses.
- **RmiBindAddress**: The specific address the RMI server portion of the **`NamingService`** listens on. This can be used on a multi-homed host for a **`java.net.ServerSocket`** that will only accept connect requests on one of its addresses. If this is not specified and the **`BindAddress`** is, the **`RmiBindAddress`** defaults to the **`BindAddress`** value.
- **Backlog**: The maximum queue length for incoming connection indications (a request to connect) is set to the **`backlog`** parameter. If a connection indication arrives when the queue is full, the connection is refused.
- **ClientSocketFactory**: An optional custom **`java.rmi.server.RMIClientSocketFactory`** implementation class name. If not specified the default **`RMIClientSocketFactory`** is used.

- **ServerSocketFactory**: An optional custom `java.rmi.server.RMIServerSocketFactory` implementation class name. If not specified the default `RMIServerSocketFactory` is used.

- **JNPServerSocketFactory**: An optional custom `javax.net.ServerSocketFactory` implementation class name. This is the factory for the `ServerSocket` used to bootstrap the download of the JBoss Naming Service `Naming` interface. If not specified the `javax.net.ServerSocketFactory.getDefault()` method value is used.

The `NamingService` also creates the `java:comp` context such that access to this context is isolated based on the context class loader of the thread that accesses the `java:comp` context. This provides the application component private ENC that is required by the J2EE specs. This segregation is accomplished by binding a `javax.naming.Reference` to a context that uses the `org.jboss.naming.ENCFactory` as its `javax.naming.ObjectFactory`. When a client performs a lookup of `java:comp`, or any subcontext, the `ENCFactory` checks the thread context `ClassLoader`, and performs a lookup into a map using the `ClassLoader` as the key.

If a context instance does not exist for the class loader instance, one is created and associated with that class loader in the `ENCFactory` map. Thus, correct isolation of an application component's ENC relies on each component receiving a unique `ClassLoader` that is associated with the component threads of execution.

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

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

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

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

- Exports the `NamingServer` instance's `org.jnp.naming.interfaces.Naming` RMI interface using the configured `RmiPort`, `ClientSocketFactory`, `ServerSocketFactory` attributes.

- Creates a socket that listens on the interface given by the `BindAddress` and `Port` attributes.

- Spawns a thread to accept connections on the socket.

6.3. THE NAMING INITIALCONTEXT FACTORIES

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

6.3.1. The standard naming context factory

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

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory to use. The value of the property should be the fully qualified class name of the factory class that will create an initial context. If it is not specified, a `javax.naming.NoInitialContextException` will be thrown when an `InitialContext` object is created.

- **java.naming.provider.url**: The name of the environment property for specifying the location of the JBoss JNDI service provider the client will use. The `NamingContextFactory` class uses this information to know which JBossNS server to connect to. The value of the property should be a URL string. For JBossNS the URL format is `jnp://host:port/[jndi_path]`. The `jnp:` portion of the URL is the protocol and refers to the socket/RMI based protocol used by JBoss. The `jndi_path` portion of the URL is an optional JNDI name relative to the root context, for example, `apps` or `apps/tmp`. Everything but the host component is optional. The following examples are equivalent because the default port value is 1099.

  - `jnp://www.jboss.org:1099/`
  - `www.jboss.org:1099`
  - `www.jboss.org`

- **java.naming.factory.url.pkgs**: The name of the environment property for specifying the list of package prefixes to use when loading in URL context factories. The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory. For the JBoss JNDI provider this must be `org.jboss.naming:org.jnp.interfaces`. This property is essential for locating the `jnp:` and `java:` URL context factories of the JBoss JNDI provider.

- **jnp.socketFactory**: The fully qualified class name of the `javax.net.SocketFactory` implementation to use to create the bootstrap socket. The default value is `org.jnp.interfaces.TimedSocketFactory`. The `TimedSocketFactory` is a simple `SocketFactory` implementation that supports the specification of a connection and read timeout. These two properties are specified by:

- **jnp.timeout**: The connection timeout in milliseconds. The default value is 0 which means the connection will block until the VM TCP/IP layer times out.

- **jnp.sotimeout**: The connected socket read timeout in milliseconds. The default value is 0 which means reads will block. This is the value passed to the `Socket.setSoTimeout` on the newly connected socket.

When a client creates an `InitialContext` with these JBossNS properties available, the `org.jnp.interfaces.NamingContextFactory` object is used to create the `Context` instance that will be used in subsequent operations. The `NamingContextFactory` is the JBossNS implementation of the `javax.naming.spi.InitialContextFactory` interface. When the `NamingContextFactory` class is asked to create a `Context`, it creates an `org.jnp.interfaces.NamingContext` instance with the `InitialContext` environment and name of the context in the global JNDI namespace. It is the `NamingContext` instance that actually

performs the task of connecting to the JBossNS server, and implements the **Context** interface. The **Context.PROVIDER_URL** information from the environment indicates from which server to obtain a **NamingServer** RMI reference.

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

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

6.3.2. The org.jboss.naming.NamingContextFactory

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

6.3.3. Naming Discovery in Clustered Environments

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

- **jnp.partitionName**: The cluster partition name discovery should be restricted to. If you are running in an environment with multiple clusters, you may want to restrict the naming discovery to a particular cluster. There is no default value, meaning that any cluster response will be accepted.

- **jnp.discoveryGroup**: The multicast IP/address to which the discovery query is sent. The default is 230.0.0.4.

- **jnp.discoveryPort**: The port to which the discovery query is sent. The default is 1102.

- **jnp.discoveryTimeout**: The time in milliseconds to wait for a discovery query response. The default value is 5000 (5 seconds).

- **jnp.disableDiscovery**: A flag indicating if the discovery process should be avoided. Discovery occurs when either no **`Context.PROVIDER_URL`** is specified, or no valid naming service could be located among the URLs specified. If the **`jnp.disableDiscovery`** flag is true, then discovery will not be attempted.

6.3.4. The HTTP InitialContext Factory Implementation

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

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

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory, which must be **`org.jboss.naming.HttpNamingContextFactory`**.

- **java.naming.provider.url** (or **`Context.PROVIDER_URL`**): This must be set to the HTTP URL of the JNDI factory. The full HTTP URL would be the public URL of the JBoss servlet container plus **`/invoker/JNDIFactory`**. Examples include:

  - **`http://www.jboss.org:8080/invoker/JNDIFactory`**
  - **`http://www.jboss.org/invoker/JNDIFactory`**
  - **`https://www.jboss.org/invoker/JNDIFactory`**

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

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

The JNDI **`Context`** implementation returned by the **`HttpNamingContextFactory`** is a proxy that delegates invocations made on it to a bridge servlet which forwards the invocation to the **`NamingService`** through the JMX bus and marshalls the reply back over HTTP. The proxy needs to know what the URL of the bridge servlet is in order to operate. This value may have been bound on the

server side if the JBoss web server has a well known public interface. If the JBoss web server is sitting behind one or more firewalls or proxies, the proxy cannot know what URL is required. In this case, the proxy will be associated with a system property value that must be set in the client VM. For more information on the operation of JNDI over HTTP see Section 6.4.1, “Accessing JNDI over HTTP” .

6.3.5. The Login InitialContext Factory Implementation

JAAS is the preferred method for authenticating a remote client to JBoss. However, for simplicity and to ease the migration from other application server environment that do not use JAAS, JBoss allows you the security credentials to be passed through the **`InitialContext`**. JAAS is still used under the covers, but there is no manifest use of the JAAS interfaces in the client application.

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

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory, which must be **`org.jboss.security.jndi.LoginInitialContextFactory`**.
- **java.naming.provider.url**: This must be set to a **`NamingContextFactory`** provider URL. The **`LoginIntialContext`** is really just a wrapper around the **`NamingContextFactory`** that adds a JAAS login to the existing **`NamingContextFactory`** behavior.
- **java.naming.factory.url.pkgs**: For all JBoss JNDI provider this must be **`org.jboss.naming:org.jnp.interfaces`**. This property is essential for locating the **`jnp:`** and **`java:`** URL context factories of the JBoss JNDI provider.
- **java.naming.security.principal** (or **`Context.SECURITY_PRINCIPAL`**): The principal to authenticate. This may be either a **`java.security.Principal`** implementation or a string representing the name of a principal.
- **java.naming.security.credentials** (or **`Context.SECURITY_CREDENTIALS`**), The credentials that should be used to authenticate the principal, e.g., password, session key, etc.
- **java.naming.security.protocol**: (**`Context.SECURITY_PROTOCOL`**) This gives the name of the JAAS login module to use for the authentication of the principal and credentials.

6.3.6. The ORBInitialContextFactory

When using Sun's CosNaming it is necessary to use a different naming context factory from the default. CosNaming looks for the ORB in JNDI instead of using the the ORB configured in **`deploy/iiop-service.xml?`**. It is necessary to set the global context factory to **`org.jboss.iiop.naming.ORBInitialContextFactory`**, which sets the ORB to JBoss's ORB. This is done in the **`conf/jndi.properties`** file:

```
# DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING
#
java.naming.factory.initial=org.jboss.iiop.naming.ORBInitialContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

It is also necessary to use **`ORBInitialContextFactory`** when using CosNaming in an application client.

6.4. JNDI OVER HTTP

In addition to the legacy RMI/JRMP with a socket bootstrap protocol, JBoss provides support for accessing its JNDI naming service over HTTP.

6.4.1. Accessing JNDI over HTTP

This capability is provided by **`http-invoker.sar`**. The structure of the **`http-invoker.sar`** is:

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

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

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

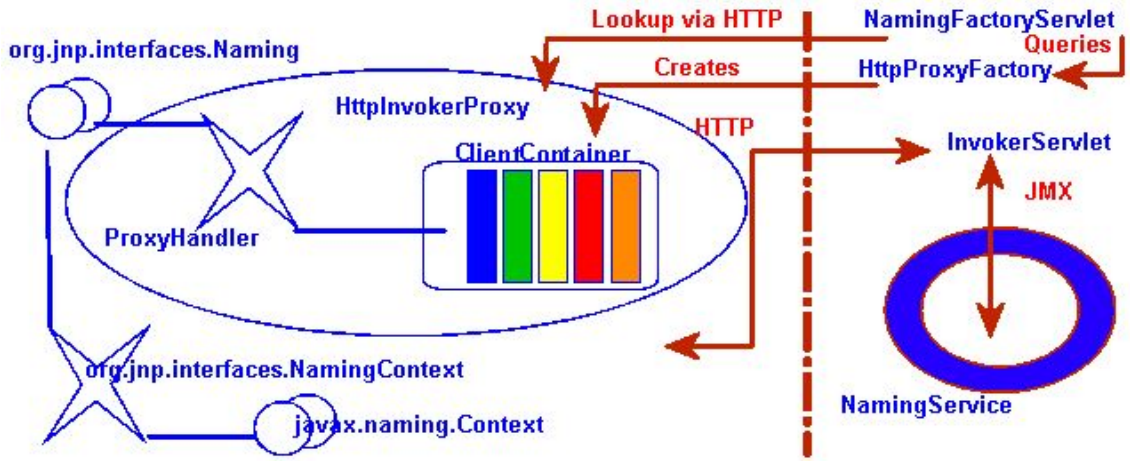


Figure 6.2. The HTTP invoker proxy/server structure for a JNDI Context

Before looking at the configurations let's look at the operation of the **`http-invoker`** services. Figure 6.2, “The HTTP invoker proxy/server structure for a JNDI Context” shows a logical view of the structure of a JBoss JNDI proxy and its relationship to the JBoss server side components of the **`http-invoker`**. The proxy is obtained from the **`NamingFactoryServlet`** using an **`InitialContext`** with

the **Context.INITIAL_CONTEXT_FACTORY** property set to **org.jboss.naming.HttpNamingContextFactory**, and the **Context.PROVIDER_URL** property set to the HTTP URL of the **NamingFactoryServlet**. The resulting proxy is embedded in an **org.jnp.interfaces.NamingContext** instance that provides the **Context** interface implementation.

The proxy is an instance of **org.jboss.invocation.http.interfaces.HttpInvokerProxy**, and implements the **org.jnp.interfaces.Naming** interface. Internally the **HttpInvokerProxy** contains an invoker that marshalls the **Naming** interface method invocations to the **InvokerServlet** via HTTP posts. The **InvokerServlet** translates these posts into JMX invocations to the **NamingService**, and returns the invocation response back to the proxy in the HTTP post response.

There are several configuration values that need to be set to tie all of these components together and Figure 6.3, "The relationship between configuration files and JNDI/HTTP component" illustrates the relationship between configuration files and the corresponding components.

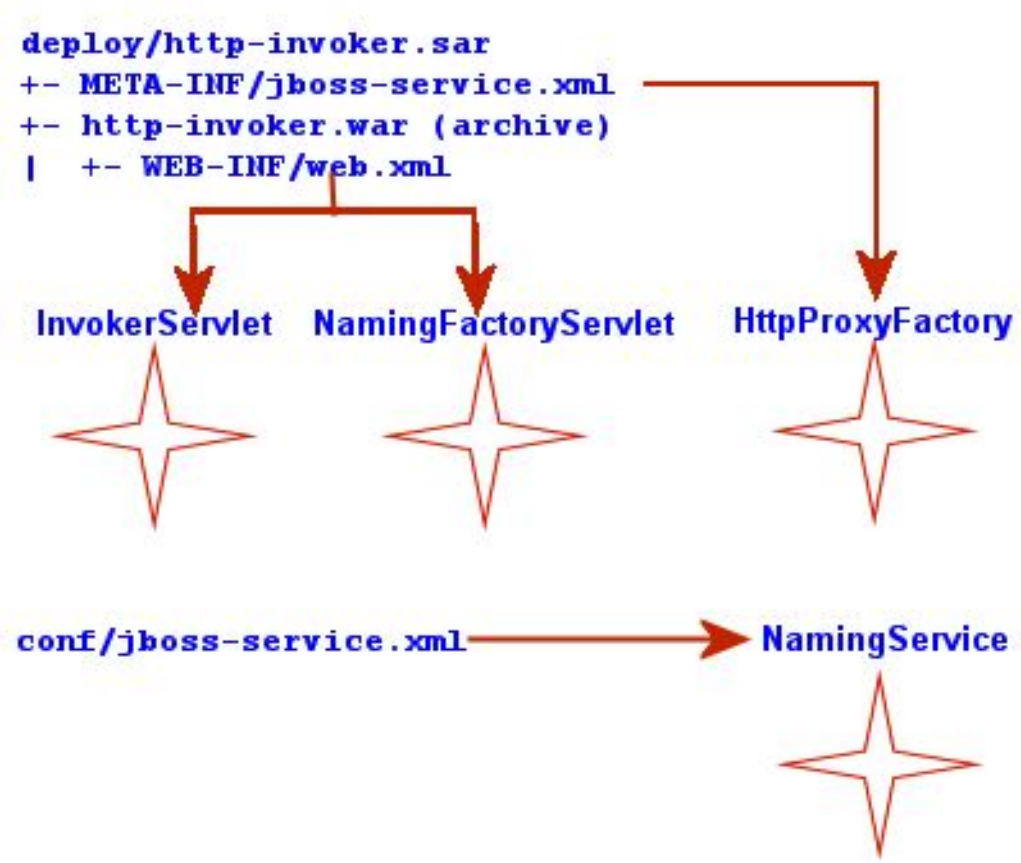


Figure 6.3. The relationship between configuration files and JNDI/HTTP component

The **http-invoker.sar/META-INF/jboss-service.xml** descriptor defines the **HttpProxyFactory** that creates the **HttpInvokerProxy** for the **NamingService**. The attributes that need to be configured for the **HttpProxyFactory** include:

- **InvokerName**: The JMX **ObjectName** of the **NamingService** defined in the **conf/jboss-service.xml** descriptor. The standard setting used in the JBoss distributions is **jboss:service=Naming**.
- **InvokerURL** or **InvokerURLPrefix** + InvokerURLSuffix + **UseHostName**. You can specify the full HTTP URL to the **InvokerServlet** using the **InvokerURL** attribute, or you can specify the hostname independent parts of the URL and have the **HttpProxyFactory** fill them in. An example **InvokerURL** value would be

`http://jbosshost1.dot.com:8080/invoker/JMXInvokerServlet`. This can be broken down into:

- **InvokerURLPrefix**: the URL prefix prior to the hostname. Typically this will be `http://` or `https://` if SSL is to be used.
- **InvokerURLSuffix**: the URL suffix after the hostname. This will include the port number of the web server as well as the deployed path to the `InvokerServlet`. For the example `InvokerURL` value the `InvokerURLSuffix` would be `:8080/invoker/JMXInvokerServlet` without the quotes. The port number is determined by the web container service settings. The path to the `InvokerServlet` is specified in the `http-invoker.sar/invoker.war/WEB-INF/web.xml` descriptor.
- **UseHostName**: a flag indicating if the hostname should be used in place of the host IP address when building the hostname portion of the full `InvokerURL`. If true, `InetAddress.getLocalHost().getHostName` method will be used. Otherwise, the `InetAddress.getLocalHost().getHostAddress()` method is used.

- **ExportedInterface**: The `org.jnp.interfaces.Naming` interface the proxy will expose to clients. The actual client of this proxy is the JBoss JNDI implementation `NamingContext` class, which JNDI client obtain from `InitialContext` lookups when using the JBoss JNDI provider.
- **JndiName**: The name in JNDI under which the proxy is bound. This needs to be set to a blank/empty string to indicate the interface should not be bound into JNDI. We can't use the JNDI to bootstrap itself. This is the role of the `NamingFactoryServlet`.

The `http-invoker.sar/invoker.war/WEB-INF/web.xml` descriptor defines the mappings of the `NamingFactoryServlet` and `InvokerServlet` along with their initialization parameters. The configuration of the `NamingFactoryServlet` relevant to JNDI/HTTP is the `JNDIFactory` entry which defines:

- A `namingProxyMBean` initialization parameter that maps to the `HttpProxyFactory` MBean name. This is used by the `NamingFactoryServlet` to obtain the `Naming` proxy which it will return in response to HTTP posts. For the default `http-invoker.sar/META-INF/jboss-service.xml` settings the name `jboss:service=invoker,type=http,target=Naming`.
- A proxy initialization parameter that defines the name of the `namingProxyMBean` attribute to query for the Naming proxy value. This defaults to an attribute name of `Proxy`.
- The servlet mapping for the `JNDIFactory` configuration. The default setting for the unsecured mapping is `/JNDIFactory/*`. This is relative to the context root of the `http-invoker.sar/invoker.war`, which by default is the WAR name minus the `.war` suffix.

The configuration of the `InvokerServlet` relevant to JNDI/HTTP is the `JMXInvokerServlet` which defines:

- The servlet mapping of the `InvokerServlet`. The default setting for the unsecured mapping is `/JMXInvokerServlet/*`. This is relative to the context root of the `http-invoker.sar/invoker.war`, which by default is the WAR name minus the `.war` suffix.

6.4.2. Accessing JNDI over HTTPS

To be able to access JNDI over HTTP/SSL you need to enable an SSL connector on the web container. The details of this are covered in the Integrating Servlet Containers for Tomcat. We will demonstrate

the use of HTTPS with a simple example client that uses an HTTPS URL as the JNDI provider URL. We will provide an SSL connector configuration for the example, so unless you are interested in the details of the SSL connector setup, the example is self contained.

We also provide a configuration of the **HttpProxyFactory** setup to use an HTTPS URL. The following example shows the section of the **http-invoker.sarjboss-service.xml** descriptor that the example installs to provide this configuration. All that has changed relative to the standard HTTP configuration are the **InvokerURLPrefix** and **InvokerURLSuffix** attributes, which setup an HTTPS URL using the 8443 port.

```
<!-- Expose the Naming service interface via HTTPS -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
       name="jboss:service=invoker,type=https,target=Naming">
    <!-- The Naming service we are proxying -->
    <attribute name="InvokerName">jboss:service=Naming</attribute>
    <!-- Compose the invoker URL from the cluster node address -->
    <attribute name="InvokerURLPrefix">https://</attribute>
    <attribute name="InvokerURLSuffix">:8443/invoker/JMXInvokerServlet
</attribute>
    <attribute name="UseHostName">true</attribute>
    <attribute name="ExportedInterface">org.jnp.interfaces.Naming
</attribute>
    <attribute name="JndiName"/>
    <attribute name="ClientInterceptors">
        <interceptors>
            <interceptor>org.jboss.proxy.ClientMethodInterceptor
</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor
</interceptor>

<interceptor>org.jboss.naming.interceptors.ExceptionInterceptor
</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor
</interceptor>
        </interceptors>
    </attribute>
</mbean>
```

At a minimum, a JNDI client using HTTPS requires setting up a HTTPS URL protocol handler. We will be using the Java Secure Socket Extension (JSSE) for HTTPS. The JSSE documentation does a good job of describing what is necessary to use HTTPS, and the following steps were needed to configure the example client shown in Example 6.2, “A JNDI client that uses HTTPS as the transport” :

- A protocol handler for HTTPS URLs must be made available to Java. The JSSE release includes an HTTPS handler in the **com.sun.net.ssl.internal.www.protocol** package. To enable the use of HTTPS URLs you include this package in the standard URL protocol handler search property, **java.protocol.handler.pkgs**. We set the **java.protocol.handler.pkgs** property in the Ant script.

- The JSSE security provider must be installed in order for SSL to work. This can be done either by installing the JSSE jars as an extension package, or programatically. We use the programatic approach in the example since this is less intrusive. Line 18 of the **ExClient** code demonstrates how this is done.

- The JNDI provider URL must use HTTPS as the protocol. Lines 24-25 of the **ExClient** code specify an HTTP/SSL connection to the localhost on port 8443. The hostname and port are

defined by the web container SSL connector.

- The validation of the HTTPS URL hostname against the server certificate must be disabled. By default, the JSSE HTTPS protocol handler employs a strict validation of the hostname portion of the HTTPS URL against the common name of the server certificate. This is the same check done by web browsers when you connect to secured web site. We are using a self-signed server certificate that uses a common name of "**`Chapter 8 SSL Example`**" rather than a particular hostname, and this is likely to be common in development environments or intranets. The JBoss **`HttpInvokerProxy`** will override the default hostname checking if a **`org.jboss.security.ignoreHttpsHost`** system property exists and has a value of true. We set the **`org.jboss.security.ignoreHttpsHost`** property to true in the Ant script.

Example 6.2. A JNDI client that uses HTTPS as the transport

```
package org.jboss.chap3.ex1;

import java.security.Security;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Properties env = new Properties();
        env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                        "org.jboss.naming.HttpNamingContextFactory");
        env.setProperty(Context.PROVIDER_URL,

"https://localhost:8443/invoker/JNDIFactorySSL");

        Context ctx = new InitialContext(env);
        System.out.println("Created InitialContext, env=" + env);

        Object data = ctx.lookup("jmx/invoker/RMIAdaptor");
        System.out.println("lookup(jmx/invoker/RMIAdaptor): " + data);
    }
}
```

To test the client, first build the chapter 3 example to create the **chap3** configuration fileset.

```
[examples]$ ant -Dchap=naming config
```

Next, start the JBoss server using the **naming** configuration fileset:

```
[bin]$ sh run.sh -c naming
```

And finally, run the **`ExClient`** using:

```
[examples]$ ant -Dchap=naming -Dex=1 run-example
...
run-example1:
```

```
[java] Created InitialContext, env={java.naming. \
provider.url=https://localhost:8443/invoker/JNDIFactorySSL, java.naming. \
factory.initial=org.jboss.naming.HttpNamingContextFactory}
     [java] lookup(jmx/invoker/RMIAdaptor): org.jboss.invocation.jrmp. \
     interfaces.JRMPInvokerP
roxy@cac3fa
```

6.4.3. Securing Access to JNDI over HTTP

One benefit to accessing JNDI over HTTP is that it is easy to secure access to the JNDI **`InitialContext`** factory as well as the naming operations using standard web declarative security. This is possible because the server side handling of the JNDI/HTTP transport is implemented with two servlets. These servlets are included in the **`http-invoker.sar/invoker.war`** directory found in the **`default`** and **`all`** configuration deploy directories as shown previously. To enable secured access to JNDI you need to edit the **`invoker.war/WEB-INF/web.xml`** descriptor and remove all unsecured servlet mappings. For example, the **`web.xml`** descriptor shown in Example 6.3, "An example web.xml descriptor for secured access to the JNDI servlets" only allows access to the **`invoker.war`** servlets if the user has been authenticated and has a role of **`HttpInvoker`**.

Example 6.3. An example web.xml descriptor for secured access to the JNDI servlets

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- ### Servlets -->
    <servlet>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <servlet-class>
            org.jboss.invocation.http.servlet.InvokerServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>   <servlet>
        <servlet-name>JNDIFactory</servlet-name>
        <servlet-class>
            org.jboss.invocation.http.servlet.NamingFactoryServlet
        </servlet-class>
        <init-param>
            <param-name>namingProxyMBean</param-name>
            <param-
value>jboss:service=invoker,type=http,target=Naming</param-value>
        </init-param>
        <init-param>
            <param-name>proxyAttribute</param-name>
            <param-value>Proxy</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <!-- ### Servlet Mappings -->
    <servlet-mapping>
        <servlet-name>JNDIFactory</servlet-name>
        <url-pattern>/restricted/JNDIFactory/*</url-pattern>
    </servlet-mapping>
```

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

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

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

The **security-domain** element defines the name of the security domain that will be used for the JAAS login module configuration used for authentication and authorization.

6.4.4. Securing Access to JNDI with a Read-Only Unsecured Context

Another feature available for the JNDI/HTTP naming service is the ability to define a context that can be accessed by unauthenticated users in read-only mode. This can be important for services used by the authentication layer. For example, the **SRPLoginModule** needs to lookup the SRP server interface used to perform authentication. The rest of this section explains how read-only works in JBoss Enterprise Application Platform.

First, the **ReadOnlyJNDIFactory** is declared in **invoker.sar/WEB-INF/web.xml**. It will be mapped to **/invoker/ReadOnlyJNDIFactory**.

```
<servlet>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <description>A servlet that exposes the JBoss JNDI Naming service stub
          through http, but only for a single read-only context. The
```

```
return content
          is serialized MarshalledValue containing the
org.jnp.interfaces.Naming
          stub.
    </description>
    <servlet-
class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-
class>
    <init-param>
        <param-name>namingProxyMBean</param-name>
        <param-
value>jboss:service=invoker,type=http,target=Naming,readonly=true</param-
value>
    </init-param>
    <init-param>
        <param-name>proxyAttribute</param-name>
        <param-value>Proxy</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!-- ... -->

<servlet-mapping>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <url-pattern>/ReadOnlyJNDIFactory/*</url-pattern>
</servlet-mapping>
```

The factory only provides a JNDI stub which needs to be connected to an invoker. Here the invoker is **jboss:service=invoker,type=http,target=Naming,readonly=true**. This invoker is declared in the **http-invoker.sar/META-INF/jboss-service.xml** file.

```
   <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
      name="jboss:service=invoker,type=http,target=Naming,readonly=true">
      <attribute name="InvokerName">jboss:service=Naming</attribute>
      <attribute name="InvokerURLPrefix">http://</attribute>
      <attribute
name="InvokerURLSuffix">:8080/invoker/readonly/JMXInvokerServlet</attribut
e>
      <attribute name="UseHostName">true</attribute>
      <attribute
name="ExportedInterface">org.jnp.interfaces.Naming</attribute>
      <attribute name="JndiName"></attribute>
      <attribute name="ClientInterceptors">
          <interceptors>

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

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.naming.interceptors.ExceptionInterceptor</intercept
or>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
```

```
          </interceptors>
      </attribute>
   </mbean>
```

The proxy on the client side needs to talk back to a specific invoker servlet on the server side. The configuration here has the actual invocations going to **`/invoker/readonly/JMXInvokerServlet`**. This is actually the standard **`JMXInvokerServlet`** with a read-only filter attached.

```
    <filter>
        <filter-name>ReadOnlyAccessFilter</filter-name>
        <filter-
class>org.jboss.invocation.http.servlet.ReadOnlyAccessFilter</filter-
class>
        <init-param>
            <param-name>readOnlyContext</param-name>
            <param-value>readonly</param-value>
            <description>The top level JNDI context the filter will
enforce
                read-only access on. If specified only Context.lookup
operations
                will be allowed on this context. Another other operations
or
                lookups on any other context will fail. Do not associate
this
                filter with the JMXInvokerServlets if you want
unrestricted
                access. </description>
        </init-param>
        <init-param>
            <param-name>invokerName</param-name>
            <param-value>jboss:service=Naming</param-value>
            <description>The JMX ObjectName of the naming service mbean
</description>
        </init-param>
    </filter>

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

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

The **`readOnlyContext`** parameter is set to **`readonly`** which means that when you access JBoss through the **`ReadOnlyJNDIFactory`**, you will only be able to access data in the **`readonly`** context. Here is a code fragment that illustrates the usage:

```
Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
                "http://localhost:8080/invoker/ReadOnlyJNDIFactory");

Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly/data");
```

Attempts to look up any objects outside of the readonly context will fail. Note that JBoss doesn't ship with any data in the **readonly** context, so the readonly context won't be bound usable unless you create it.

6.5. ADDITIONAL NAMING MBEANS

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

6.5.1. JNDI Binding Manager

The JNDI binding manager service allows you to quickly bind objects into JNDI for use by application code. The MBean class for the binding service is **org.jboss.naming.JNDIBindingServiceMgr**. It has a single attribute, **BindingsConfig**, which accepts an XML document that conforms to the **jndi-binding-service_1_0.xsd** schema. The content of the **BindingsConfig** attribute is unmarshalled using the JBossXB framework. The following is an MBean definition that shows the most basic form usage of the JNDI binding manager service.

```
<mbean code="org.jboss.naming.JNDIBindingServiceMgr"
       name="jboss.tests:name=example1">
    <attribute name="BindingsConfig" serialDataType="jbxb">
        <jndi:bindings xmlns:xs="http://www.w3.org/2001/XMLSchema-
instance"
                       xmlns:jndi="urn:jboss:jndi-binding-service:1.0"
                       xs:schemaLocation="urn:jboss:jndi-binding-service
\
         resource:jndi-binding-service_1_0.xsd">
            <jndi:binding name="bindexample/message">
                <jndi:value trim="true">
                    Hello, JNDI!
                </jndi:value>
            </jndi:binding>
        </jndi:bindings>
    </attribute>
</mbean>
```

This binds the text string "**Hello, JNDI!**" under the JNDI name **bindexample/message**. An application would look up the value just as it would for any other JNDI value. The **trim** attribute specifies that leading and trailing whitespace should be ignored. The use of the attribute here is purely for illustrative purposes as the default value is true.

```
InitialContext ctx  = new InitialContext();
String         text = (String) ctx.lookup("bindexample/message");
```

String values themselves are not that interesting. If a JavaBeans property editor is available, the desired class name can be specified using the **type** attribute

```
<jndi:binding name="urls/jboss-home">
    <jndi:value type="java.net.URL">http://www.jboss.org</jndi:value>
</jndi:binding>
```

The **editor** attribute can be used to specify a particular property editor to use.

```
<jndi:binding name="hosts/localhost">
    <jndi:value editor="org.jboss.util.propertyeditor.InetAddressEditor">
        127.0.0.1
    </jndi:value>
</jndi:binding>
```

For more complicated structures, any JBossXB-ready schema may be used. The following example shows how a **java.util.Properties** object would be mapped.

```
<jndi:binding name="maps/testProps">
    <java:properties xmlns:java="urn:jboss:java-properties"
                     xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                     xs:schemaLocation="urn:jboss:java-properties \
       resource:java-properties_1_0.xsd">
        <java:property>
            <java:key>key1</java:key>
            <java:value>value1</java:value>
        </java:property>
        <java:property>
            <java:key>key2</java:key>
            <java:value>value2</java:value>
        </java:property>
    </java:properties>
</jndi:binding>
```

6.5.2. The org.jboss.naming.NamingAlias MBean

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

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

The following example provides a mapping of the JNDI name **QueueConnectionFactory** to the name **ConnectionFactory**.

```
<mbean code="org.jboss.naming.NamingAlias"
```

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

6.5.3. org.jboss.naming.ExternalContext MBean

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

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

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

- myldap.properties

The MBean definition below shows a binding to an external LDAP context into the JBoss JNDI namespace under the name **external/ldap/jboss**.

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"

name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/jboss">
    <attribute name="JndiName">external/ldap/jboss</attribute>
    <attribute name="Properties">
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url=ldap://ldaphost.jboss.org:389/o=jboss.org
        java.naming.security.principal=cn=Directory Manager
        java.naming.security.authentication=simple
        java.naming.security.credentials=secret
    </attribute>
    <attribute name="InitialContext"> javax.naming.ldap.InitialLdapContext
</attribute>
    <attribute name="RemoteAccess">true</attribute>
</mbean>
```

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

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

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

```
<!-- Bind the /usr/local file system directory  -->
<mbean code="org.jboss.naming.ExternalContext"

name="jboss.jndi:service=ExternalContext,jndiName=external/fs/usr/local">
    <attribute name="JndiName">external/fs/usr/local</attribute>
    <attribute name="Properties">

java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
        java.naming.provider.url=file:///usr/local
    </attribute>
    <attribute
name="InitialContext">javax.naming.IntialContext</attribute>
</mbean>
```

This configuration describes binding a local file system directory **/usr/local** into the JBoss JNDI namespace under the name **external/fs/usr/local**.

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

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

6.5.4. The org.jboss.naming.JNDIView MBean

The JNDIView MBean allows the user to view the JNDI namespace tree as it exists in the JBoss server using the JMX agent view interface. To view the JBoss JNDI namespace using the JNDIView MBean, you connect to the JMX Agent View using the http interface. The default settings put this at **`http://localhost:8080/jmx-console/`**. On this page you will see a section that lists the registered MBeans sorted by domain. It should look something like that shown in Figure 6.4, "The JMX Console view of the configured JBoss MBeans".

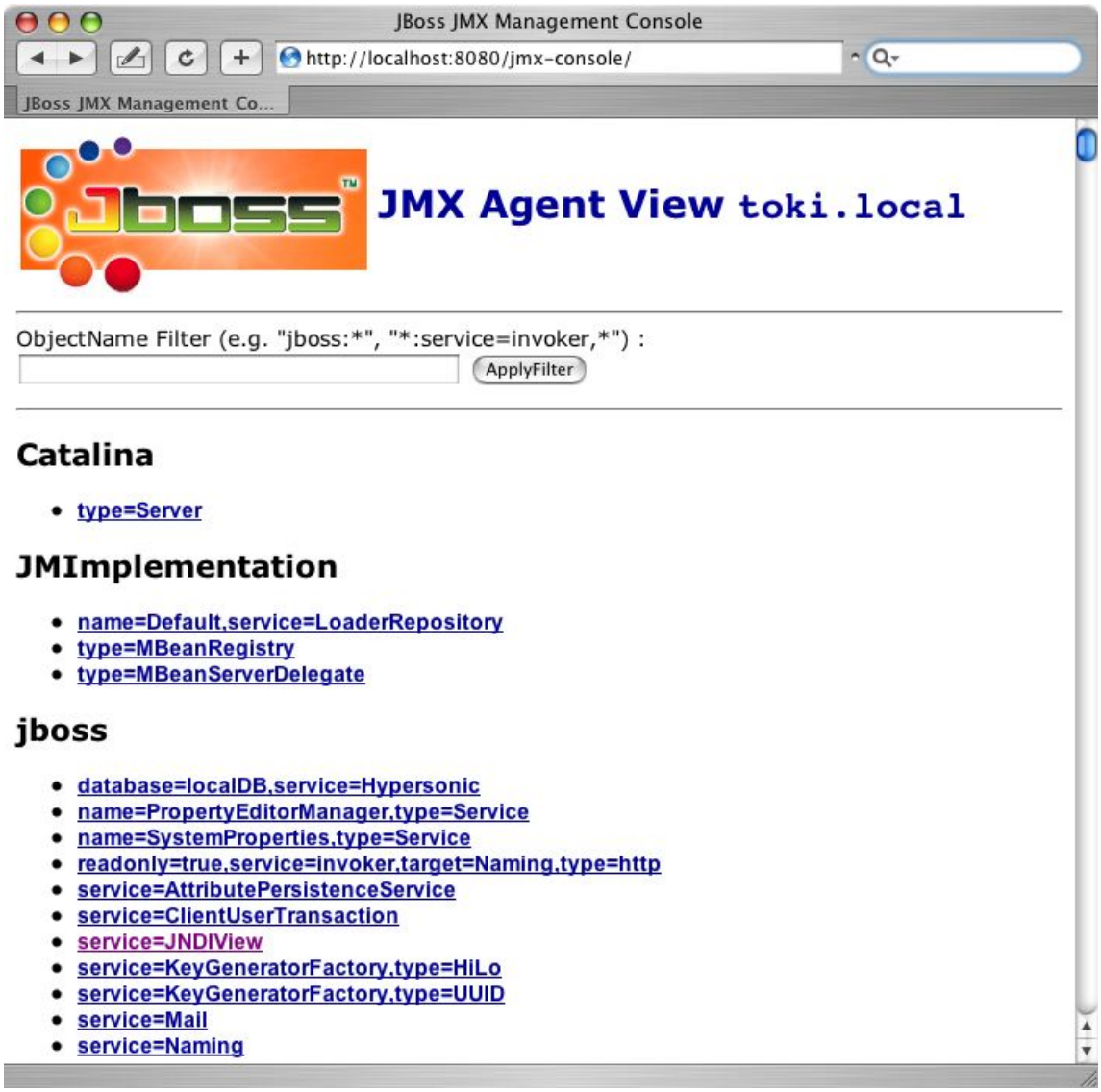


Figure 6.4. The JMX Console view of the configured JBoss MBeans

Selecting the JNDIView link takes you to the JNDIView MBean view, which will have a list of the JNDIView MBean operations. This view should look similar to that shown in Figure 6.5, "The JMX Console view of the JNDIView MBean".

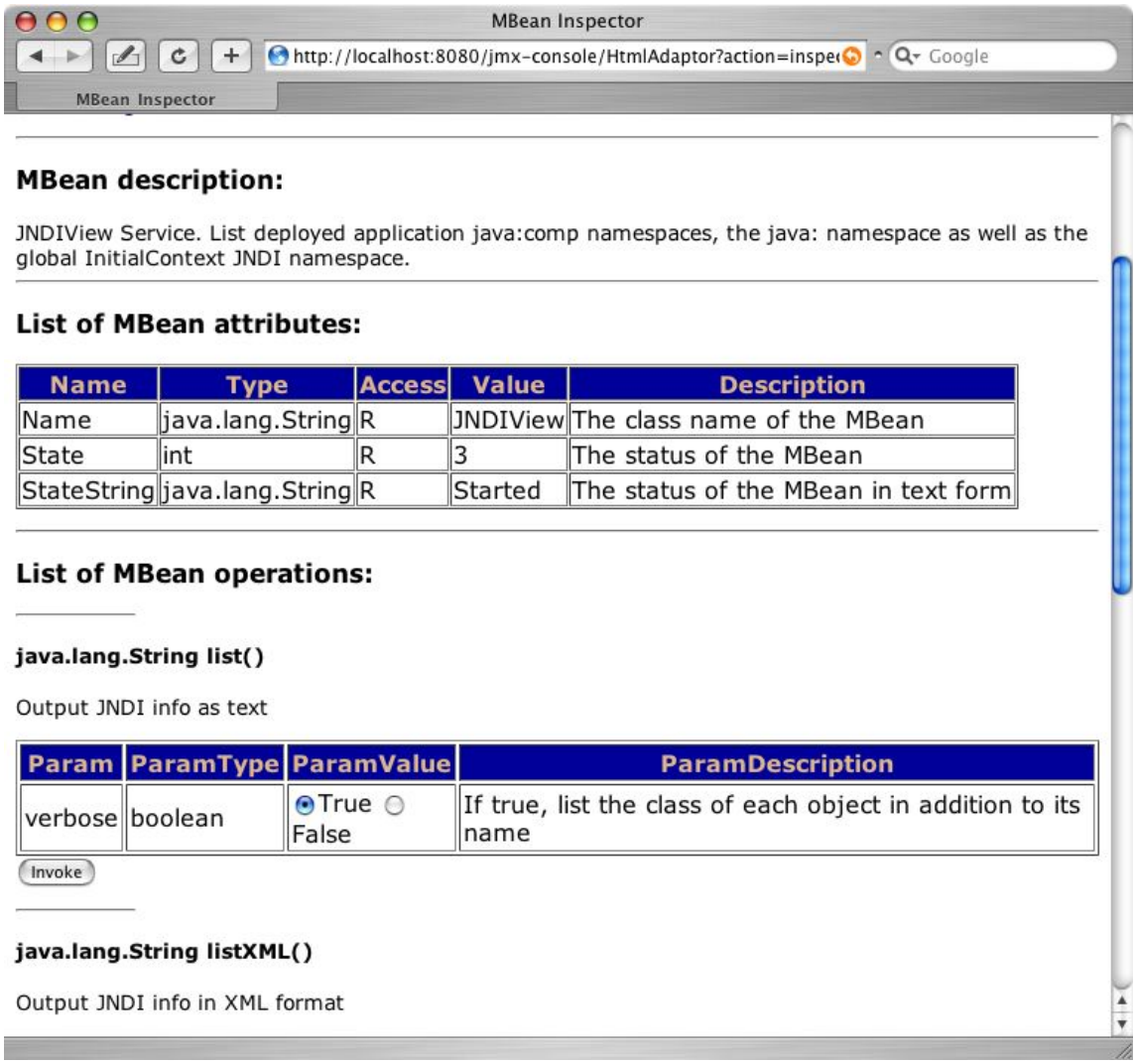


Figure 6.5. The JMX Console view of the JNDIView MBean

The list operation dumps out the JBoss server JNDI namespace as an HTML page using a simple text view. As an example, invoking the list operation produces the view shown in Figure 6.6, "The JMX Console view of the JNDIView list operation output".

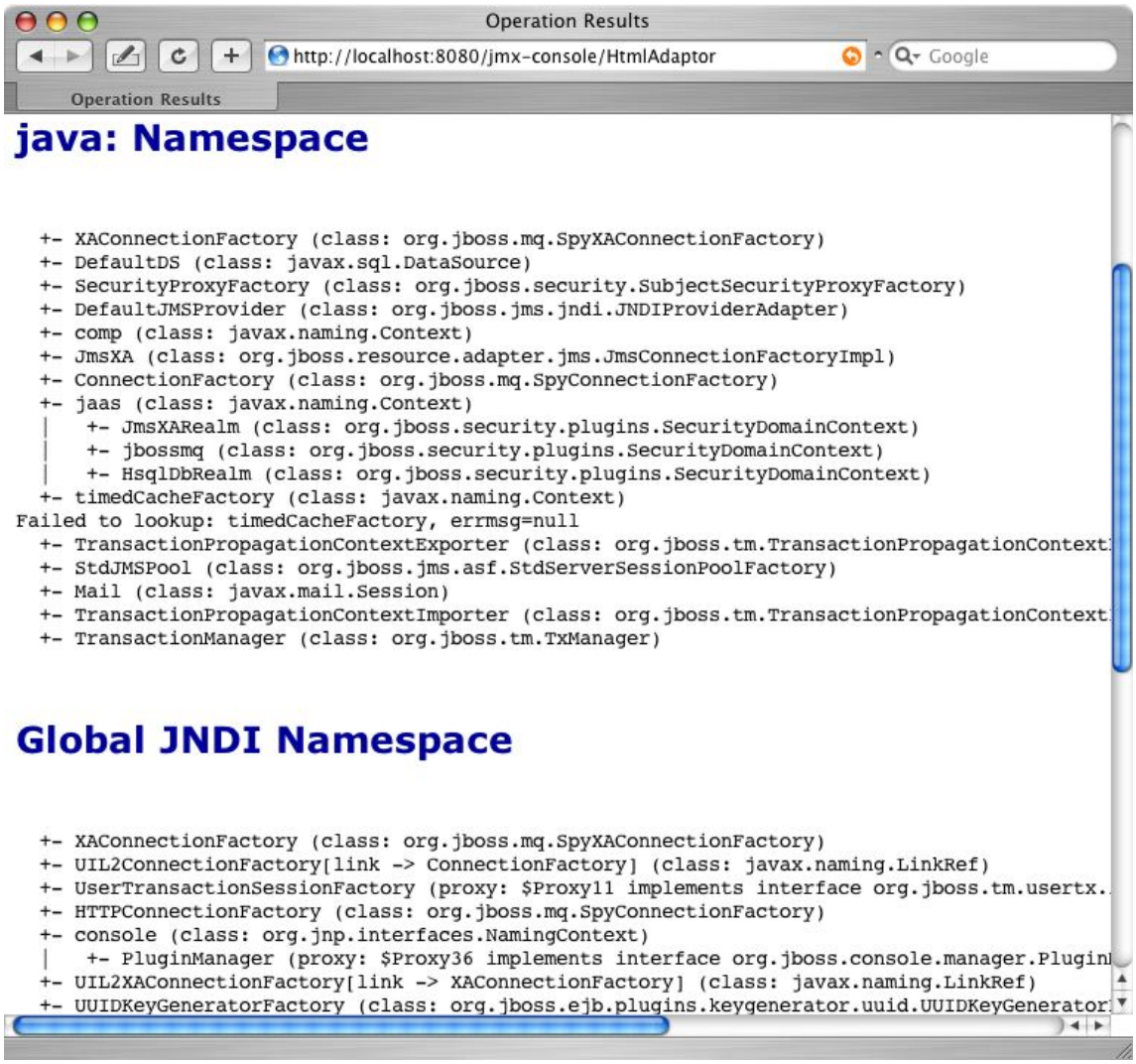


Figure 6.6. The JMX Console view of the JNDIView list operation output

6.6. J2EE AND JNDI - THE APPLICATION COMPONENT ENVIRONMENT

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

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

- The component container uses the deployment package information to build the complete component ENC at runtime

The complete specification regarding the use of JNDI in the J2EE platform can be found in section 5 of the J2EE 1.4 specification.

An application component instance locates the ENC using the JNDI API. An application component instance creates a **`javax.naming.InitialContext`** object by using the no argument constructor and then looks up the naming environment under the name **`java:comp/env`**. The application component's environment entries are stored directly in the ENC, or in its subcontexts. Example 6.4, "ENC access sample code" illustrates the prototypical lines of code a component uses to access its ENC.

Example 6.4. ENC access sample code

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

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

There are three commonly used levels of naming scope in JBoss: names under **`java:comp`**, names under **`java:`**, and any other name. As discussed, the **`java:comp`** context and its subcontexts are only available to the application component associated with that particular context. Subcontexts and object bindings directly under **`java:`** are only visible within the JBoss server virtual machine and not to remote clients. Any other context or object binding is available to remote clients, provided the context or object supports serialization. You'll see how the isolation of these naming scopes is achieved in the Section 6.2, "The JBoss Naming Service Architecture".

An example of where the restricting a binding to the **`java:`** context is useful would be a **`javax.sql.DataSource`** connection factory that can only be used inside of the JBoss server where the associated database pool resides. On the other hand, an EJB home interface would be bound to a globally visible name that should accessible by remote client.

6.6.1. ENC Usage Conventions

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

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

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

6.6.1.1. Environment Entries

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

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

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

An example of an **`env-entry`** fragment from an **`ejb-jar.xml`** deployment descriptor is given in Example 6.5, “An example ejb-jar.xml env-entry fragment” . There is no JBoss specific deployment descriptor element because an **`env-entry`** is a complete name and value specification. Example 6.6, “ENC env-entry access code fragment” shows a sample code fragment for accessing the **`maxExemptions`** and **`taxRate`** and **`env-entry`** values declared in the deployment descriptor.

Example 6.5. An example ejb-jar.xml env-entry fragment

```
<!-- ... -->
<session>
    <ejb-name>ASessionBean</ejb-name>
    <!-- ... -->
    <env-entry>
        <description>The maximum number of tax exemptions allowed
</description>
        <env-entry-name>maxExemptions</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>15</env-entry-value>
    </env-entry>
    <env-entry>
        <description>The tax rate </description>
        <env-entry-name>taxRate</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>0.23</env-entry-value>
    </env-entry>
</session>
<!-- ... -->
```

Example 6.6. ENC env-entry access code fragment

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

6.6.1.2. EJB References

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

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

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

- An optional **description** element that provides the purpose of the reference.
- An **ejb-ref-name** element that specifies the name of the reference relative to the **`java:comp/env`** context. To place the reference under the recommended **`java:comp/env/ejb`** context, use an **`ejb/link-name`** form for the **`ejb-ref-name`** value.

- An **ejb-ref-type** element that specifies the type of the EJB. This must be either `Entity` or `Session`.
- A **home** element that gives the fully qualified class name of the EJB home interface.
- A **remote** element that gives the fully qualified class name of the EJB remote interface.
- An optional **ejb-link** element that links the reference to another enterprise bean in the same EJB JAR or in the same J2EE application unit. The `ejb-link` value is the `ejb-name` of the referenced bean. If there are multiple enterprise beans with the same `ejb-name`, the value uses the path name specifying the location of the `ejb-jar` file that contains the referenced component. The path name is relative to the referencing `ejb-jar` file. The Application Assembler appends the `ejb-name` of the referenced bean to the path name separated by `#`. This allows multiple beans with the same name to be uniquely identified.

An EJB reference is scoped to the application component whose declaration contains the `ejb-ref` element. This means that the EJB reference is not accessible from other application components at runtime, and that other application components may define `ejb-ref` elements with the same `ejb-ref-name` without causing a name conflict. Example 6.7, "An example ejb-jar.xml ejb-ref descriptor fragment" provides an `ejb-jar.xml` fragment that illustrates the use of the `ejb-ref` element. A code sample that illustrates accessing the **ShoppingCartHome** reference declared in Example 6.7, "An example ejb-jar.xml ejb-ref descriptor fragment" is given in Example 6.8, "ENC ejb-ref access code fragment".

Example 6.7. An example ejb-jar.xml ejb-ref descriptor fragment

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

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

</session>

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

```
    </ejb-ref>
</session>

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

<!--...-->
```

Example 6.8. ENC ejb-ref access code fragment

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

6.6.1.3. EJB References with `jboss.xml` and `jboss-web.xml`

The JBoss specific **`jboss.xml`** EJB deployment descriptor affects EJB references in two ways. First, the **`jndi-name`** child element of the **`session`** and **`entity`** elements allows the user to specify the deployment JNDI name for the EJB home interface. In the absence of a **`jboss.xml`** specification of the **`jndi-name`** for an EJB, the home interface is bound under the **`ejb-jar.xmlejb-name`** value. For example, the session EJB with the **`ejb-name`** of **`ShoppingCartBean`** in Example 6.7, "An example ejb-jar.xml ejb-ref descriptor fragment" would have its home interface bound under the JNDI name **`ShoppingCartBean`** in the absence of a **`jboss.xmljndi-name`** specification.

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

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

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

Example 6.9, "An example jboss.xml ejb-ref fragment" provides an example **`jboss.xml`** descriptor fragment that illustrates the following usage points:

- The **`ProductBeanUserejb-ref`** link destination is set to the deployment name of **`jboss/store/ProductHome`**
- The deployment JNDI name of the **`ProductBean`** is set to **`jboss/store/ProductHome`**

Example 6.9. An example jboss.xml ejb-ref fragment

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

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

6.6.1.4. EJB Local References

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

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

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

ejb-name of the referenced bean to the path name separated by #. This allows multiple beans with the same name to be uniquely identified. An **ejb-link** element must be specified in JBoss to match the local reference to the corresponding EJB.

An EJB local reference is scoped to the application component whose declaration contains the **ejb-local-ref** element. This means that the EJB local reference is not accessible from other application components at runtime, and that other application components may define **ejb-local-ref** elements with the same **ejb-ref-name** without causing a name conflict. Example 6.10, "An example ejb-jar.xml ejb-local-ref descriptor fragment" provides an **ejb-jar.xml** fragment that illustrates the use of the **ejb-local-ref** element. A code sample that illustrates accessing the **ProbeLocalHome** reference declared in Example 6.10, "An example ejb-jar.xml ejb-local-ref descriptor fragment" is given in Example 6.11, "ENC ejb-local-ref access code fragment" .

Example 6.10. An example ejb-jar.xml ejb-local-ref descriptor fragment

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

Example 6.11. ENC ejb-local-ref access code fragment

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

6.6.1.5. Resource Manager Connection Factory References

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

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

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

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

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

Example 6.12, “A web.xml resource-ref descriptor fragment” shows an example **`web.xml`** descriptor fragment that illustrates the **`resource-ref`** element usage. Example 6.13, “ENC resource-ref access sample code fragment” provides a code fragment that an application component would use to access the **`DefaultMail`** resource declared by the **`resource-ref`**.

Example 6.12. A web.xml resource-ref descriptor fragment

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

Example 6.13. ENC resource-ref access sample code fragment

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

6.6.1.6. Resource Manager Connection Factory References with jboss.xml and jboss-web.xml

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

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

JBoss

- A **res-url** element that specifies the URL string in the case of a `resource-ref` of type `java.net.URL`

Example 6.14, "A sample jboss-web.xml resource-ref descriptor fragment" provides a sample `jboss-web.xml` descriptor fragment that shows sample mappings of the `resource-ref` elements given in Example 6.12, "A web.xml resource-ref descriptor fragment" .

Example 6.14. A sample jboss-web.xml resource-ref descriptor fragment

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

6.6.1.7. Resource Environment References

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

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

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

Example 6.15, "An example ejb-jar.xml resource-env-ref fragment" provides an example **resource-ref-env** element declaration by a session bean. Example 6.16, "ENC resource-env-ref access code fragment" gives a code fragment that illustrates how to look up the **StockInfo** queue declared by the **resource-env-ref**.

Example 6.15. An example ejb-jar.xml resource-env-ref fragment

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

Example 6.16. ENC resource-env-ref access code fragment

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

6.6.1.8. Resource Environment References and jboss.xml, jboss-web.xml

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

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

Example 6.17, "A sample jboss.xml resource-env-ref descriptor fragment" provides a sample **jboss.xml** descriptor fragment that shows a sample mapping for the **StockInforesource-env-ref**.

Example 6.17. A sample jboss.xml resource-env-ref descriptor fragment

```
<session>
    <ejb-name>MyBean</ejb-name>
    <!-- ... -->
    <resource-env-ref>
        <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
```

```
        <jndi-name>queue/StockInfoQueue</jndi-name>
    </resource-env-ref>
    <!-- ... -->
</session>
```

CHAPTER 7. 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.

WARNING

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

7.1. THE NEED FOR WEB SERVICES

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

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

7.2. 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.

7.3. DOCUMENT/LITERAL

With document style web services two business partners agree on the exchange of complex business documents that are well defined in XML schema. For example, one party sends a document describing a purchase order, the other responds (immediately or later) with a document that describes the status of the purchase order. The payload of the SOAP message is an XML document that can be validated against XML schema. The document is defined by the style attribute on the SOAP binding.

```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
   <soap:binding style='document'
transport='http://schemas.xmlsoap.org/soap/http'/>
   <operation name='concat'>
      <soap:operation soapAction=''/>
      <input>
         <soap:body use='literal'/>
      </input>
      <output>
         <soap:body use='literal'/>
      </output>
   </operation>
</binding>
```

With document style web services the payload of every message is defined by a complex type in XML schema.

```
<complexType name='concatType'>
   <sequence>
      <element name='String_1' nillable='true' type='string'/>
      <element name='long_1' type='long'/>
   </sequence>
</complexType>
<element name='concat' type='tns:concatType'/>
```

Therefore, message parts must refer to an element from the schema.

```
<message name='EndpointInterface_concat'>
   <part name='parameters' element='tns:concat'/>
</message>
```

The following message definition is invalid.

```
<message name='EndpointInterface_concat'>
   <part name='parameters' type='tns:concatType'/>
</message>
```

7.4. DOCUMENT/LITERAL (BARE)

Bare is an implementation detail from the Java domain. Neither in the abstract contract (for instance, wsdl+schema) nor at the SOAP message level is a bare endpoint recognizable. A bare endpoint or client uses a Java bean that represents the entire document payload.

```
@WebService
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
```

```
public class DocBareServiceImpl
{
   @WebMethod
   public SubmitBareResponse submitPO(SubmitBareRequest poRequest)
   {
      ...
   }
}
```

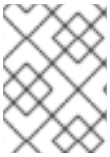
The trick is that the Java beans representing the payload contain JAXB annotations that define how the payload is represented on the wire.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SubmitBareRequest",
namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/", propOrder
= { "product" })
@XmlRootElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss.
org/", name = "SubmitPO")
public class SubmitBareRequest
{

@XmlElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/
",  required = true)
   private String product;

   ...
}
```

7.5. DOCUMENT/LITERAL (WRAPPED)

Wrapped is an implementation detail from the Java domain. Neither in the abstract contract (for instance, wsdl+schema) nor at the SOAP message level is a wrapped endpoint recognizable. A wrapped endpoint or client uses the individual document payload properties. Wrapped is the default and does not have to be declared explicitly.

```
@WebService
public class DocWrappedServiceImpl
{
   @WebMethod
   @RequestWrapper (className="org.somepackage.SubmitPO")
   @ResponseWrapper (className="org.somepackage.SubmitPOResponse")
   public String submitPO(String product, int quantity)
   {
   ...
   }
}
```



NOTE

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

7.6. RPC/LITERAL

With RPC there is a wrapper element that names the endpoint operation. Child elements of the RPC parent are the individual parameters. The SOAP body is constructed based on some simple rules:

- The port type operation name defines the endpoint method name
- Message parts are endpoint method parameters

RPC is defined by the style attribute on the SOAP binding.

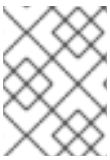
```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
   <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http'/>
   <operation name='echo'>
      <soap:operation soapAction=''/>
      <input>
         <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
use='literal'/>
      </input>
      <output>
         <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
use='literal'/>
      </output>
   </operation>
</binding>
```

With RPC style web services the portType names the operation (i.e. the java method on the endpoint)

```
<portType name='EndpointInterface'>
   <operation name='echo' parameterOrder='String_1'>
      <input message='tns:EndpointInterface_echo'/>
      <output message='tns:EndpointInterface_echoResponse'/>
   </operation>
</portType>
```

Operation parameters are defined by individual message parts.

```
<message name='EndpointInterface_echo'>
   <part name='String_1' type='xsd:string'/>
</message>
<message name='EndpointInterface_echoResponse'>
   <part name='result' type='xsd:string'/>
</message>
```



NOTE

There is no complex type in XML schema that could validate the entire SOAP message payload.

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
   @WebMethod
   @WebResult(name="result")
```

```
    public String echo(@WebParam(name="String_1") String input)
    {
    ...
    }
}
```

The element names of RPC parameters/return values may be defined using the JAX-WS Annotations#javax.jws.WebParam and JAX-WS Annotations#javax.jws.WebResult respectively.

7.7. RPC/ENCODED

SOAP encoding style is defined by chapter 5 of the SOAP-1.1 specification. It has inherent interoperability issues that cannot be fixed. The Basic Profile-1.0 prohibits this encoding style in 4.1.7 SOAP encodingStyle Attribute. JBossWS has basic support for RPC/Encoded that is provided as is for simple interop scenarios with SOAP stacks that do not support literal encoding. Specifically, JBossWS does not support:-

- element references
- soap arrays as bean properties

NOTE

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

7.8. WEB SERVICE ENDPOINTS

JAX-WS simplifies the development model for a web service endpoint a great deal. In short, an endpoint implementation bean is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes the abstract contract (for instance, wsdl+schema) for client consumption. All marshalling/unmarshalling is delegated to JAXB.

7.9. PLAIN OLD JAVA OBJECT (POJO)

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

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

7.10. THE ENDPOINT AS A WEB APPLICATION

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

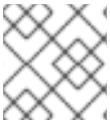
```
<web-app ...>
```

```
   <servlet>
      <servlet-name>TestService</servlet-name>
      <servlet-
class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>TestService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

7.11. PACKAGING THE ENDPOINT

A JSR-181 java service endpoint (JSE) is packaged as a web application in a **`*.war`** file.

```
<war warfile="${build.dir}/libs/jbossws-samples-jsr181pojo.war"
webxml="${build.resources.dir}/samples/jsr181pojo/WEB-INF/web.xml">
   <classes dir="${build.dir}/classes">
      <include
name="org/jboss/test/ws/samples/jsr181pojo/JSEBean01.class"/>
   </classes>
</war>
```



NOTE

Only the endpoint implementation bean and **`web.xml`** file are required.

7.12. ACCESSING THE GENERATED WSDL

A successfully deployed service endpoint will show up in the service endpoint manager. This is also where you find the links to the generated WSDL.

```
http://yourhost:8080/jbossws/services
```

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

7.13. EJB3 STATELESS SESSION BEAN (SLSB)

The JAX-WS programming model support the same set of annotations on EJB3 stateless session beans as on Plain old Java Object (POJO) endpoints. EJB-2.1 endpoints are supported using the JAX-RPC progamming model.

```
@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
{
   @WebMethod
```

```
   public String echo(String input)
   {
   ...
   }
}
```

Above you see an EJB-3.0 stateless session bean that exposes one method both on the remote interface and as an endpoint operation.

Packaging the endpoint

A JSR-181 EJB service endpoint is packaged as an ordinary ejb deployment.

```
<jar jarfile="${build.dir}/libs/jbossws-samples-jsr181ejb.jar">
   <fileset dir="${build.dir}/classes">
      <include
name="org/jboss/test/ws/samples/jsr181ejb/EJB3Bean01.class"/>
      <include
name="org/jboss/test/ws/samples/jsr181ejb/EJB3RemoteInterface.class"/>
   </fileset>
</jar>
```

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the service endpoint manager. This is also where you will find the links to the generated WSDL.

```
  http://yourhost:8080/jbossws/services
```

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

7.14. ENDPOINT PROVIDER

JAX-WS services typically implement a native Java service endpoint interface (SEI), perhaps mapped from a WSDL port type, either directly or via the use of annotations.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The Provider interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML message level.

A Provider based service instance's invoke method is called for each message received for the service.

```
@WebServiceProvider
@ServiceMode(value = Service.Mode.PAYLOAD)
public class ProviderBeanPayload implements Provider<Source>
{
   public Source invoke(Source req)
   {
      // Access the entire request PAYLOAD and return the response PAYLOAD
   }
}
```

Service.Mode.PAYLOAD is the default and does not have to be declared explicitly. You can also use Service.Mode.MESSAGE to access the entire SOAP message (for example, with MESSAGE the Provider can also see SOAP Headers)

7.15. WEBSERVICECONTEXT

The **`WebServiceContext`** is treated as an injectable resource that can be set at the time an endpoint is initialized. The **`WebServiceContext`** object will then use thread-local information to return the correct information regardless of how many threads are concurrently being used to serve requests addressed to the same endpoint object.

```
@WebService
public class EndpointJSE
{
   @Resource
   WebServiceContext wsCtx;

   @WebMethod
   public String testGetMessageContext()
   {
      SOAPMessageContext jaxwsContext =
(SOAPMessageContext)wsCtx.getMessageContext();
      return jaxwsContext != null ? "pass" : "fail";
   }
   ...
   @WebMethod
   public String testGetUserPrincipal()
   {
      Principal principal = wsCtx.getUserPrincipal();
      return principal.getName();
   }

   @WebMethod
   public boolean testIsUserInRole(String role)
   {
      return wsCtx.isUserInRole(role);
   }
}
```

7.16. WEB SERVICE CLIENTS

7.16.1. Service

`Service` is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at a particular endpoint address.

For most clients, you will start with a set of stubs generated from the WSDL. One of these will be the service, and you will create objects of that class in order to work with the service (see "static case" below).

7.16.1.1. Service Usage

Static case

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

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

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

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

```
// Generated Service Class

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

   public StockQuoteService()
   {
      super(new URL("http://example.com/stocks.wsdl"), new
QName("http://example.com/stocks", "StockQuoteService"));
   }

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

   ...
}
```

Section Dynamic Proxy explains how to obtain a port from the service and how to invoke an operation on the port. If you need to work with the XML payload directly or with the XML representation of the entire SOAP message, have a look at Dispatch.

Dynamic case

In the dynamic case, when nothing is generated, a web service client uses **`Service.create`** to create Service instances, the following code illustrates this process.

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service service = Service.create(wsdlLocation, serviceName);
```

This is not the recommended way to use JBossWS.

7.16.1.2. Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers,

that may be used to extend the capabilities of a JAX-WS runtime system. Handler Framework describes the handler framework in detail. A **Service** instance provides access to a **HandlerResolver** via a pair of **getHandlerResolver** and **setHandlerResolver** methods that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a **Service** instance is used to create a proxy or a **Dispatch** instance then the handler resolver currently registered with the service is used to create the required handler chain. Subsequent changes to the handler resolver configured for a **Service** instance do not affect the handlers on previously created proxies, or **Dispatch** instances.

7.16.1.3. Executor

Service instances can be configured with a **java.util.concurrent.Executor**. The executor will then be used to invoke any asynchronous callbacks requested by the application. The **setExecutor** and **getExecutor** methods of **Service** can be used to modify and retrieve the executor configured for a service.

7.16.2. Dynamic Proxy

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

```
/**
 * The getPort method returns a proxy. A service client
 * uses this proxy to invoke operations on the target
 * service endpoint. The <code>serviceEndpointInterface</code>
 * specifies the service endpoint interface that is supported by
 * the created dynamic proxy instance.
 */
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
{
...
}

/**
 * The getPort method returns a proxy. The parameter
 * <code>serviceEndpointInterface</code> specifies the service
 * endpoint interface that is supported by the returned proxy.
 * In the implementation of this method, the JAX-WS
 * runtime system takes the responsibility of selecting a protocol
 * binding (and a port) and configuring the proxy accordingly.
 * The returned proxy should not be reconfigured by the client.
 *
 */
public <T> T getPort(Class<T> serviceEndpointInterface)
{
...
}
```

The *Service Endpoint Interface* (SEI) is usually generated using tools. For details see Top Down (Using wsconsume).

A generated static **Service** usually also offers typed methods to get ports. These methods also return dynamic proxies that implement the SEI.

```
@WebServiceClient(name = "TestEndpointService", targetNamespace =
"http://org.jboss.ws/wsref",
  wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-
webserviceref?wsdl")
public class TestEndpointService extends Service
{
   ...

   public TestEndpointService(URL wsdlLocation, QName serviceName) {
      super(wsdlLocation, serviceName);
   }

   @WebEndpoint(name = "TestEndpointPort")
   public TestEndpoint getTestEndpointPort()
   {
      return (TestEndpoint)super.getPort(TESTENDPOINTPORT,
TestEndpoint.class);
   }
}
```

7.16.3. WebServiceRef

The **`WebServiceRef`** annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the **`javax.annotation.Resource`** annotation in JSR-250 [5]

There are two uses to the **`WebServiceRef`** annotation:

1. To define a reference whose type is a generated service class. In this case, the type and value element will both refer to the generated service class type. Moreover, if the reference type can be inferred by the field or method declaration then the annotation is applied to the type, and value elements *may* have the default value (**`Object.class`**, that is). If the type cannot be inferred, then at least the type element *must* be present with a non-default value.

2. To define a reference whose type is a SEI. In this case, the type element *may* be present with its default value if the type of the reference can be inferred from the annotated field and method declaration, but the value element *must* always be present and refer to a generated service class type (a subtype of **`javax.xml.ws.Service`**). The wsdlLocation element, if present, overrides 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>
```

7.16.4. Dispatch

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

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

Message

In this mode, client applications work directly with protocol-specific message structures. For example, when used with a SOAP protocol binding, a client application would work directly with a SOAP message.

Message Payload

In this mode, client applications work with the payload of messages rather than the messages themselves. For example, when used with a SOAP protocol binding, a client application would work with the contents of the SOAP Body rather than the SOAP message as a whole.

Dispatch is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. Dispatch is a generic class that supports input and output of messages or message payloads of any type.

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class,
Mode.PAYLOAD);

String payload = "<ns1:ping
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new
StringReader(payload)));
```

7.16.5. Asynchronous Invocations

The **`BindingProvider`** interface represents a component that provides a protocol binding for use by clients, it is implemented by proxies and is extended by the **`Dispatch`** interface.

`BindingProvider` instances may provide asynchronous operation capabilities. When used, asynchronous operation invocations are decoupled from the **`BindingProvider`** instance at invocation time such that the response context is not updated when the operation completes. Instead a separate response context is made available using the **`Response`** interface.

```
public void testInvokeAsync() throws Exception
{
   URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-
samples-asynchronous?wsdl");
   QName serviceName = new QName(targetNS, "TestEndpointService");
   Service service = Service.create(wsdlURL, serviceName);
   TestEndpoint port = service.getPort(TestEndpoint.class);

   Response response = port.echoAsync("Async");

   // access future
   String retStr = (String) response.get();
   assertEquals("Async", retStr);
}
```

7.16.6. Oneway Invocations

@Oneway indicates that the given web method has only an input message and no output. Typically, a one-way method returns the thread of control to the calling application prior to executing the actual business method.

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
   private static String feedback;
   ...
   @WebMethod
   @Oneway
   public void ping()
   {
      log.info("ping");
      feedback = "ok";
   }
   ...
   @WebMethod
   public String feedback()
   {
   log.info("feedback");
   return feedback;
   }
}
```

7.17. COMMON API

This sections describes concepts that apply equally to Web Service Endpoints and Web Service Clients

7.17.1. Handler Framework

The handler framework is implemented by a JAX-WS protocol binding in both client and server side runtimes. Proxies, and Dispatch instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

7.17.1.1. Logical Handler

Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler`.

7.17.1.2. Protocol Handler

Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message. Protocol handlers are handlers that implement any interface derived from `javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

7.17.1.3. Service endpoint handlers

On the service endpoint, handlers are defined using the @HandlerChain annotation.

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
{
...
}
```

The location of the handler chain file supports 2 formats

1. An absolute java.net.URL in externalForm. (ex: http://myhandlers.foo.com/handlerfile1.xml)

2. A relative path from the source file or class file. (ex: bar/handlerfile1.xml)

7.17.1.4. Service client handlers

On the client side, handler can be configured using the @HandlerChain annotation on the SEI or dynamically using the API.

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain); // important!
```

7.17.2. Message Context

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

Properties are scoped as either APPLICATION or HANDLER. All properties are available to all handlers associated with particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during the execution. APPLICATION scoped properties are also made available to client applications and service endpoint implementations. The default scope for a property is HANDLER.

7.17.2.1. Accessing the message context

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

7.17.2.2. Logical Message Context

LogicalMessageContext is passed to **Logical Handlers** at invocation time. LogicalMessageContext extends MessageContext with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of a message. A protocol binding defines what component of a message are available via a logical message context. The SOAP binding defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers whereas the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

7.17.2.3. SOAP Message Context

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

7.17.3. Fault Handling

An implementation may throw a SOAPFaultException

```
public void throwSoapFaultException()
{
   SOAPFactory factory = SOAPFactory.newInstance();
   SOAPFault fault = factory.createFault("this is a fault string!", new
QName("http://foo", "FooCode"));
   fault.setFaultActor("mr.actor");
   fault.addDetail().addChildElement("test");
   throw new SOAPFaultException(fault);
}
```

or an application specific user exception

```
public void throwApplicationException() throws UserException
{
   throw new UserException("validation", 123, "Some validation error");
}
```

NOTE

In case of the latter JBossWS generates the required fault wrapper beans at runtime if they are not part of the deployment

7.18. DATABINDING

7.18.1. Using JAXB with non annotated classes

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

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

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

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

Complete documentation can be found here:

- JAXB Introductions

7.19. ATTACHMENTS

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

7.19.1. MTOM/XOP

This section describes Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP), a means of more efficiently serializing XML Infosets that have certain types of content. The related specifications are

- SOAP Message Transmission Optimization Mechanism (MTOM)
- XML-binary Optimized Packaging (XOP)

7.19.1.1. Supported MTOM parameter types

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

The above table shows a list of supported endpoint parameter types. The recommended approach is to use the javax.activation.DataHandler classes to represent binary data as service endpoint parameters.

NOTE

Microsoft endpoints tend to send any data as application/octet-stream. The only Java type that can easily cope with this ambiguity is javax.activation.DataHandler

7.19.1.2. Enabling MTOM per endpoint

On the server side MTOM processing is enabled through the **`@BindingType`** annotation. JBossWS does handle SOAP1.1 and SOAP1.2. Both come with or without MTOM flavours:

MTOM enabled service implementations

```
package org.jboss.test.ws.jaxws.samples.xop.doclit;

import javax.ejb.Remote;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.BindingType;

@Remote
@WebService(targetNamespace = "http://org.jboss.ws/xop/doclit")
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT, parameterStyle =
SOAPBinding.ParameterStyle.BARE)
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
(1)
public interface MTOMEndpoint
{
...
}
```

1. The MTOM enabled SOAP 1.1 binding ID

MTOM enabled clients

Web service clients can use the same approach described above or rely on the **`Binding`** API to enable MTOM (Excerpt taken from the **`org.jboss.test.ws.jaxws.samples.xop.doclit.XOPTestCase`**):

```
...
Service service = Service.create(wsdlURL, serviceName);
port = service.getPort(MTOMEndpoint.class);

// enable MTOM
binding = (SOAPBinding)((BindingProvider)port).getBinding();
binding.setMTOMEnabled(true);
```

NOTE

You might as well use the JBossWS configuration templates to setup deployment defaults.

7.19.2. SwaRef

WS-I Attachment Profile 1.0 defines mechanism to reference MIME attachment parts using swaRef. In this mechanism the content of XML element of type 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.

7.19.2.1. Using SwaRef with JAX-WS endpoints

JAX-WS endpoints delegate all marshalling/unmarshalling to the JAXB API. The most simple way to enable SwaRef encoding for **`DataHandler`** types is to annotate a payload bean with the **`@XmlAttachmentRef`** annotation as shown below:

```
/**
 * Payload bean that will use SwaRef encoding
```

```
 */
@XmlRootElement
public class DocumentPayload
{
   private DataHandler data;

   public DocumentPayload()
   {
   }

   public DocumentPayload(DataHandler data)
   {
      this.data = data;
   }

   @XmlElement
   @XmlAttachmentRef
   public DataHandler getData()
   {
      return data;
   }

   public void setData(DataHandler data)
   {
      this.data = data;
   }
}
```

With document wrapped endpoints you may even specify the **`@XmlAttachmentRef`** annotation on the service endpoint interface:

```
@WebService
public interface DocWrappedEndpoint
{
   @WebMethod
   DocumentPayload beanAnnotation(DocumentPayload dhw, String test);

   @WebMethod
   @XmlAttachmentRef
   DataHandler parameterAnnotation(@XmlAttachmentRef DataHandler data,
String test);

}
```

The message would then refer to the attachment part by CID:

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
   <env:Header/>
   <env:Body>
      <ns2:parameterAnnotation
xmlns:ns2='http://swaref.samples.jaxws.ws.test.jboss.org/'>
         <arg0>cid:0-1180017772935-32455963@ws.jboss.org</arg0>
         <arg1>Wrapped test</arg1>
```

```
      </ns2:parameterAnnotation>
   </env:Body>
</env:Envelope>
```

7.19.2.2. Starting from WSDL

If you chose the contract first approach then you need to ensure that any element declaration that should use SwaRef encoding simply refers to wsi:swaRef schema type:

```
<element name="data" type="wsi:swaRef"
xmlns:wsi="http://ws-i.org/profiles/basic/1.1/xsd"/>
```

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

7.20. TOOLS

The JAX-WS tools provided by JBossWS can be used in a variety of ways. First we will look at server-side development strategies, and then proceed to the client. When developing a Web Service Endpoint (the server-side) you have the option of starting from Java (bottom-up development), or from the 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 without breaking compatibility with older clients
- Exposing a service that conforms to a contract specified by a third party (e.g. a vender that calls you back using an already defined protocol).
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front

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

Command	Description
wsprovide	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
wsconsume	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development

wsrunclient	Executes a Java client (that has a main method) using the JBossWS classpath.

7.20.1. Bottom-Up (Using wsprovide)

The bottom-up strategy involves developing the Java code for your service, and then annotating it using JAX-WS annotations. These annotations can be used to customize the contract that is generated for your service. For example, you can change the operation name to map to anything you like. However, all of the annotations have sensible defaults, so only the @WebService annotation is required.

This can be as simple as creating a single class:

```
package echo;

@javax.jws.WebService
public class Echo
{
   public String echo(String input)
   {
      return input;
   }
}
```

A JSE or EJB3 deployment can be built using this class, and it is the only Java code needed to deploy on JBossWS. The WSDL, and all other Java artifacts called "wrapper classes" will be generated for you at deploy time. This actually goes beyond the JAX-WS specification, which requires that wrapper classes be generated using an offline tool. The reason for this requirement is purely a vender implementation problem, and since we do not believe in burdening a developer with a bunch of additional steps, we generate these as well. However, if you want your deployment to be portable to other application servers, you will need to use a tool and add the generated classes to your deployment.

This is the primary purpose of the wsprovide tool, to generate portable JAX-WS artifacts. Additionally, it can be used to "provide" the abstract contract (WSDL file) for your service. This can be obtained by invoking wsprovide using the "-w" option:

```
$ javac -d . -classpath jboss-jaxws.jar Echo.java
$ wsprovide -w echo.Echo
Generating WSDL:
EchoService.wsdl
Writing Classes:
echo/jaxws/Echo.class
echo/jaxws/EchoResponse.class
```

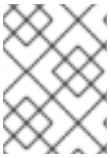
Inspecting the WSDL reveals a service called EchoService:

```
<service name='EchoService'>
   <port binding='tns:EchoBinding' name='EchoPort'>
      <soap:address location='REPLACE_WITH_ACTUAL_URL'/>
   </port>
</service>
```

As expected, this service defines one operation, "echo":

-

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



NOTE

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

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

At deploy time JBossWS will internally invoke wsprovide, which will generate the WSDL. If deployment was successful, and you are using the default settings, it should be available here: http://localhost:8080/echo/Echo?wsdl

For a portable JAX-WS deployment, the wrapper classes generated earlier could be added to the deployment.

7.20.2. Top-Down (Using wsconsume)

The top-down development strategy begins with the abstract contract for the service, which includes the WSDL file and zero or more schema files. The wsconsume tool is then used to consume this contract, and produce annotated Java classes (and optionally sources) that define it.

NOTE

wsconsume seems to have a problem with symlinks on unix systems

Using the WSDL file from the bottom-up example, a new Java implementation that adheres to this service can be generated. The "-k" option is passed to wsconsume to preserve the Java source files that are generated, instead of providing just classes:

```
$ wsconsume -k EchoService.wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The following table shows the purpose of each generated file:

File	Purpose
Echo.java	Service Endpoint Interface
Echo_Type.java	Wrapper bean for request message
EchoResponse.java	Wrapper bean for response message
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
EchoService.java	Used only by JAX-WS clients

Examining the Service Endpoint Interface reveals annotations that are more explicit than in the class written by hand in the bottom-up example, however, these evaluate to the same contract:

```
@WebService(name = "Echo", targetNamespace = "http://echo/")
public interface Echo
{
   @WebMethod
   @WebResult(targetNamespace = "")
   @RequestWrapper(localName = "echo", targetNamespace = "http://echo/",
className = "echo.Echo_Type")
   @ResponseWrapper(localName = "echoResponse", targetNamespace =
"http://echo/", className = "echo.EchoResponse")
   public String echo(@WebParam(name = "arg0", targetNamespace = "")
String arg0);
}
```

The only missing piece (besides the packaging) is the implementation class, which can now be written using the above interface.

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo
{
   public String echo(String arg0)
   {
      return arg0;
   }
}
```

7.20.3. Client Side

Before going into detail on the client-side it is important to understand the decoupling concept that is central to Web Services. Web Services are not the best fit for internal RPC, even though they can be used in this way; there are much better technologies for achieving this (CORBA, and RMI for example). Web Services were designed specifically for interoperable coarse-grained correspondence. There is no expectation or guarantee that any party participating in a Web Service interaction will be at any particular location, running on any particular operating system, or written in any particular programming language. So because of this, it is important to clearly separate client and server implementations. The only thing they should have in common is the abstract contract definition. If, for whatever reason, your software does not adhere to this principal, then you should not be using Web Services. For the above reasons, the ***recommended methodology for developing a client is***to follow ***the top-down approach*** , even if the client is running on the same server.

Let's repeat the process of the top-down section, although using the deployed WSDL, instead of the one generated offline by wsprovide. The reason why we do this is just to get the right value for soap:address. This value must be computed at deploy time, since it is based on container configuration specifics. You could of course edit the WSDL file yourself, although you need to ensure that the path is correct.

Offline version:

```
<service name='EchoService'>
   <port binding='tns:EchoBinding' name='EchoPort'>
      <soap:address location='REPLACE_WITH_ACTUAL_URL'/>
```

```
   </port>
</service>
```

Online version:

```
<service name="EchoService">
   <port binding="tns:EchoBinding" name="EchoPort">
      <soap:address
location="http://localhost.localdomain:8080/echo/Echo"/>
   </port>
</service>
```

Using the online deployed version with wsconsume:

```
$ wsconsume -k http://localhost:8080/echo/Echo?wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The one class that was not examined in the top-down section, was **`EchoService.java`**. Notice how it stores the location the WSDL was obtained from.

```
@WebServiceClient(name = "EchoService", targetNamespace = "http://echo/",
wsdlLocation = "http://localhost:8080/echo/Echo?wsdl")
public class EchoService extends Service
{
   private final static URL ECHOSERVICE_WSDL_LOCATION;

   static
   {
      URL url = null;
      try
      {
         url = new URL("http://localhost:8080/echo/Echo?wsdl");
      }
      catch (MalformedURLException e)
      {
         e.printStackTrace();
      }
      ECHOSERVICE_WSDL_LOCATION = url;
   }

   public EchoService(URL wsdlLocation, QName serviceName)
   {
      super(wsdlLocation, serviceName);
   }
```

```
   public EchoService()
   {
      super(ECHOSERVICE_WSDL_LOCATION, new QName("http://echo/",
"EchoService"));
   }

   @WebEndpoint(name = "EchoPort")
   public Echo getEchoPort()
   {
      return (Echo)super.getPort(new QName("http://echo/", "EchoPort"),
Echo.class);
   }
}
```

As you can see, this generated class extends the main client entry point in JAX-WS, **`javax.xml.ws.Service`**. While you can use **`Service`** directly, this is far simpler since it provides the configuration info for you. The only method we really care about is the **`getEchoPort()`** method, which returns an instance of our **`Service Endpoint Interface`**. Any Web Services operation can then be called by just invoking a method on the returned interface.

NOTE

It is not recommended to refer to a remote WSDL URL in a production application. This causes network I/O every time you instantiate the Service Object. Instead, use the tool on a saved local copy, or use the URL version of the constructor to provide a new WSDL location.

All that is left to do, is write and compile the client:

```
import echo.*;
..
public class EchoClient
{
   public static void main(String args[])
   {
      if (args.length != 1)
      {
         System.err.println("usage: EchoClient <message>");
         System.exit(1);
      }

      EchoService service = new EchoService();
      Echo echo = service.getEchoPort();
      System.out.println("Server said: " + echo.echo(args[0]));
   }
}
```

It can then be easily executed using the wsrunclient tool. This is just a convenience tool that invokes java with the needed classpath:

```
$ wsrunclient EchoClient 'Hello World!'
Server said: Hello World!
```

It is easy to change the endpoint address of your operation at runtime, setting ENDPOINT_ADDRESS_PROPERTY as shown below:

```
...
EchoService service = new EchoService();
Echo echo = service.getEchoPort();

/* Set NEW Endpoint Location */
String endpointURL = "http://NEW_ENDPOINT_URL";
BindingProvider bp = (BindingProvider)echo;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
endpointURL);

System.out.println("Server said: " + echo.echo(args[0]));
...
```

7.20.4. Command-line & Ant Task Reference

- wsconsume reference page
- wsprovide reference page
- wsrunclient reference page

7.20.5. JAX-WS binding customization

An introduction to binding customizations:

- http://java.sun.com/webservices/docs/2.0/jaxws/customizations.html

The schema for the binding customization files can be found here:

- binding customization

7.21. WEB SERVICE EXTENSIONS

7.21.1. WS-Addressing

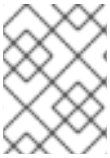
This section describes how WS-Addressing can be used to provide a stateful service endpoint.

7.21.1.1. Specifications

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

- Web Services Addressing 1.0 - Core
- Web Services Addressing 1.0 - SOAP Binding

7.21.1.2. Addressing Endpoint



NOTE

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

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

```
@WebService(name = "StatefulEndpoint", targetNamespace =
"http://org.jboss.ws/samples/wsaddressing", serviceName = "TestService")
@Addressing(enabled=true, required=true)
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class StatefulEndpointImpl implements StatefulEndpoint,
ServiceLifecycle
{
   @WebMethod
   public void addItem(String item)
   { ... }

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

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

It uses the JAX-WS 2.1 defined **`javax.xml.ws.soap.Addressing`** annotation to enable the server side addressing handler.

7.21.1.3. Addressing Client

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

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

A client connecting to the stateful endpoint

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

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

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

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

SOAP message exchange

Below you see the SOAP messages that are 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:Actio
n>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</env:Header>
<env:Body>
<ns1:addItemResponse xmlns:ns1='http://org.jboss.ws/samples/wsaddr'/>
</env:Body>
</env:Envelope>

...

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>uri:jbossws-samples-wsaddr/TestService</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/action</wsa:Action>
<wsa:ReferenceParameters>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</wsa:ReferenceParameters>
</env:Header>
<env:Body>
<ns1:getItems xmlns:ns1='http://org.jboss.ws/samples/wsaddr'/>
</env:Body>
</env:Envelope>
```

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/actionReply</wsa:Actio
n>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</env:Header>
<env:Body>
<ns1:getItemsResponse xmlns:ns1='http://org.jboss.ws/samples/wsaddr'>
<result>[Ice Cream, Ferrari]</result>
</ns1:getItemsResponse>
</env:Body>
</env:Envelope>
```

7.21.2. WS-Security

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

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

Specifications

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

- SOAP Message Security 1.0
- Username Token Profile 1.0
- X.509 Token Profile 1.0
- W3C XML Encryption
- W3C XML Signature
- Basic Security Profile 1.0 (Still in Draft)

7.21.2.1. Endpoint configuration

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

NOTE

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

7.21.2.2. Server side WSSE declaration (jboss-wsse-server.xml)

In this example we configure both the client and the server to sign the message body. Both also require

this from each other. So, if you remove either the client or the server security deployment descriptor, you will notice that the other party will throw a fault explaining that the message did not conform to the proper security requirements.

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
(1)  <key-store-file>WEB-INF/wsse.keystore</key-store-file>
(2)  <key-store-password>jbossws</key-store-password>
(3)  <trust-store-file>WEB-INF/wsse.truststore</trust-store-file>
(4)  <trust-store-password>jbossws</trust-store-password>
(5)  <config>
(6)     <sign type="x509v3" alias="wsse"/>
(7)     <requires>
(8)        <signature/>
        </requires>
     </config>
</jboss-ws-security>
```

1. This specifies that the key store we wish to use is **`WEB-INF/wsse.keystore`**, which is located in our war file.

2. This specifies that the store password is "jbossws". Password can be 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 using X.509v3 certificate (a standard certificate). The alias option says that the certificate and key pair to use for signing is in the key store under the "wsse" alias

7. Here we start our optional requires block. This block specifies all security requirements that must be met when the server receives a message.

8. This means that all web services in this war file require the message body to be signed.

By default an endpoint does not use the WS-Security configuration. Users can use proprietary **`@EndpointConfig`** annotation to set the config name. See JAX-WS_Endpoint_Configuration for the list of available config names.

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

7.21.2.3. Client side WSSE declaration (jboss-wsse-client.xml)

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
(1)  <config>
(2)     <sign type="x509v3" alias="wsse"/>
(3)     <requires>
(4)        <signature/>
        </requires>
     </config>
</jboss-ws-security>
```

1. Here we start our root config block. The root config block is the default configuration for all web service clients (Call, Proxy objects).

2. This means that the client must sign the message body of all requests it sends. Type means that we are to use a X.509v3 certificate (a standard certificate). The alias option says that the certificate/key pair to use for signing is in the key store under the "wsse" alias

3. Here we start our optional requires block. This block specifies all security requirements that must be met when the client receives a response.

4. This means that all web service clients must receive signed response messages.

7.21.2.3.1. Client side key store configuration

We did not specify a key store or trust store, because client apps instead use the wsse System properties instead. If this was a web or ejb client (meaning a webservice client in a war or ejb jar file), then we would have specified them in the client descriptor.

Here is an excerpt from the JBossWS samples:

```
<sysproperty key="org.jboss.ws.wsse.keyStore"
value="${tests.output.dir}/resources/jaxrpc/samples/wssecurity/wsse.keysto
re"/>
<sysproperty key="org.jboss.ws.wsse.trustStore"
value="${tests.output.dir}/resources/jaxrpc/samples/wssecurity/wsse.trusts
tore"/>
<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 therefore can not be manipulated in transit.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Header>
<wsse:Security env:mustUnderstand="1" ...>
<wsu:Timestamp wsu:Id="timestamp">...</wsu:Timestamp>
```

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

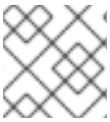
7.21.2.4. Installing the BouncyCastle JCE provider

The information below has originaly been provided by The Legion of the Bouncy Castle .

The provider can be configured as part of your environment via static registration by adding an entry to the **`java.security`** properties file (found in **`$JAVA_HOME/jre/lib/security/java.security`**, where **`$JAVA_HOME`** is the location of your JDK and JRE distribution). You will find detailed instructions in the file but basically it comes down to adding a line:

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

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



NOTE

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

Where users will put the provider jar is mostly up to them, although with jdk5 the best (and in some cases only) place to have it is in **`$JAVA_HOME/jre/lib/ext`**. Under Windows there will normally be a JRE and a JDK install of Java. If user think he have installed it correctly and it still doesn't work then with high probability the provider installation is not used.

7.21.2.5. Username Token Authentication *JBOSSCC-50*

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

Example 7.1. Basic Username Token Configuration

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

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
```

```
instance"
                          xsi:schemaLocation="http://www.jboss.com/ws-
security/config
                          http://www.jboss.com/ws-
security/schema/jboss-ws-security_1_0.xsd">
     <config>
(1)    <username/>
(2)    <timestamp ttl="300"/>
     </config>
</jboss-ws-security>
```

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

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

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                   xsi:schemaLocation="http://www.jboss.com/ws-
security/config
                   http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
     <config>
          <timestamp ttl="300"/>
          <requires/>
     </config>
</jboss-ws-security>
```

WARNING

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

Password Digest, Nonces, and Timestamp

Example 7.1, “Basic Username Token Configuration” results in the client password being sent as plain text. You can use a combination of *digested passwords*, *nonces*, and *timestamps* to provide further protection from replay attacks.

To enable password digesting, you must implement the following items as described in Example 7.2, “Enable Password Digesting”:

Example 7.2. Enable Password Digesting

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

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

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.com/ws-security/config
    http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
  <config>
(3)     <username digestPassword="true" useNonce="true"
useCreated="true"/>
       <timestamp ttl="300"/>
  </config>
</jboss-ws-security>
```

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

Example 7.3. UsernameTokenCallback Module

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

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

- plug the **UsernameTokenCallback** callback into your login module
- extend the **org.jboss.security.auth.spi.UsernamePasswordLoginModule**
- set the hash attributes (***hashAlgorithm***, ***hashEncoding***, ***hashUserPassword***, ***hashStorePassword***) as shown in Example 7.3, “UsernameTokenCallback Module”.

Advanced Tuning - Nonce Factory

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

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

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

Advanced Tuning - Timestamp Verification

If a Timestamp is present in the **`wsse:Security`** header, header verification does not allow for any tolerance whatsoever in the time comparisons. If the message appears to have been created even slightly in the future or if the message has just expired it will be rejected. A new element called <timestamp-verification> is available for the wsse configuration. Example 7.4, "<timestamp-verification> Configuration" describes the required attributes for the <timestamp-verification> element.

Example 7.4. <timestamp-verification> Configuration

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

```
<jboss-ws-security xmlns='http://www.jboss.com/ws-security/config'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance'
                   xsi:schemaLocation='http://www.jboss.com/ws-
security/config
                   http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd'>
  <timestamp-verification createdTolerance="5" warnCreated="false"
expiresTolerance="10" warnExpires="false" />
</jboss-ws-security>
```

`createdTolerance`

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

`expiresTolerance`

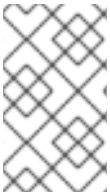
Number of seconds a message is rejected after being classed as expired. The default value is **`0`**.

`warnCreated`

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

`warnExpires`

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



NOTE

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

7.21.2.5.1. Secure Transport

7.21.2.6. X509 Certificate Token*JBOSSCC-50*

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

Encryption

To configure encryption, you must specify the items in Example 7.5, "X509 Encryption Configuration". The configuration is the same for clients and servers.

Example 7.5. X509 Encryption Configuration

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                   xsi:schemaLocation="http://www.jboss.com/ws-
security/config
                   http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
(1) <key-store-file>WEB-INF/bob-sign_enc.jks</key-store-file>
    <key-store-password>password</key-store-password>
    <key-store-type>jks</key-store-type>
    <trust-store-file>WEB-INF/wsse10.truststore</trust-store-file>
    <trust-store-password>password</trust-store-password>

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

The server configuration includes the following encryption information:

1. Keystore and Truststore information: location of each store, the password, and type of store.

2. Signature configuration: you must provide the certificate and key pair aliases to use. ***includeTimestamp*** specifies whether the timestamp is signed to prevent tampering.

3. Encryption configuration: you must provide the certificate and key pair aliases to use. Refer to Algorithms for more information.

4. Optional security requirements: incoming messages must be both signed, and encrypted.

Dynamic Encryption

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

Example 7.6. Dynamic Encryption Configuration

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

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                   xsi:schemaLocation="http://www.jboss.com/ws-
security/config
                   http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
    <key-store-file>WEB-INF/bob-sign_enc.jks</key-store-file>
    <key-store-password>password</key-store-password>
    <key-store-type>jks</key-store-type>
    <trust-store-file>WEB-INF/wsse10.truststore</trust-store-file>
    <trust-store-password>password</trust-store-password>

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

Algorithms

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

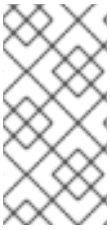
The supported encryption algorithms include:

- AES 128 (aes-128) (default)

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

The supported key-wrap algorithms include:

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



NOTE

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

Encryption Token Reference

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

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

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

NOTE

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

Targets Configuration

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

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

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

Payload Carriage Returns

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

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

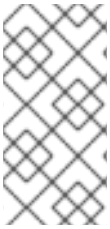
7.21.2.7. JAAS Integration *JBOSSCC-50*

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

Username Token

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

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/ws-security/config
                      http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
  <config>
    <username/>
    <authenticate>
      <usernameAuth/>
    </authenticate>
  </config>
</jboss-ws-security>
```



NOTE

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

X.509 Certificate Token

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

-

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/ws-security/config
                      http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
  <key-store-file>META-INF/bob-sign.jks</key-store-file>
  <key-store-password>password</key-store-password>
  <key-store-type>jks</key-store-type>
  <trust-store-file>META-INF/wsse10.truststore</trust-store-file>
  <trust-store-password>password</trust-store-password>
  <config>
    <sign type="x509v3" alias="1" includeTimestamp="false"/>
    <requires>
      <signature/>
    </requires>
    <authenticate>
(1)   <signatureCertAuth
certificatePrincipal="org.jboss.security.auth.certs.SubjectCNMapping"/>
    </authenticate>
  </config>
</jboss-ws-security>
```

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

The configured security domain must have a correctly configured BaseCertLoginModule, as described in Example 7.7, “BaseCertLoginModule Security Domain”.

Example 7.7. BaseCertLoginModule Security Domain

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

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

The BaseCertLoginModule uses a central keystore to authenticate users. This store is configured through the **org.jboss.security.plugins.JaasSecurityDomain** MBean as shown in Example 7.8, “BaseCertLoginModule Keystore”.

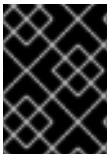
Example 7.8. BaseCertLoginModule Keystore

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

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

7.21.2.8. POJO Endpoint Authentication and Authorization *JBOSSCC-50*

The credentials obtained by WS-Security are generally used for EJB endpoints, or for POJO endpoints when they make a call to another secured resource. It is now possible to enable authentication and authorization checking for POJO endpoints.



IMPORTANT

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

Procedure 7.1. Enabling POJO Authentication and Authorization

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

1. **Define Security Domain in Web Archive**
   You must define a security domain in the WAR containing the POJO.

   Specify a <security-domain> in the jboss-web deployment descriptor within the **`/WEB-INF`** folder.

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

2. **Configure the jboss-wsse-server.xml <authorize> element**
   Specify an <authorize> element within the <config> element.

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

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

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

Unchecked Authentication

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

Example 7.9. Unchecked Authentication

```
<jboss-ws-security>

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

</jboss-ws-security>
```

Role-based Authentication

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

NOTE

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

Example 7.10. Role-based Authentication

```
<jboss-ws-security>

  <config>
    <authorize>
      <role>friend</role>
      <role>family</role>
    </authorize>
  </config>

</jboss-ws-security>
```

7.21.3. XML Registries

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

Section 7.21.3, “XML Registries” describes how to configure the jUDDI registry in JBoss and some sample code outlines for using JAXR API to publish and query the jUDDI registry.

7.21.3.1. Apache jUDDI Configuration

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

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

DataSources configuration

```
<!-- Datasource to Database -->
<attribute name="DataSourceUrl">java:/DefaultDS</attribute>
```

Database Tables (Should they be created on start, Should they be dropped on stop, Should they be dropped on start etc)

```
<!-- Should all tables be created on Start-->
<attribute name="CreateOnStart">false</attribute>
<!-- Should all tables be dropped on Stop-->
<attribute name="DropOnStop">true</attribute>
<!-- Should all tables be dropped on Start-->
<attribute name="DropOnStart">false</attribute>
```

JAXR Connection Factory to be bound in JNDI. (Should it be bound? and under what name?)

```
<!-- Should I bind a Context to which JaxrConnectionFactory bound-->
<attribute name="ShouldBindJaxr">true</attribute>

<!-- Context to which JaxrConnectionFactory to bind to. If you have remote
clients, please bind it to the global namespace(default behavior).
To just cater to clients running on the same VM as JBoss, change to
java:/JAXR -->
<attribute name="BindJaxr">JAXR</attribute>
```

Other common configuration:

Add authorized users to access the jUDDI registry. (Add a sql insert statement in a single line)

```


Look at the script META-INF/ddl/juddi_data.ddl for more details. Example
for a user 'jboss'

INSERT INTO PUBLISHER (PUBLISHER_ID,PUBLISHER_NAME,
EMAIL_ADDRESS,IS_ENABLED,IS_ADMIN)
VALUES ('jboss','JBoss User','jboss@xxx','true','true');
```

7.21.3.2. JBoss JAXR Configuration

In this section, we will discuss the configuration needed to run the JAXR API. The JAXR configuration relies on System properties passed to the JVM. The System properties that are needed are:

-

```
javax.xml.registry.ConnectionFactoryClass=org.apache.ws.scout.registry.
ConnectionFactoryImpl
jaxr.query.url=http://localhost:8080/juddi/inquiry
jaxr.publish.url=http://localhost:8080/juddi/publish
scout.proxy.transportClass=org.jboss.jaxr.scout.transport.SaajTransport
```

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

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

- When the client code is running inside JBoss (maybe a servlet or an EJB). Then you will need to pass the System properties in the **`run.sh`** or **`run.bat`** scripts to the java process via the **`"-D"`** option.
- When the client code is running in an external JVM. Then you can pass the properties either as "-D" options to the java process or explicitly set them in the client code(not recommended).

```
System.setProperty(propertyname, propertyvalue);
```

7.21.3.3. JAXR Sample Code

There are two categories of API: JAXR Publish API and JAXR Inquiry API. The important JAXR interfaces that any JAXR client code will use are the following.

- javax.xml.registry.RegistryService From J2EE 5.0 JavaDoc: "This is the principal interface implemented by a JAXR provider. A registry client can get this interface from a Connection to a registry. It provides the methods that are used by the client to discover various capability specific interfaces implemented by the JAXR provider."
- javax.xml.registry.BusinessLifeCycleManager From J2EE 5.0 JavaDoc: "The **`BusinessLifeCycleManager`** interface, which is exposed by the Registry Service, implements the life cycle management functionality of the Registry as part of a business level API. There is no authentication information provided, because the Connection interface keeps that state and context on behalf of the client."
- javax.xml.registry.BusinessQueryManager From J2EE 5.0 JavaDoc: "The **`BusinessQueryManager`** interface, which is exposed by the Registry Service, implements the business style query interface. It is also referred to as the focused query interface."

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

Getting a JAXR Connection to the registry.

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

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

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

Authentication with the registry.

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

   connection.setCredentials(creds);
}
```

Save a Business

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

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

   User user = blm.createUser();
   org.setPrimaryContact(user);
   PersonName personName = blm.createPersonName("Anil S");
   TelephoneNumber telephoneNumber = blm.createTelephoneNumber();
   telephoneNumber.setNumber("111-111-7777");
   telephoneNumber.setType(null);
   PostalAddress address = blm.createPostalAddress("111", "My Drive",
"BuckHead", "GA", "USA", "1111-111", "");
```

```
   Collection postalAddresses = new ArrayList();
   postalAddresses.add(address);
   Collection emailAddresses = new ArrayList();
   EmailAddress emailAddress = blm.createEmailAddress("anil@apache.org");
   emailAddresses.add(emailAddress);

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

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

   return org;
}
```

Query a Business

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

      // Define find qualifiers and name patterns
      Collection findQualifiers = new ArrayList();
      findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);
      Collection namePatterns = new ArrayList();
      String pattern = "%" + bizname + "%";
      LocalizedString ls = blm.createLocalizedString(Locale.getDefault(),
pattern);
      namePatterns.add(ls);

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

7.21.3.4. Troubleshooting

- **I cannot connect to the registry from JAXR.** Please check the inquiry and publish url passed to the JAXR ConnectionFactory.
- **I cannot connect to the jUDDI registry.** Please check the jUDDI configuration and see if there are any errors in the server.log. And also remember that the jUDDI registry is available only in the "all" configuration.
- **I cannot authenticate to the jUDDI registry.** Have you added an authorized user to the jUDDI database, as described earlier in the chapter?
- **I would like to view the SOAP messages in transit between the client and the UDDI Registry.** Please use the tcpmon tool to view the messages in transit. TCPMon

7.21.3.5. Resources

- JAXR Tutorial and Code Camps
- J2EE 1.4 Tutorial
- J2EE Web Services by Richard Monson-Haefel

7.22. JBOSSWS EXTENSIONS

This section describes propriatary JBoss extensions to JAX-WS.

7.22.1. Proprietary Annotations

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

7.22.1.1. EndpointConfig

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

7.22.1.2. WebContext

```
/**
 * Provides web context specific meta data to EJB based web service
endpoints.
 *
 * @author thomas.diesler@jboss.org
 * @since 26-Apr-2005
 */
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = { ElementType.TYPE })
public @interface WebContext
{
   ...
   /**
    * The contextRoot element specifies the context root that the web
service endpoint is deployed to.
```

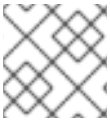
```
    * If it is not specified it will be derived from the deployment short
name.
    *
    * Applies to server side port components only.
    */
   String contextRoot() default "";
   ...
   /**
    * The virtual hosts that the web service endpoint is deployed to.
    *
    * Applies to server side port components only.
    */
   String[] virtualHosts() default {};

   /**
    * Relative path that is appended to the contextRoot to form fully
qualified
    * endpoint address for the web service endpoint.
    *
    * Applies to server side port components only.
    */
   String urlPattern() default "";

   /**
    * The authMethod is used to configure the authentication mechanism for
the web service.
    * As a prerequisite to gaining access to any web service which are
protected by an authorization
    * constraint, a user must have authenticated using the configured
mechanism.
    *
    * Legal values for this element are "BASIC", or "CLIENT-CERT".
    */
   String authMethod() default "";

   /**
    * The transportGuarantee specifies that the communication
    * between client and server should be NONE, INTEGRAL, or
    * CONFIDENTIAL. NONE means that the application does not require any
    * transport guarantees. A value of INTEGRAL means that the application
    * requires that the data sent between the client and server be sent in
    * such a way that it can't be changed in transit. CONFIDENTIAL means
    * that the application requires that the data be transmitted in a
    * fashion that prevents other entities from observing the contents of
    * the transmission. In most cases, the presence of the INTEGRAL or
    * CONFIDENTIAL flag will indicate that the use of SSL is required.
    */
   String transportGuarantee() default "";

   /**
    * A secure endpoint does not by default publish it's wsdl on an
unsecure transport.
    * You can override this behaviour by explicitly setting the
secureWSDLAccess flag to false.
    *
    * Protect access to WSDL. See http://jira.jboss.org/jira/browse/JBWS-
```

```
723
    */
   boolean secureWSDLAccess() default true;
}
```

7.22.1.3. SecurityDomain

```
/**
 * Annotation for specifying the JBoss security domain for an EJB
 */
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface SecurityDomain
{
   /**
    * The required name for the security domain.
    *
    * Do not use the JNDI name
    *
    *    Good: "MyDomain"
    *    Bad:  "java:/jaas/MyDomain"
    */
   String value();

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

7.23. WEB SERVICES APPENDIX



NOTE

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

JAX-WS Endpoint Configuration

JAX-WS Client Configuration

JAX-WS Annotations

7.24. REFERENCES

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

CHAPTER 8. 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 is typically scattered across methods, classes, object hierarchies, or even entire object models. It is behavior that looks and smells like it should have structure, but you can't find a way to express this structure in code with traditional object-oriented techniques.

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

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

Pointcut

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

Introduction

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

Aspect

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

Interceptor

An *interceptor* is an aspect with only one advice, named **`invoke`**. It is a specific interface that you can implement if you want your code to be checked by forcing your class to implement an interface. It also will be portable and can be reused in other JBoss environments like EJBs and JMX MBeans.

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

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

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

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

1. It's extremely difficult to turn metrics on and off, as you have to manually add the code in the **`try/finally`** blocks to each and every method or constructor you want to benchmark.

2. Profiling code should not be combined with your application code. It makes your code more verbose and difficult to read, since the timings must be enclosed within the **`try/finally`** blocks.

3. If you wanted to expand this functionality to include a method or failure count, or even to register these statistics to a more sophisticated reporting mechanism, you'd have to modify a lot of different files (again).

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

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

8.2. CREATING ASPECTS IN JBOSS AOP

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

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

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

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

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

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

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

8.3. APPLYING ASPECTS IN JBOSS AOP

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

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

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

```
1. <bind pointcut="public void com.mc.BankAccountDAO->withdraw(double
```

```
amount)">
2.       <interceptor class="com.mc.Metrics"/>
3. </bind >

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

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

JBoss AOP has a rich set of pointcut expressions that you can use to define various points or events in your Java application. Once your points are defined, you can apply aspects to them. You can attach your aspects to a specific Java class in your application or you can use more complex compositional pointcuts to specify a wide range of classes within one expression.

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

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

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

8.4. PACKAGING AOP APPLICATIONS

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

In the JBoss Enterprise Application Platform 5, you *must* specify the schema used, otherwise your information will not be parsed correctly. You do this by adding the **xmlns="urn:jboss:aop-beans:1:0** attribute to the root **aop** element, as shown here:

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

If you want to create anything more than a non-trivial example, using the **.aop** JAR files, you can make any top-level deployment contain an AOP file containing the XML binding configuration. For

instance you can have an AOP file in an EAR file, or an AOP file in a WAR file. The bindings specified in the **META-INF/jboss-aop.xml** file contained in the AOP file will affect all the classes in the whole WAR file.

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

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

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

IMPORTANT

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

8.5. THE JBOSS ASPECTMANAGER SERVICE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.6. LOADTIME TRANSFORMATION IN THE JBOSS ENTERPRISE APPLICATION PLATFORM USING SUN JDK

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

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

- Set the **enableLoadtimeWeaving** attribute/property to **true**. By default, JBoss Application Server will not do load-time bytecode manipulation of AOP files unless this is set. If **suppressTransformationErrors** is **true**, failed bytecode transformation will only give an error warning. This flag is needed because sometimes a JBoss deployment will not include all of the classes referenced.

- Copy the **pluggable-instrumentor.jar** from the **lib/** directory of your JBoss AOP distribution to the **bin/** directory of your JBoss Enterprise Application Platform.
- Next edit **run.sh** or **run.bat** (depending on what OS you're on) and add the following to the **JAVA_OPTS** environment variable:

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

IMPORTANT

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

8.7. JROCKIT

JRockit also supports the **-javaagent** switch mentioned in Section 8.6, "Loadtime transformation in the JBoss Enterprise Application Platform Using Sun JDK". If you wish to use that, then the steps in Section 8.6, "Loadtime transformation in the JBoss Enterprise Application Platform Using Sun JDK" are sufficient. However, JRockit also comes with its own framework for intercepting when classes are loaded, which might be faster than the **-javaagent** switch. If you want to do load-time transformations using the special JRockit hooks, these are the steps you must take.

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

```
# Setup JBoss specific properties

JAVA_OPTS="$JAVA_OPTS -Dprogram.name=$PROGNAME \

-
Xmanagement:class=org.jboss.aop.hook.JRockitPluggableClassPreProcess
or"

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

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

8.8. IMPROVING LOADTIME PERFORMANCE IN THE JBOSS ENTERPRISE APPLICATION PLATFORM ENVIRONMENT

The same rules apply to the JBoss Enterprise Application Platform for tuning loadtime weaving performance as standalone Java. Switches such as pruning, optimized, include and exclude are configured through the **`jboss-5.x.x.GA/server/xxx/conf/aop.xml`** file talked about earlier in this chapter.

8.9. SCOPING THE AOP TO THE CLASSLOADER

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

8.9.1. Deploying as part of a scoped classloader

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

8.9.2. Attaching to a scoped deployment

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

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

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

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

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

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

CHAPTER 9. TRANSACTION MANAGEMENT

This chapter presents a brief overview of the main configuration options for the JBoss Transaction Service. For more information, please refer to the *JBoss Transactions Administration Guide*.

9.1. OVERVIEW

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

Two additional, optional, JBoss Transaction Service transaction modules are also shipped with JBoss Enterprise Application Platform and may be deployed to provide additional functionality if required.

JBoss Transaction Service JTS

A Transaction Manager capable of distributing transaction context on remote IIOP method calls, creating a single distributed transaction which spans multiple Java Virtual Machines. This is useful for large-scale applications that span multiple servers, or for standards based interoperability with transactional business logic running in CORBA based systems. The functionality of this module can be accessed through the standard JTA API. In this way, it is a drop-in replacement and does not require changes to transactional business logic. To enable it, refer to Section 9.8, “Using the JTS Module” for more information.

JBoss Transaction Service XTS

A Transaction Manager, based on XML, which implements the *WS-AtomicTransaction (WS-AT)* and *WS-BusinessActivity (WS-BA)* specifications. This additional module uses core transaction support provided by the JTA or JTS managers, along with web services functionality provided by JBossWS Native. It is deployed into the server as an application. Applications may use WS-AT to provide standards based, distributed ACID transactions in a manner similar to JTS but using a Web Services transport, instead of CORBA. The WS-BA implementation compliments this by providing an alternative, compensation-based transaction model, well suited to coordinating long-running, loosely coupled business processes. XTS also implements a *WS-Coordination (WS-C)* service which is usually accessed internally by the local WS-AT and WS-BA implementations. However, this WS-C service can also be used to provide remote coordination for WS-AT and WS-BA transactions created in other JBoss server instances or non-JBoss containers. Refer to the JBoss Transactions Web Services Programmer's Guide for more details. To enable XTS, refer to Section 9.9, “Using the XTS Module”.

9.2. CONFIGURATION ESSENTIALS

Configuration of the default JBossTS JTA is managed though a combination of the transaction manager's own properties file and the application server's deployment configuration. The configuration file resides at ***$JBOSS_HOME*/server/[*name*]/conf/jbossts-properties.xml**. It contains defaults for the most commonly used properties. Many more are detailed in the accompanying JBoss Transaction Service Administration Guide. Each setting has a hard-coded default, but the system may not function properly if a configuratino file does not exist. Additional configuratino is also possible as part of the Microcontainer beans configuration found in the ***$JBOSS_HOME*/server/[*name*]/deploy/transaction-jboss-beans.xml** file. This ties the transaction manager into the overall server configuration, overriding the transaction configuration file settings with values specific to the application server where appropriate. In particular, it uses the

Service Binding Manager to set port binding information, as well as overriding selected other properties. Configuration properties are read by the Transaction Service at server initialization, and the server must be restarted to incorporate any changes made to the configuration files.

Table 9.1. Most Critical Properties for JBoss Transaction Service

Property Name	Default Value	Description
transactionTimeout	300 seconds	the default time, in seconds, after which a transaction will time out and be rolled back by. Adjust this to suit your environment and workload. It may come as a surprise that transactions are processed asynchronously. This was a design decision, and needs to be accounted for by your code.
objectStoreDir		The directory where transaction data is logged. The transaction log is required to complete transactions in the case of system failure, and needs to be on reliable storage. Normally one file is generated per transaction, and each file is a few kilobytes in size. These are distributed over a directory tree for optimal performance. If a RAID controller is used, it should be configured for write through cache, in much the same manner as database storage devices. Writing of the transaction log is automatically skipped in the case of transactions that are rolling back or contain only a single resource.

Table 9.2. Additional Properties for JBoss Transaction Service

Property Name	Default Value	Description

Property Name	Default Value	Description
com.arjuna.common.util.logging.DebugLevel	**`0x00000000`**, which equates to no logging	determines the internal log threshold for the transaction manager codebase. It is independent of the overall server's **log4j** logging configuration, and acts to suppress extraneous log entries from being printed. When the default value is active, INFO and WARN messages are still printed, and this setting provides optimal performance. **`0xffffffff`** enables full debug logging. This setting results in large log files. Log messages that pass the internal **`DebugLevel`** check are passed to the server's logging system for further processing. In theory, full debugging may be left on and **log4j** can be used to turn logging on or off, but in reality this has a performance impact.
com.arjuna.ats.arjuna.coordinator.commitOnePhase	**YES**	Determines whether the transaction manager automatically applies the one-phase commit optimization to the transaction completion protocol, when only a single resource is registered with the transaction. Enabled by default to prevent writing transaction logs needlessly.
com.arjuna.ats.arjuna.objectstore.transactionSync	**ON**	Controls the flushing of transaction logs to disk during transaction termination. The default value results in a **`FileDescriptor.sync`** call for each committing transaction. This behavior is required to provide recovery and ACID properties. If these features are unimportant to the application in question, you can achieve better performance by disabling this property. This is discouraged, since it is usually better to write such applications in a way that avoids using transactions at all.

Property Name	Default Value	Description
com.arjuna.ats.arjuna.xa.nodeIdentifier com.arjuna.ats.jta.xaRecoveryNode		These properties determine the behavior of the transaction recovery system. They must be configured correctly to ensure that transactions are resolved correctly so that recovery can happen if the server crashes. Please refer to the Recovery chapter of the JBoss Transactions Administration Guide for more details.
com.arjuna.ats.arjuna.coordinator.enableStatistics	**NO**	Enables gathering of transaction statistics. The statistics can be viewed using methods on the **`TransactionManagerService`** bean or its corresponding JMX MBean. Disabled by default.

9.3. TRANSACTIONAL RESOURCES

The Transaction Service coordinates transaction state updates using **`XAResource`** implementations, which are provided by the various resource managers. Resource managers may include databases, message queues or third-party JCA resource adapters. The list of databases and JDBC drivers which have been certified on JBoss Enterprise Application Platform is located at http://www.jboss.com/products/platforms/application/supportedconfigurations/. Most standards-compliant JDBC drivers should function correctly, but you should perform extensive testing when using an uncertified configuration, since interpretations of the XA specifications different from one vendor to another.

Database connection pools are configured via the application server's Datasource files, which are files named like **`-ds.xml`**. Datasources which use the <xa-datasource> property automatically interact with the transaction manager. Connections obtained by looking up such datasource in JNDI and calling **`getConnection`** automatically participate in ongoing transactions. This is the preferred use case when transactional guarantees for data access are required.

If you are using a database which cannot support XA transactions, you can deploy a connection pool using <local-xa-datasource>. This type of datasource participates in the managed transaction using the Section 9.4, “Last Resource Commit Optimization (LRCO)” , providing more limited transactional guarantees. Connections obtained from a <no-tx-datasource> do not interact with the transaction manager, and any work done on such connections must be explicitly committed or rolled back by the application, using the JDBC API.

Many databases require additional configuration before they can function as XA resource managers. Vendor-specific information for configuring databases is presented in Appendix A, *Vendor-Specific Datasource Definitions*. Refer to your database administrator and the documentation which ships with your database for additional configuration directives. In addition, please consult the JBoss Transactions Administration Guide for information on setting up XA recovery properly.

JBoss Messaging provides an XA-aware driver and can participate in XA transactions. Please consult the JBoss Messaging User Guide for more details.

9.4. LAST RESOURCE COMMIT OPTIMIZATION (LRCO)

Although the XA transaction protocol is designed to provide ACID properties by using a two-phase commit protocol, model may not always be appropriate. Sometimes it is necessary to allow a non-XA-aware resource manager to participate in a transaction. This is often the case with data stores that do not support distributed transactions.

In this situation, you can use a technique known as *Last Resource Commit Optimization (LRCO)*. This is sometimes called the *Last Resource Gambit*. The one-phase-aware resource is processed last in the **`prepare`** phase of the transaction, at which time an attempt is made to commit it. If the attempt is successful, the transaction log is written and the remaining resources go through the phase-two commit. If the last resource fails to commit, the transaction is rolled back. Although this protocol allows most transactions to complete normally, some errors can cause an inconsistent transaction outcome. For this reason, use LRCO as a last resort. When a single <local-tx-datasource> is used in a transaction, the LRCO is automatically applied to it. In other situations, you can designate a last resource by using a special marker interface. Refer to the JBoss Transactions Programmer's Guide for more details.

Using more than a single one-phase resource in the same transaction is not transactionally safe, and is not recommended. JBoss Transaction Service sees an attempt to enlist a second such resource as an error and terminates the transaction. This type of error is most often found when migrating from a legacy version of JBoss Application Server. Whenever possible the <local-tx-datasource> should be converted to an <xa-datasource> to resolve the difficulty.

9.5. TRANSACTION TIMEOUT HANDLING

In order to prevent indefinite locking of resources, the transaction manager aborts in-flight transactions that have not completed after a specified interval, using a set of background processes coordinated by the **`TransactionReaper`**. The reaper rolls back transactions without interrupting any threads that may be operating within their scope. This prevents instability that results from interrupting threads executing arbitrary code. Furthermore, it allows for timely abort of transactions where the business logic thread may be executing non-interruptable operations such as network I/O operations. This approach may, cause unexpected behavior in code that is not designed to handle multithreaded transactions. Warning or error messages may be printed from transaction-aware components as a result of the unexpected change in transaction status. The transaction outcome should usually be unaffected. Any problems can be minimized by tuning the transaction timeout values. See Chapter 13, *Datasource Configuration* for more information.

9.6. RECOVERY CONFIGURATION

To ensure that your configuration is robust, it is important to configure JBoss Transaction Service properly for failure and recovery. This is covered in detail in the *JBoss Transactions Administration Guide*, in the "Resource Recovery in JBoss Transaction Service" chapter.

9.7. TRANSACTION SERVICE FAQ

This section presents some of the most common configuration issues with JBoss Transaction Service.

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

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

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

A log statement must pass both filters to be printed. A typical mistake is enabling only one or the other of the logging systems. See Table 9.2, "Additional Properties for JBoss Transaction Service" for more information.

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

A: You are probably using a <local-xa-datasource> and trying to use more than one one-phase aware participant. This is a configuration to be avoided. See Section 9.4, "Last Resource Commit Optimization (LRCO)" for more information. If you have further concerns, please contact Global Support Services.

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

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

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

A: Transactions which exceed their timeout may be rolled back. This is done by a background thread, which can confuse some application code that may be expecting an interrupt. Refer to Section 9.5, "Transaction Timeout Handling" for more information.

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

9.8. USING THE JTS MODULE

If you need transaction propagation betweenb usiness logic in different servers, you can use the JTS API. Although you can use it directly, it is typical to access it via the standard JTA classes. It is a drop-in replacement for the default local-only JTA implementation. The necessary classes are already in place, and you only need to modify the **`jbossts-properties.xml`** file to move between the JTA and JTS modules.

A sample **`jbossts-properties.xml`** file is located in the ***`$JBOSS_HOME`*`/docs/examples/transactions/`** directory. Consult the **`README.txt`** file in the same directory for more information about changes that need to be made to other files, including the **`transactions-jboss-beans.xml`** file. An ANT script is provided to perform all of the steps automatically, but it is recommended to consult the **`README.txt`** carefully before running the script, as well as backing up your existing configuration.

The JTS requires the server configuration to also contain the CORBA ORB service. The "all" configuration referenced in the examples is a good starting point. The choice of JTS or JTA impacts the entire server, and JTS does require additional resources. Therefore, only use it when it is needed.

At application start-up, a server that is configured to use JTA outputs log files like this one:

```
      INFO [TransactionManagerService] JBossTS Transaction Service (JTA
version - ...)
```

If JTS is enabled, the message looks like this one:

```
      INFO [TransactionManagerService] JBossTS Transaction Service (JTS
version - ...)
```

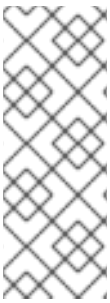
9.9. USING THE XTS MODULE

XTS, which is the Web Services component of JBoss Transaction Service, can be installed to provide WS-AT and WS-BA support for web services hosted on the Enterprise Application Platform. The module is packaged as a *Service Archive (.sar)* located in ***$JBOSS_HOME*/docs/examples/transactions/**.

Procedure 9.1. Installing the XTS Module

1. Create a subdirectory in the ***$JBOSS_HOME*/server/[*name*]/deploy/** directory, called **jbossxts.sar/**.
2. Unpack the .sar, which is a ZIP archive, into this new directory.
3. Restart JBoss Enterprise Application Platform for the module to be active.

The server must use either the JTA or JTS module, as well as JBossWS Native.



NOTE

XTS is not currently expected to work with other JBossWS backends such as CXF. The default XTS configuration is suitable for most deployments. It automatically detects information about the network interfaces and port bindings from the EAP configuration. manual configuration changes are only necessary for deployments whose applications need to use a transaction coordinator on a separate host. Consult the JBoss Web Service Transactions Programmer's Guide for more information.

Developers can link against the **jbossxts-api.jar** file included in the XTS Service Archive, but should avoid packaging it with their applications, to avoid classloading problems. All other JAR files contain internal implementation classes and should not be used directly.

Consult ***$JBOSS_HOME*/docs/examples/transactions/README.txt** for emore configuration information. The JBoss Web Services Transactions User Guide contains information about using XTS in your applications.

9.10. TRANSACTION MANAGEMENT CONSOLE

The Transaction Management Console is a simle GUI tool that is included in ***$JBOSS_HOME*/docs/example/transactions/**. It is provided as an unsupported, experimental prototype. Consult the **README.txt** file for its capabilities and information about its use.

9.11. EXPERIMENTAL COMPONENTS

In addition to the supported components of JBoss Transaction Service which are included in JBoss Enterprise Application Platform, there is ongoing feature work that may eventually find its way into future releases of the product. In the meantime, these prototype components are available via from the http://jboss.org Community website.

WARNING

There is no guarantee these components will work correctly and they are not covered under the Enterprise Application Platform support agreement. However, some of the advanced functionality available may useful for projects in the early stages of development. Users downloading these prototypes must be aware of the limitations concerning module compatibility, in accordance with the Section 9.12, "Source Code and Upgrading".

txbridge

Sometimes you may need the ability to invoke traditional transaction components, such as EJBs, within the scope of a Web Services transaction. Conversely, some traditional transactional applications may need to invoke transactional web services. The Transaction Bridge (txbridge) provides mechanisms for linking these two types of transactional services together.

BA Framework

The XTS API operates at a very low level, requiring the developer to undertake much of the transaction infrastructure work involved in WS-BA. The BA Framework provides high-level annotations that enable JBoss Transaction Service to handle this infrastructure. The developer can then focus more on business logic instead.

9.12. SOURCE CODE AND UPGRADING

Most problems relating to transactions can be diagnosed by Global Support Services, after you provide debug logging information from the server.

However, you can debug or review the source code yourself, using your own tools. You can download the source code using the Subversion repository at http://anonsvn.jboss.org/repos/labs/labs/jbosstm/. Enterprise Application Platform outputs the version of the Transaction Service at start-up, using a string similar to this one:

```
      INFO [TransactionManagerService] JBossTS Transaction Service (JTA
version - tag:JBOSSTS_4_6_1_GA_CP02) - JBoss Inc.
```

The **tag** element corresponds to a tree under /tags/ in the Subversion repository. Note that the version refers to the version of the JBoss Transaction Service component used in the Enterprise Application Platform, not the version of EAP itself. If you build Enterprise Application Platform from source, you can also find the version by searching for the string **version.jboss.jbossts** in the **component-matix/pom.xml** file.

WARNING

Installing any version of JBossTS other than those provided with the Enterprise Application Platform you are using is not supported. While some JBoss Transaction Service components are packaged separately, it is unsupported to use different versions than the ones suppolied with Enterprise Application Platform.

CHAPTER 10. 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.

Out of the box, Remoting supplies multiple transports (bisocket, http, rmi, socket, servlet, and their ssl enabled counterparts), standard and compressing data marshallers, and a configurable facility for switching between standard jdk serialization and JBoss Serializabion. It is also capable of remote classloading, has extensive facilities for connection failure notification, performs call by reference optimization for client/server invocations collocated in a single JVM, and implements multihomed servers.

In the Enterprise Application Platform, Remoting supplies the transport layer for the EJB2, EJB3, and Messaging subsystems. In each case, the configuration of Remoting is largely predetermined and fixed, but there are times when it is useful to know how to alter a Remoting configuration.

10.1. BACKGROUND

A Remoting server consists of a Connector, which wraps and configures a transport specific server invoker. A connector is represented by an InvokerLocator string, such as

```
socket://bluemonkeydiamond.com:8888/?timeout=10000&serialization=jboss
```

which indicates that a server using the socket transport is accessible at port 8888 of host bluemonkeydiamond.com, and that the server is configured to use a socket timeout of 10000 and to use JBoss Serialization. A Remoting client can use an InvokerLocator to connect to a given server.

In the Enterprise Application Platform, Remoting servers and clients are created far below the surface and are accessible only through configuration files. Moreover, when a proxy for a SLSB, for example, is downloaded from the JNDI directory, it comes with a copy of the InvokerLocator, so that it knows how to contact the appropriate Remoting server. **The important fact to note is that, since the server and its clients share the InvokerLocator, the parameters in the InvokerLocator serve to configure both clients and servers.**

10.2. JBOSS REMOTING CONFIGURATION

There are two kinds of XML files that can be used to create and configure a Remoting Connector. A file with a name of the form *-service.xml can be used to define a Connector as an MBean, and a file of the form *-jboss-beans.xml can be used to define a Connector as a POJO.

10.2.1. MBeans

In the JBoss Messaging JMS subsystem, a Remoting server is configured in the file remoting-bisocket-service.xml, which, in abbreviated form, looks like

```
<mbean code="org.jboss.remoting.transport.Connector"
       name="jboss.messaging:service=Connector,transport=bisocket"
       display-name="Bisocket Transport Connector">
  <attribute name="Configuration">
    <config>
```

```
          <invoker transport="bisocket">
            <attribute name="marshaller"
isParam="true">org.jboss.jms.wireformat.JMSWireFormat</attribute>
            <attribute name="unmarshaller"
isParam="true">org.jboss.jms.wireformat.JMSWireFormat</attribute>
            <attribute name="serverBindAddress">${jboss.bind.address}
</attribute>
            <attribute name="serverBindPort">4457</attribute>
            <attribute name="callbackTimeout">10000</attribute>
                 ...
          </invoker>
             ...
        </config>
      </attribute>
    </mbean>
```

This configuration file tells us several facts, including

- This server uses the bisocket transport;
- it runs on port 4457 of host ${jboss.bind.address}; and
- JBoss Messaging uses its own marshalling algorithm.

The InvokerLocator is derived from this file. **The important fact to note is that the attribute "isParam" determines if a parameter is to be included in the InvokerLocator.** If "isParam" is omitted or set to false, the parameter will apply only to the server. In this case, the parameter will not be transmitted to the client. The InvokerLocator for a Remoting server with a ${jboss.bind.address} of bluemonkeydiamond.com would be:

```
      bisocket://bluemonkeydiamond.com:4457/?marshaller=
      org.jboss.jms.wireformat.JMSWireFormat&
      unmarshaller=org.jboss.jms.wireformat.JMSWireFormat
```

Note that the parameter "callbackTimeout" is not included in the InvokerLocator.

10.2.2. POJOs

The same Connector could be configured by way of the **`org.jboss.remoting.ServerConfiguration`** POJO:

```
     <bean name="JBMConnector"
class="org.jboss.remoting.transport.Connector">
       <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
        (name="jboss.messaging:service=Connector,transport=bisocket",

exposedInterface=org.jboss.remoting.transport.ConnectorMBean.class,
        registerDirectly=true)</annotation>
       <property name="serverConfiguration"><inject
bean="JBMConfiguration"/></property>
     </bean>

     <!-- Remoting server configuration -->
     <bean name="JBMConfiguration"
class="org.jboss.remoting.ServerConfiguration">
```

```
        <constructor>
          <parameter>bisocket</parameter>
        </constructor>

         <!-- Parameters visible to both client and server -->
        <property name="invokerLocatorParameters">
          <map keyClass="java.lang.String" valueClass="java.lang.String">
            <entry>
              <key>serverBindAddress</key>
              <value>
                <value-factory bean="ServiceBindingManager"
method="getStringBinding">
                  <parameter>JBMConnector</parameter>
                  <parameter>${host}</parameter>
                </value-factory>
              </value>
            </entry>
            <entry>
              <key>serverBindPort</key>
              <value>
                <value-factory bean="ServiceBindingManager"
method="getStringBinding">
                  <parameter>JBMConnector</parameter>
                  <parameter>${port}</parameter>
                </value-factory>
              </value>
            </entry>
               ...
            <entry><key>marshaller</key>
<value>org.jboss.jms.wireformat.JMSWireFormat</value></entry>
            <entry><key>unmarshaller</key>
<value>org.jboss.jms.wireformat.JMSWireFormat</value></entry>
          </map
        </property>

        <!-- Parameters visible only to server -->
        <property name="serverParameters">
          <map keyClass="java.lang.String" valueClass="java.lang.String">
            <entry><key>callbackTimeout</key> <value>10000</value></entry>
          </map>
        </property>

         ...
      </bean>
```

In this version, the configuration information is expressed in the JBMConfiguration **`ServerConfiguration`** POJO, which is then injected into the JBMConnector **`org.jboss.remoting.transport.Connector`** POJO. The syntax is that of the Microcontainer, which is beyond the scope of this chapter. See Chapter 5, *Microcontainer* for details. One variation from the MBean version is the use of the ServiceBindingManager, which is also beyond the scope of this chapter. Note that the @org.jboss.aop.microcontainer.aspects.jmx.JMX annotation causes the JBMConnector to be visible as an MBean named "jboss.messaging:service=Connector,transport=bisocket".

10.3. MULTIHOMED SERVERS

Remoting can create servers bound to multiple interfaces. One application of this facility would be binding a server to one interface that faces the internet and another that faces a LAN. For example, the preceding POJO example can be modified by (1) adding POJOs

```
   <bean name="homes1" class="java.lang.StringBuffer">
      <constructor>
         <parameter class="java.lang.String">
            <value-factory bean="ServiceBindingManager"
method="getStringBinding">
               <parameter>JBMConnector:bindingHome1</parameter>
               <parameter>${host}:${port}</parameter>
            </value-factory>
         </parameter>
      </constructor>
   </bean>

   <bean name="homes2" class="java.lang.StringBuffer">
      <constructor factoryMethod="append">
         <factory bean="homes1"/>
         <parameter>
            <value-factory bean="ServiceBindingManager"
method="getStringBinding">
               <parameter>JBMConnector:bindingHome2</parameter>
               <parameter>!${host}:${port}</parameter>
            </value-factory>
         </parameter>
      </constructor>
   </bean>
```

which results in a StringBuffer with a value something like (according to the ServiceBindingManager configuration values for JBMConnector:bindingHome1 and JBMConnector:bindingHome2) "external.acme.com:5555!internal.acme.com:4444", and (2) replacing the "serverBindAddress" and "serverBindPort" parameters with

```
    <entry>
      <key>homes</key>
      <value><value-factory bean="homes2" method="toString"/></value>
    </entry>
```

which transforms the StringBuffer into the String "external.acme.com:5555!internal.acme.com:4444" and injects it into the JBMConnector. The resulting InvokerLocator will look like

```
    bisocket://multihome/?homes=external.acme.com:5555!internal.acme.com:
    4444&marshaller=org.jboss.jms.wireformat.JMSWireFormat&
    unmarshaller=org.jboss.jms.wireformat.JMSWireFormat
```

10.4. ADDRESS TRANSLATION

Sometimes a server must be accessed through an address translating firewall, and a Remoting server can be configured with both a binding address/port and an address/port to be used by a client. Two more parameters are used: "clientConnectAddress" and "clientConnectPort". The "serverBindAddress" and "serverBindPort" values are used to create the server, and the values of "clientConnectAddress" and "clientConnectPort" are used in the InvokerLocator, which tells the client

where the server is. There is also an analogous "connecthomes" parameter for multihome servers. In this case, "homes" is used to configure the server, and "connecthomes" tells the client where the server is.

10.5. WHERE ARE THEY NOW?

The actual Remoting configuration files for the supported subsystems are as follows:

EJB2: ${JBOSS_HOME}/server/${CONFIG}/deploy/remoting-jboss-beans.xml

EJB3: ${JBOSS_HOME}/server/${CONFIG}/deploy/ejb3-connectors-jboss-beans.xml

JBM: ${JBOSS_HOME}/server/${CONFIG}/deploy/messaging/remoting-bisocket-service.xml

10.6. FURTHER INFORMATION.

Additional details may be found in the Remoting Guide at http://www.jboss.org/jbossremoting/docs/guide/2.5/html/index.html.

CHAPTER 11. JBOSS MESSAGING

The most current information about using JBoss Messaging is always available from the relevant *JBoss Messaging User Guide* at http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/.

CHAPTER 12. USE ALTERNATIVE DATABASES WITH JBOSS ENTERPRISE APPLICATION PLATFORM

12.1. 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 Enterprise Application Platform. For a complete list of certified databases, refer to http://www.jboss.com/products/platforms/application/supportedconfigurations/.

Please note that in this chapter, we explain how to use alternative databases to support all services in JBoss Enterprise 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.

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

12.2. INSTALL JDBC DRIVERS

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

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

JBDC Driver Download Locations

MySQL

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

PostgreSQL

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

Oracle

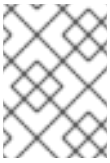
Download from http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html.

IBM

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

Sybase

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



NOTE

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

Microsoft

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

12.2.1. Special Notes on Sybase

Some of the services in JBoss uses null values for the default tables that are created. Sybase Adaptive Server should be configured to allow nulls by default.

```
sp_dboption db_name, "allow nulls by default", true
```

Refer to the Sybase manuals for more options.

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

12.2.1.1. Enable JAVA services

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

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

Refer to the sybase manuals for more information.

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

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

12.2.1.2. CMP Configuration

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

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

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

12.2.1.3. Installing Java Classes

1. You have to be a super-user with required privileges to install Java classes.
2. The JAR file you are trying to install should be created without compression.
3. Java classes that you install and use in the server must be compiled with JDK 1.2.2. If you compile a class with a later JDK, you will be able to install it in the server using the **installjava** utility, but you will get a java.lang.ClassFormatError exception when you attempt to use the class. This is because Sybase Adaptive Server uses an older JVM internally, and requires the Java classes to be compiled with the same.

12.2.2. Configuring JDBC DataSources

Datasources correspond to the simplified JCA Datasource configuration specifications.

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

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

See Chapter 13, *Datasource Configuration* for information on configuring datasources. As a minimum, you will need to change the **connection-url**, **user-name**, and **password** to correspond to your database of choice.

12.3. COMMON DATABASE-RELATED TASKS

12.3.1. Security and Pooling

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

NOTE

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

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

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

12.3.3. Support Foreign Keys in CMP Services

Next, we need to go change the **$JBOSS_HOME/server/$PROFILE/conf/standardjbosscmp-jdbc.xml** file so that the **fk-constraint** property is **true**. That is needed for all external databases we support on the JBoss Enterprise Application Platform. This file configures the database connection settings for the EJB2 CMP beans deployed in the JBoss Enterprise Application Platform.

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

12.3.4. Specify Database Dialect for Java Persistence API

The Java Persistence API (JPA) entity manager can save EJB3 entity beans to any backend database. Hibernate provides the JPA implementation in JBoss Enterprise 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 **$JBOSS_HOME/server/$PROFILE/deployers/ejb3.deployer/META-INF/jpa-deployers-jboss-beans.xml** file. To configure this file you need to uncomment the set of tags related to the map entry **hibernate.dialect** and change the values to the following based on the database you setup.

- Oracle 10g: **org.hibernate.dialect.Oracle10gDialect**
- Oracle 11g: **org.hibernate.dialect.Oracle10gDialect**
- Microsoft SQL Server 2005: **org.hibernate.dialect.SQLServerDialect**
- Microsoft SQL Server 2008: **org.hibernate.dialect.SQLServerDialect**
- 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.SybaseASE15Dialect**

12.3.5. 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.

12.3.5.1. The Easy Way

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

To change the JNDI name, just open the ***-ds.xml** file for your external database, and change the value of the **jndi-name** property to **DefaultDS**. For instance, in **mysql-ds.xml**, you would change **MySqlDS** to **DefaultDS** and so on. You will need to remove the **$JBOSS_HOME/server/$PROFILE/deploy/hsqldb-ds.xml** file after you are done to avoid duplicated **DefaultDS** definition.

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

```
<mbean

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


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

12.3.5.2. The More Flexible Way

Changing the external datasource to **DefaultDS** is convenient. But if you have applications that assume the **DefaultDS** always points to the factory-default HSQL DB, that approach could break your application. Also, changing **DefaultDS** destination forces all JBoss services to use the external database. What if you want to use the external database only on some services?

A safer and more flexible way to hook up JBoss Enterprise 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 (for example, the **MySqlDS** in **mysql-ds.xml**, etc.). Below is a complete list of files that contain **DefaultDS**. You can update them all to use the external database on all JBoss services or update some of them to use different combination of DataSources for different services.

- **$JBOSS_HOME/server/$PROFILE/conf/login-config.xml**: This file is used in Java EE container managed security services.
- **$JBOSS_HOME/server/$PROFILE/conf/standardjbosscmp-jdbc.xml**: This file configures the CMP beans in the EJB container.
- **$JBOSS_HOME/server/$PROFILE/deploy/ejb2-timer-service.xml**: This file configures the EJB timer services.

- **$JBOSS_HOME/server/$PROFILE/deploy/juddi-service.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- **$JBOSS_HOME/server/$PROFILE/deploy/juddi-service.sar/juddi.war/WEB-INF/jboss-web.xml**: This file configures the UUDI service.
- **$JBOSS_HOME/server/$PROFILE/deploy/juddi-service.sar/juddi.war/WEB-INF/juddi.properties**: This file configures the UUDI service.
- **$JBOSS_HOME/server/$PROFILE/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml**: This file configures the UUDI service.
- **$JBOSS_HOME/server/$PROFILE/deploy/messaging/messaging-jboss-beans.xml** and **$JBOSS_HOME/server/$PROFILE/deploy/messaging/persistence-service.xml**: Those files configure the JMS persistence service as we discussed earlier.

12.3.6. 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.

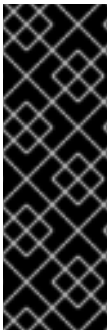
The Oracle database creates tables of the form **schemaname.tablename**. The **TIMERS** and **HILOSEQUENCES** tables needed by JBoss Enterprise Application Platform would not be created on a schema if the table already existed on a different schema. To work around this issue, you need to edit the ***$JBOSS_HOME*/server/*$PROFILE*/deploy/ejb2-timer-service.xml** file to change the table name from **TIMERS** to something like **schemaname2.tablename**.

```
          <mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"

name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
          <!-- DataSourceBinding ObjectName -->
          <depends optional-attribute-name="DataSource">
          jboss.jca:service=DataSourceBinding,name=DefaultDS
          </depends>
          <!-- The plugin that handles database persistence -->
          <attribute name="DatabasePersistencePlugin">
          org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin
          </attribute>
          <!-- The timers table name -->
          <attribute name="TimersTable">TIMERS</attribute>
          </mbean>
```

Similarly, you need to change the ***$JBOSS_HOME*/server/*$PROFILE*/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml** file to change the table name from **HILOSEQUENCES** to something like **schemaname2.tablename** as well.

```
 <!-- HiLoKeyGeneratorFactory --> <mbean

code="org.jboss.ejb.plugins.keygenerator.hilo.HiLoKeyGeneratorFactory"
          name="jboss:service=KeyGeneratorFactory,type=HiLo">

          <depends>jboss:service=TransactionManager</depends>
```

```
          <!-- Attributes common to HiLo factory instances -->

          <!-- DataSource JNDI name -->
          <depends optional-attribute-
name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</depe
nds>

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



IMPORTANT

Oracle JDBC driver version 11.1.0.7.0 causes the JBoss Messaging Test Suite to fail with a **SQLException** ("Bigger type length than Maximum") on Oracle 11g R1.

This is caused by a regression in Oracle JDBC driver 11.1.0.7.0.

We recommend Oracle JDBC driver version 11.2.0.1.0 for use with Oracle 11g R1, Oracle 11g R2, Oracle RAC 11g R1 and Oracle RAC 11g R2.

CHAPTER 13. DATASOURCE CONFIGURATION

WARNING

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

Known issues with the Hypersonic Database include:

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

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

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

13.1. TYPES OF DATASOURCES

Datasource Definitions

<no-tx-datasource>

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

<local-tx-datasource>

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

<xa-datasource>

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

13.2. DATASOURCE PARAMETERS

Common Datasource Parameters

<mbean>

A standard JBoss MBean deployment.

<depends>

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

<jndi-name>

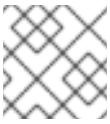
The JNDI name under which the Datasource should be bound.

<use-java-context>

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

<user-name>

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

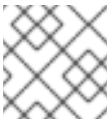


NOTE

Not used when security is configured.

<password>

The password used to create the connection to the datasource.



NOTE

Not used when security is configured.

<transaction-isolation>

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

Possible values for <transaction-isolation>

- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE
- TRANSACTION_NONE

<new-connection-sql>

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

<check-valid-connection-sql>

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

<valid-connection-checker-class-name>

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

<exception-sorter-class-name>

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

<track-statements>

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

<prepared-statement-cach-size>

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

<share-prepared-statements>

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

Example 13.1. Using <share-prepared-statements>

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

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

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

<set-tx-query-timeout>

Whether to enable query timeout based on the length of time remaining until the transaction times out. Defaults to **FALSE**.

<query-timeout>

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

<metadata>><type-mapping>

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

<validate-on-match>

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

<prefill>

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

<background-validation>

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

<background-validation-millis>

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

<idle-timeout-minutes>

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

<track-connection-by-tx>

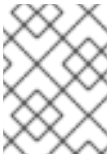
Whether the connection should be locked to the transaction, instead of returning it to the pool at the end of the transaction. In previous releases, this was **`true`** for local connection factories and **`false`** for XA connection factories. The default is now **`true`** for both local and XA connection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

<interleaving>

Enables interleaving for XA connection factories.

<background-validation-minutes>

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



NOTE

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

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

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

<stale-connection-checker-class-name>

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

<max-pool-size>

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

<min-pool-size>

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

<blocking-timeout-millis>

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

<use-fast-fail>

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

Parameters for `javax.sql.XADataSource` Usage

<connection-url>

The JDBC driver connection URL string

<driver-class>

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

<connection-property>

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

Example 13.2. Example <connection-property>

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

-

Parameters for `javax.sql.XADataSource` Usage

<xa-datasource-class>

The class implementing the **`XADataSource`**

<xa-datasource-property>

Properties used to configure the **`XADataSource`**.

Example 13.3. Example <xa-datasource-property> Declarations

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

<xa-resource-timeout>

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

<isSameRM-override-value>

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

<no-tx-separate-pools>

Pool transactional and non-transactinal connections separately

WARNING

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

Used to fix problems with Oracle.

Security Parameters

`<application-managed-security>`

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

<security-domain>

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

<security-domain-and-application>

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

13.3. DATASOURCE EXAMPLES

For database-specific examples, see Appendix A, *Vendor-Specific Datasource Definitions*.

13.3.1. Generic Datasource Example

Example 13.4. Generic Datasource Example

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

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

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

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

```
queries -->

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

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

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

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

  </local-tx-datasource>

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


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

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



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

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

```
connection-sql>
    -->


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

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

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

  </xa-datasource>

</datasources>
```

13.3.2. Configuring a DataSource for Remote Usage

JBoss EAP supports accessing a DataSource from a remote client. See Example 13.5, “Configuring a Datasource for Remote Usage” for the change that gives the client the ability to look up the DataSource from JNDI, which is to specify **`use-java-context=false`**.

Example 13.5. Configuring a Datasource for Remote Usage

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

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

NOTE

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

13.3.3. Configuring a Datasource to Use Login Modules

Procedure 13.1. Configuring a Datasource to Use Login Modules

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

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

2. **Add an application policy to the `login-config.xml` file.**
   The authentication section needs to include the configuration for your login-module. For example, to encrypt the database password, use the **`SecureIdentityLoginModule`** login module.

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

3. If you plan to fetch the data source connection from a web application, authentication must be enabled for the web application, so that the **`Subject`** is populated.

4. If users need the ability to connect anonymously, add an additional login module to the application-policy, to populate the security credentials.

5. Add the **`UsersRolesLoginModule`** module to the beginning of the chain. The **`usersProperties`** and **`rolesProperties`** parameters can be directed to dummy files.

```
<login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
   <module-option name="unauthenticatedIdentity">nobody</module-
option>
   <module-option
name="usersProperties">props/users.properties</module-option>
   <module-option
name="rolesProperties">props/roles.properties</module-option>
</login-module>
```

CHAPTER 14. POOLING

14.1. STRATEGY

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

xml	mbean	Internal Name	Description
	ByNothing	OnePool	A single pool of equivalent connections
<application-managed-security/>	ByApplication	PoolByCRI	Use the connection properties from allocateConnection()
<security-domain/>	ByContainer	PoolBySubject	A pool per Subject, e.g. preconfigured or EJB/Web login subjects
<security-domain-and-applicaton/>	ByContainerAndApplicaton	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
)
```

14.2. 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

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.

14.3. 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/>.

14.4. 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

You may want to use the <allocation-retry/> and <allocation-retry-wait-millis/> elements to have the pool retry to obtain a connection before throwing the exception.

14.5. 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.

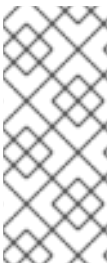
14.6. IDLE CONNECTIONS

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.



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.

14.7. 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.

14.7.1. 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/>
```

14.7.2. 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</excepti
on-sorter-class-name>
```

For **FATAL** errors, the connection will be closed.

14.7.3. Changing/Closing/Flushing the pool

- change or flush() the pool
- closing/undeploying the pool will do a flush first

14.7.4. Other pooling

Thirdparty Pools - only if you know what you are doing

CHAPTER 15. FREQUENTLY ASKED QUESTIONS

15.1. 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.

PART III. CLUSTERING GUIDE

CHAPTER 16. 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*.
- Keeping deployed content in sync on all nodes in the cluster via the `Farm` service.

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.

16.1. 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.

16.1.1. Initial Preparation

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

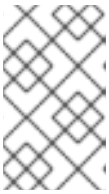
- **Install JBoss Enterprise Application Platform on all your servers.** In its simplest form, this is just a matter of unzipping the JBoss download onto the filesystem on each server.

  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 for a good discussion on multicast address assignment. We'll cover how to use the address you pick in the next section.

See Section 25.6.2, "Isolating JGroups Channels" for more on isolating clusters.

16.1.2. 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**. 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** earlier in the previous section.

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

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
    -b 192.168.0.101 -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/bindingservice.beans/META-INF/bindings-jboss-beans.xml** file for the full ServiceBindingManager configuration.

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

NOTE

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

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

16.1.3. 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 22, *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 17.2.2, "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 22.1, "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 22.2, "Configuring HTTP session state replication" for more advanced configuration options.

16.1.4. 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 20, *Clustered Session EJBs* for more advanced configuration options.

16.1.5. 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 21, *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.

CHAPTER 17. CLUSTERING CONCEPTS

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

17.1. 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.

On the same Enterprise Application Platform instance, different services can create their own **`Channel`**. In a standard startup of the Enterprise Application Platform 5 *all* configuration, two different services create a total of four different channels – JBoss Messaging creates two and a core general purpose clustering service known as HAPartition creates two more. If you deploy clustered web applications, clustered EJB3 SFSBs or a clustered JPA/Hibernate entity cache, additional channels will be created. The channels the Enterprise Application Platform connects can be divided into three broad categories: a general purpose channel used by the HAPartition service, channels created by JBoss Cache for special purpose caching and cluster wide state replication, and two channels used by JBoss Messaging.

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 multiple channels, and hence multiple two node clusters.

On the same network, you may have different sets of servers whose services wish to cluster. Figure 17.1, "Clusters and server nodes" shows an example network of JBoss server instances divided into three sets, with the third set only having one node. This sort of topology can be set up simply by configuring the Enterprise 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 channel configurations and names match while they differ from any other channels on the same network. The Enterprise Application Platform tries to make this is easy as possible, such that servers that are meant to cluster only need to have the same values passed on the command line to the **`-g`** (partition name) and **`-u`** (multicast address) startup switches. For each set of servers, different values should be chosen. The sections on "JGroups Configuration" and "Isolating JGroups Channels" cover in detail how to configure the Enterprise Application Platform such that desired peers find each other and unwanted peers do not.

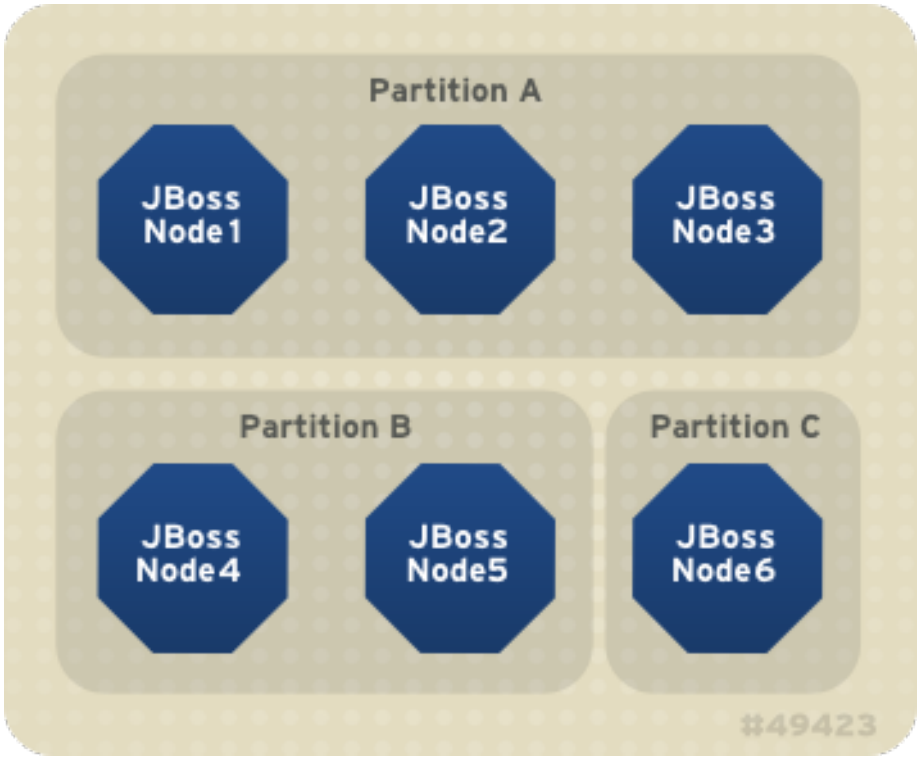


Figure 17.1. Clusters and server nodes

17.2. SERVICE ARCHITECTURES

The clustering topography defined by the JGroups configuration on each node is of great importance to system administrators. But for most application developers, the greater concern is probably the cluster architecture from a client application's point of view. Two basic clustering architectures are used with JBoss Enterprise 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.

17.2.1. Client-side interceptor architecture

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

The proxy's clustering logic maintains up-to-date knowledge about the cluster. For instance, it knows the IP addresses of all available server nodes, the algorithm to distribute load across nodes (see next section), and how to failover the request if the target node 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 17.2, “The client-side interceptor (proxy) architecture for clustering”.

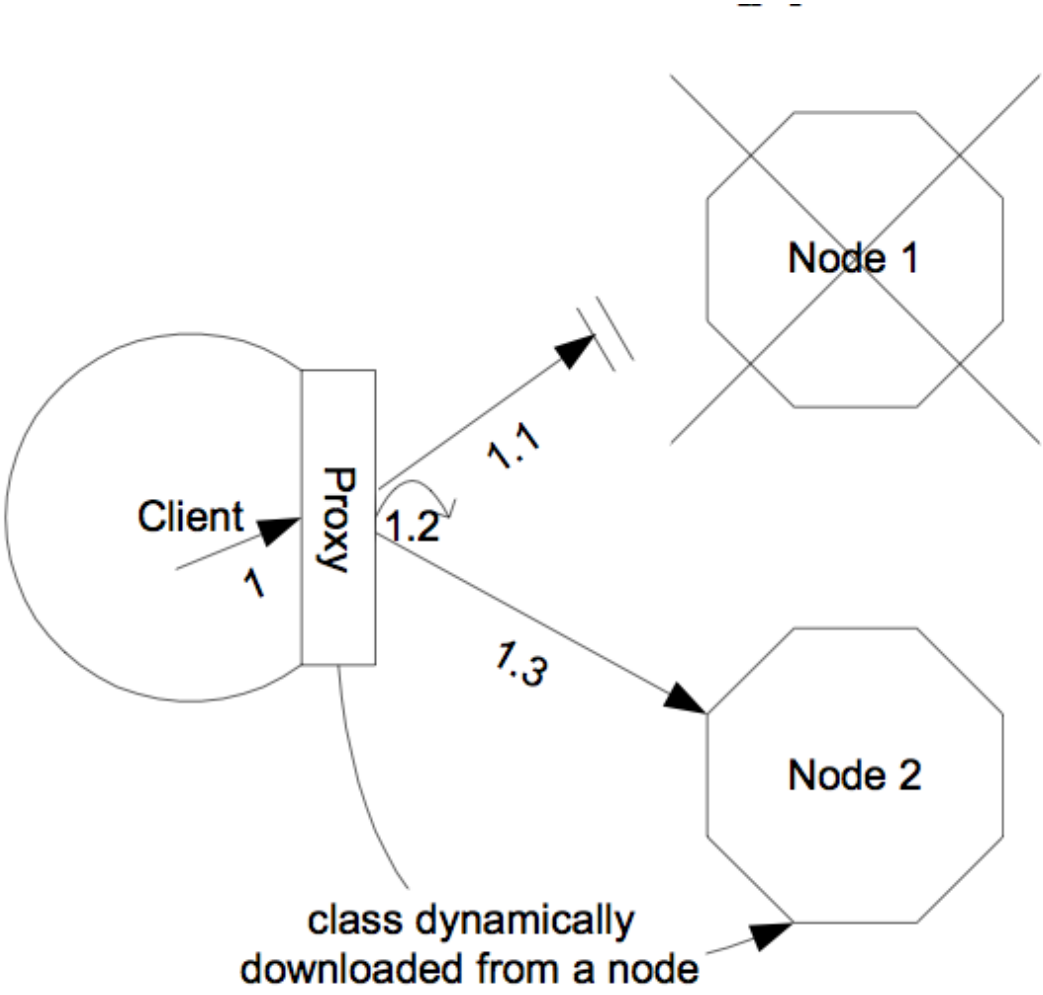


Figure 17.2. The client-side interceptor (proxy) architecture for clustering

17.2.2. External Load Balancer Architecture

The HTTP-based JBoss services do not require the client to download anything. The client (for example, a web browser) sends in requests and receives responses directly over the wire using the HTTP protocol). In this case, an external load balancer is required to process all requests and dispatch them to server nodes in the cluster. The client only needs to know how to contact the load balancer; it has no knowledge of the JBoss Enterprise 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 17.3, “The external load balancer architecture for clustering” .

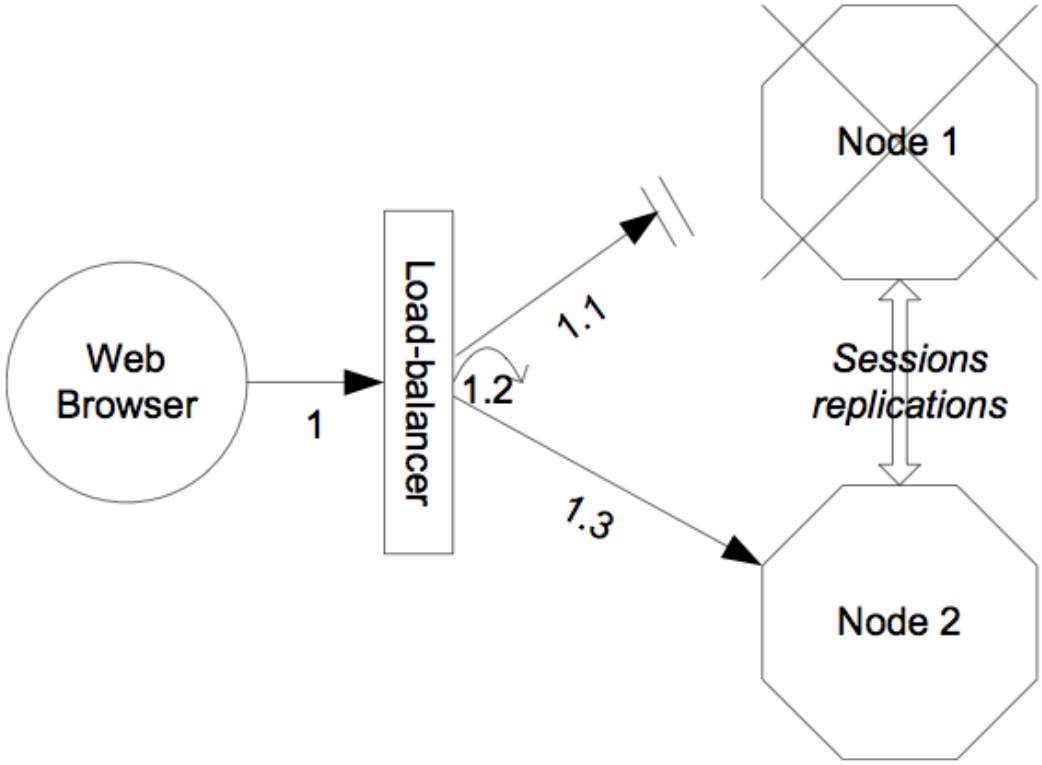


Figure 17.3. The external load balancer architecture for clustering

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.

17.3. LOAD BALANCING POLICIES

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

17.3.1. Client-side interceptor architecture

In JBoss Enterprise Application Platform 5, the following load balancing options are available when the client-side interceptor architecture is used. The client-side stub maintains a list of all nodes providing the target service; the job of the load balance policy is to pick a node from this list for each request. Each policy has two implementation classes, one meant for use by legacy services like EJB2 that use the legacy detached invoker architecture, and the other meant for services like EJB3 that use AOP-based invocations.

- Round-Robin: each call is dispatched to a new node, proceeding sequentially through the list of nodes. The first target node is randomly selected from the list. Implemented by **`org.jboss.ha.framework.interfaces.RoundRobin`** (legacy) and **`org.jboss.ha.client.loadbalance.RoundRobin`** (EJB3).

- Random-Robin: for each call the target node is randomly selected from the list. Implemented by **`org.jboss.ha.framework.interfaces.RandomRobin`** (legacy) and **`org.jboss.ha.client.loadbalance.RandomRobin`** (EJB3).

- First Available: one of the available target nodes is elected as the main target and is thereafter

used for every call; this elected member is randomly chosen from the list of members in the cluster. When the list of target nodes changes (because a node starts or dies), the policy will choose a new target node unless the currently elected node is still available. Each client-side proxy elects its own target node independently of the other proxies, so if a particular client downloads two proxies for the same target service (for example, an EJB), each proxy will independently pick its target. This is an example of a policy that provides “session affinity” or “sticky sessions”, since the target node does not change once established. Implemented by **`org.jboss.ha.framework.interfaces.FirstAvailable`** (legacy) and **`org.jboss.ha.client.loadbalance.aop.FirstAvailable`** (EJB3).

- First Available Identical All Proxies: has the same behavior as the "First Available" policy but the elected target node is shared by all proxies in the same client-side VM that are associated with the same target service. So if a particular client downloads two proxies for the same target service (e.g. an EJB), each proxy will use the same target. Implemented by **`org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies`** (legacy) and **`org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies`** (EJB3).

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.

17.3.2. External load balancer architecture

New in JBoss Enterprise Application Platform 5 are a set of "TransactionSticky" load balance policies. These extend the standard policies above to add behavior such that all invocations that occur within the scope of a transaction are routed to the same node (if that node still exists). These are based on the legacy detached invoker architecture, so they are not available for AOP-based services like EJB3.

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

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

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

- Transaction-Sticky First Available Identical All Proxies: Transaction-sticky variant of First Available Identical All Proxies. Implemented by **`org.jboss.ha.framework.interfaces.TransactionStickyFirstAvailableIdenticalAllProxies`**.

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

CHAPTER 18. 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 18.1, "The JBoss Enterprise Application Platform clustering architecture" shows the main pieces:

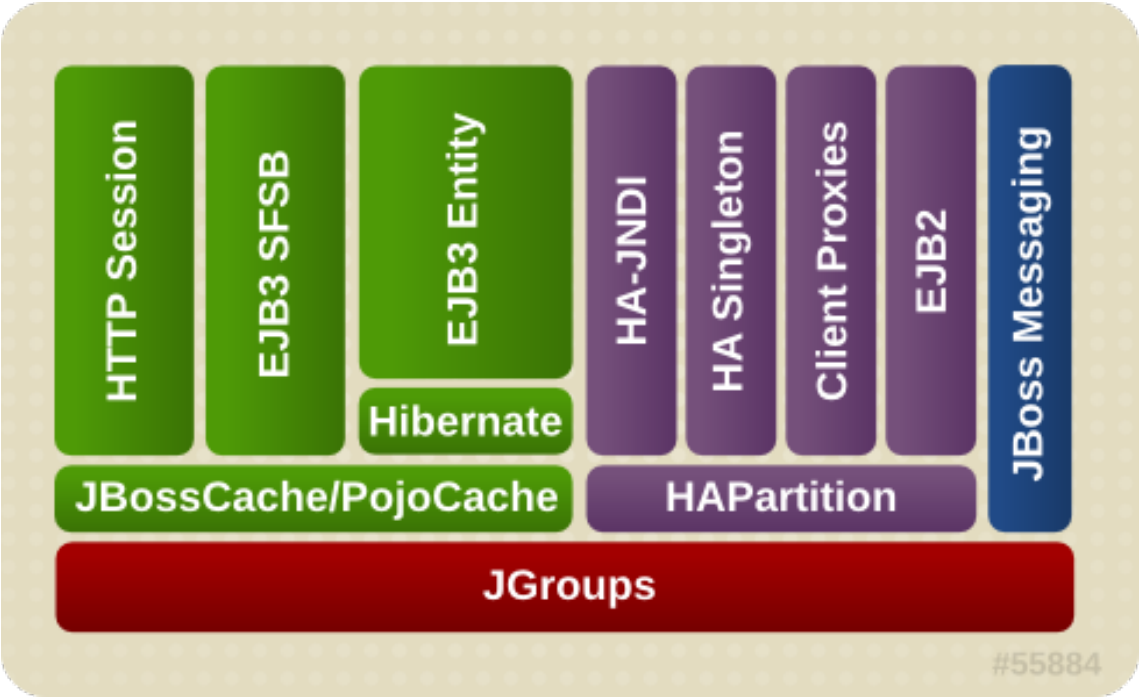


Figure 18.1. The JBoss Enterprise Application Platform clustering architecture

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 18.2, "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 18.3, "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.

18.1. GROUP COMMUNICATION WITH JGROUPS

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

The characteristics of a JGroups **Channel** are determined by the set of *protocols* that compose it. Each protocol handles a single aspect of the overall group communication task; for example the **UDP** protocol handles the details of sending and receiving UDP datagrams. A **Channel** that uses the **UDP** protocol is capable of communicating with UDP unicast and multicast; alternatively one that uses the **TCP** protocol uses TCP unicast for all messages. JGroups supports a wide variety of different protocols (see Section 25.1, "Configuring a JGroups Channel's Protocol Stack" for details), but the Enterprise Application Platform ships with a default set of channel configurations that should meet most needs.

By default, UDP multicast is used by all JGroups channels used by the Enterprise Application Platform (except for one TCP-based channel used by JBoss Messaging).

18.1.1. The Channel Factory Service

A significant difference in JBoss Enterprise Application Platform 5 versus previous releases is that JGroups Channels needed by clustering services (for example, a channel used by a distributed HttpSession cache) are no longer configured in detail as part of the consuming service's configuration, and are no longer directly instantiated by the consuming service. Instead, a new **ChannelFactory** service is used as a registry for named channel configurations and as a factory for **Channel** instances. A service that needs a channel requests the channel from the **ChannelFactory**, passing in the name of the desired configuration.

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

NOTE

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

18.1.1.1. Standard Protocol Stack Configurations

The standard protocol stack configurations that ship with Enterprise Application Platform 5 are described below. Note that not all of these are actually used; many are included as a convenience to users who may wish to alter the default server configuration. The configurations actually used in a stock Enterprise Application Platform 5 **all** configuration are **udp**, **jbm-control** and **jbm-data**, with all clustering services other than JBoss Messaging using **udp**.

You can add a new stack configuration by adding a new `stack` element to the `server/all/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml` file. You can alter the behavior of an existing configuration by editing this file. Before doing this though, have a look at the other standard configurations the Enterprise Application Platform ships; perhaps one of those meets your needs. Also, please note that before editing a configuration you should understand what services are using that configuration; make sure the change you are making is appropriate for all affected services. If the change isn't appropriate for a particular service, perhaps its better to create a new configuration and change some services to use that new configuration.

- **udp**

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

- **udp-async**

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

- **udp-sync**

  UDP multicast based stack, without flow control and without message bundling. This can be used instead of **udp** if (1) synchronous calls are used and (2) the message volume (rate and size) is not that large. Don't use this configuration if you send messages at a high sustained rate, or you might run out of memory.

- **tcp**

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

- **tcp-sync**

  TCP based stack, without flow control and without message bundling. TCP stacks are usually used when IP multicasting cannot be used in a network (e.g.routers discard multicast). This configuration should be used instead of `tcp` above when (1) synchronous calls are used and (2) the message volume (rate and size) is not that large. Don't use this configuration if you send messages at a high sustained rate, or you might run out of memory.

- **jbm-control**

  Stack optimized for the JBoss Messaging Control Channel. By default uses the same UDP transport protocol configuration as is used for the default **udp** stack defined above. This allows the JBoss Messaging Control Channel to use the same sockets, network buffers and thread pools as are used by the other standard JBoss Enterprise Application Platform clustered services (see Section 18.1.2, “The JGroups Shared Transport”)

- **jbm-data**

TCP-based stack optimized for the JBoss Messaging Data Channel.

18.1.2. The JGroups Shared Transport

As the number of JGroups-based clustering services running in the Enterprise Application Platform has risen over the years, the need to share the resources (particularly sockets and threads) used by these channels became a glaring problem. A stock Enterprise Application Platform 5 **all** configuration will connect 4 JGroups channels during startup, and a total of 7 or 8 will be connected if distributable web apps, clustered EJB3 SFSBs and a clustered JPA/Hibernate second level cache are all used. So many channels can consume a lot of resources, and can be a real configuration nightmare if the network environment requires configuration to ensure cluster isolation.

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

To configure a transport protocol for sharing, simply add a **`singleton_name="someName"`** attribute to the protocol's configuration. All channels whose transport protocol configuration uses the same **`singleton_name`** value will share their transport. All other protocols in the stack will not be shared. Figure 18.2, “Services using a Shared Transport” illustrates 4 services running in a VM, each with its own channel. Three of the services are sharing a transport; the fourth is using its own transport.

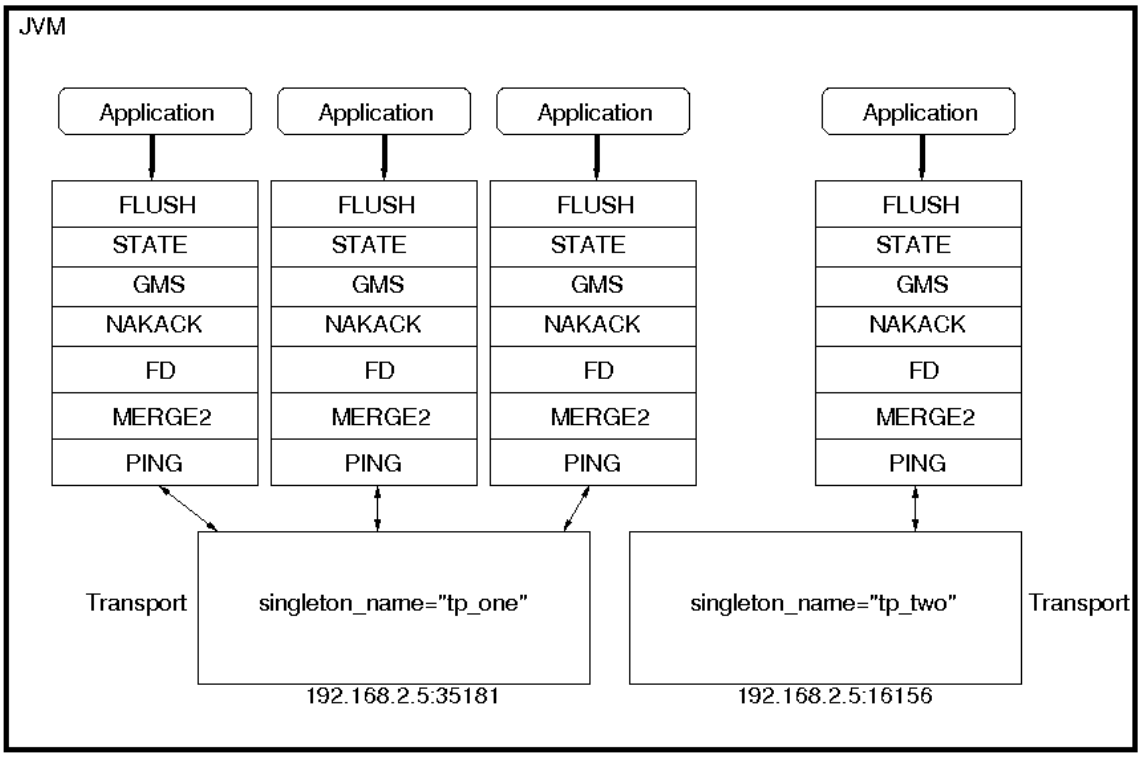


Figure 18.2. Services using a Shared Transport

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

18.2. 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 26, *JBoss Cache Configuration and Deployment*for more on this.

18.2.1. 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 Enterprise Application Platform 5 way.

In JBoss Enterprise Application Platform 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.

18.2.1.1. 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 26.2.1, "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 configuration appropriate for JPA/Hibernate entity/collection caching that uses JBoss Cache's MVCC locking (see notes below).

- **optimistic-entity**

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

- **pessimistic-entity**

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

- **mvcc-entity-repeatable**

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

- **pessimistic-entity-repeatable**

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

- **local-query**

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

- **replicated-query**

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

- **timestamps-cache**

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

- **mvcc-shared**

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

- **optimistic-shared**

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

- **pessimistic-shared**

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

- **mvcc-shared-repeatable**

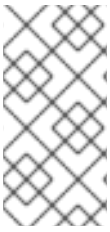
  Same as "mvcc-shared" but uses JBoss Cache's REPEATABLE_READ isolation level instead of READ_COMMITTED (see notes below).

- **pessimistic-shared-repeatable**

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

NOTE

For more on JBoss Cache's locking schemes, see Section 26.1.4, “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.

18.2.1.2. 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 configuration 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 configuration shows the standard aliases in Enterprise Application Platform 5:

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

    . . .

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

18.3. 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 allows services that use it to share a single **Channel** and multiplex RPC invocations over it, eliminating the configuration complexity and runtime overhead of having each service create its own **Channel**. HAPartition also supports a distributed registry of which clustering services are running on which cluster members. It provides notifications to interested listeners when the cluster membership changes or the clustered service registry changes. HAPartition forms the core of many of the clustering services we'll be discussing in the rest of this guide, including smart client-side clustered proxies, EJB 2 SFSB replication and entity cache management, farming, HA-JNDI and HA singletons. Custom services can also make use of HAPartition.

The following snippet shows the **HAPartition** service definition packaged with the standard JBoss Enterprise Application Platform distribution. This configuration can be found in the **server/all/deploy/cluster/hapartition-jboss-beans.xml** file.

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

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

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

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

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

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

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

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

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

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

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

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

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

 <property name="threadPool"><inject
bean="jboss.system:service=ThreadPool"/></property>
 <property name="distributedStateImpl">
 <bean name="DistributedState"
class="org.jboss.ha.framework.server.DistributedStateImpl">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX

(name="jboss:service=DistributedState,partitionName=${jboss.partition.name
:DefaultPartition}",
exposedInterface=org.jboss.ha.framework.server.DistributedStateImplMBean.c
lass, registerDirectly=true)</annotation>

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

```
  </bean>
  </property>
</bean>
```

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

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

- **partitionName** 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 server startup.
- **nodeAddress** is unused and can be ignored.
- **stateTransferTimeout** specifies the timeout (in milliseconds) for initial application state transfer. State transfer refers to the process of obtaining a serialized copy of initial application state from other already-running cluster members at service startup. Its default value is **30000**.
- **methodCallTimeout** specifies the timeout (in milliseconds) for obtaining responses to group RPCs from the other cluster members. Its default value is **60000**.

The **HAPartitionCacheHandler** is a small utility service that helps the HAPartition integrate with JBoss Cache (see Section 18.2.1, "The JBoss Enterprise Application Platform CacheManager Service"). HAPartition exposes a child service called DistributedState (see Section 18.3.2, "DistributedState Service") that uses JBoss Cache; the **HAPartitionCacheHandler** helps ensure consistent configuration between the JGroups **Channel** used by Distributed State's cache and the one used directly by HAPartition.

- **cacheConfigName** the name of the JBoss Cache configuration to use for the HAPartition-related cache. Indirectly, this also specifies the name of the JGroups protocol stack configuration HAPartition should use. See Section 26.1.5, "JGroups Integration" for more on how the JGroups protocol stack is configured.

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

You can view the current cluster information by pointing your browser to the JMX console of any JBoss instance in the cluster (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.

18.3.1. 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 17.2.1, “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.

18.3.2. 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 Enterprise Application Platform 5 the **`DistributedState`** service actually delegates to an underlying JBoss Cache instance.

18.3.3. 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.

CHAPTER 19. 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.

19.1. 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 19.2, “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:

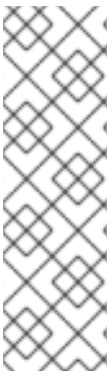
- It avoids migration issues with applications that assume that their JNDI implementation is local. This allows clustering to work out-of-the-box with just a few tweaks of configuration files.
- In a homogeneous cluster, this configuration actually cuts down on the amount of network traffic. A homogenous 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.

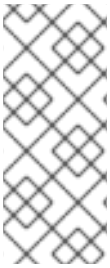


NOTE

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.

19.2. 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.

19.2.1. For clients running inside the Enterprise Application Platform

If you want to access HA-JNDI from inside the Enterprise Application Platform, you must explicitly configure your **`InitialContext`** by passing in JNDI properties to the constructor. The following code shows how to create a naming Context bound to HA-JNDI:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
// HA-JNDI is listening on the address passed to JBoss via -b
String bindAddress = System.getProperty("jboss.bind.address",
"localhost");
p.put(Context.PROVIDER_URL, bindAddress + ":1100"); // HA-JNDI address and
port.
return new InitialContext(p);
```

The Context.PROVIDER_URL property points to the HA-JNDI service configured in the **`deploy/cluster/hajndi-jboss-beans.xml`** file (see Section 19.3, “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 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.

19.2.1.1. 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.

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

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

No other jndi.properties file should be deployed inside the Enterprise Application Platform 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.

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

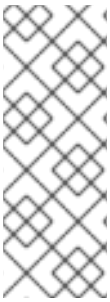
Go into the the jmx-console and execute the **list** operation on the **jboss:service=JNDIView** mbean. Towards the bottom of the results, the contents of the "HA-JNDI Namespace" are listed. Typically this will be empty; if any of your own deployments are shown there and you didn't explicitly bind them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

19.2.2. For clients running outside the Enterprise Application Platform

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 19.3, “JBoss configuration” for how to configure the servers and ports).

```
java.naming.provider.url=server1:1100,server2:1100,server3:1100,server4:11
00
```

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



NOTE

There is no load balancing behavior in the JNP client lookup process itself. It just goes through the provider lists and uses the first available server to obtain the stub. The HA-JNDI provider list only needs to contain a subset of HA-JNDI nodes in the cluster; once the HA-JNDI stub is downloaded, the stub will include information on all the available servers. A good practice is to include a set of servers such that you are certain that at least one of those in the list will be available.

The downloaded smart proxy contains the list of currently running nodes and the logic to load balance naming requests and to fail-over to another node if necessary. Furthermore, each time a JNDI invocation is made to the server, the list of targets in the proxy interceptor is updated (only if the list has changed since the last call).

If the property string **java.naming.provider.url** is empty or if all servers it mentions are not

reachable, the JNP client will try to discover a HA-JNDI server through a multicast call on the network (auto-discovery). See Section 19.3, "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) 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.

19.3. 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</pro
perty>

      <!-- 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 18.3, "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 18.3, "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 HA-JNDI 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 LoadBalancePolicy implementation that should be included in the client proxy. See Chapter 16, *Introduction and Quick Start* 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`.

19.3.1. Adding a Second HA-JNDI Service

It is possible to start several HA-JNDI services that use different HAPartitions. This can be used, for example, if a node is part of many logical clusters. In this case, make sure that you set a different port or IP address for each service. For instance, if you wanted to hook up HA-JNDI to the example cluster

you set up and change the binding port, the bean descriptor would look as follows (properties that do not vary from the standard deployments are omitted):

```
   <-- Cache Handler for secondary HAPartition -->
   <bean name="SecondaryHAPartitionCacheHandler"

class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
         <property name="cacheManager"><inject bean="CacheManager"/>
</property>
         <property name="cacheConfigName">secondary-ha-
partition</property>
   </bean>

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

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

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

exposedInterface=org.jboss.ha.framework.server.ClusterPartitionMBean.class
, registerDirectly=true)</annotation>

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

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

      ....
   </bean>

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

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

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

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

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

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

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

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

CHAPTER 20. 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.

20.1. STATELESS SESSION BEAN IN EJB 3.0

Clustering stateless session beans is probably the easiest case since no state is involved. Calls can be load balanced to 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, simply annotate the bean class with the **`@Clustered`** annotation. This annotation contains optional parameters for overriding both the load balance policy and partition to use.

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

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

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

  `org.jboss.ha.client.loadbalance.RoundRobin`

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

  `org.jboss.ha.client.loadbalance.RandomRobin`

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

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

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

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

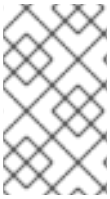
  Similar to **`FirstAvailable`**, except that the favored target is shared across all proxies.

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
   }
}
```

Rather than using the **@Clustered** annotation, you can also enable clustering for a session bean in jboss.xml:

```
<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>NonAnnotationStateful</ejb-name>
         <clustered>true</clustered>
         <cluster-config>
            <partition-name>FooPartition</partition-name>
            <load-balance-
policy>org.jboss.ha.framework.interfaces.RandomRobin</load-balance-policy>
         </cluster-config>
      </session>
   </enterprise-beans>
</jboss>
```



NOTE

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

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

20.2. 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.

20.2.1. The EJB application configuration

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

The **@org.jboss.ejb3.annotation.CacheConfig** annotation can also be applied to the bean to override the default caching behavior. 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 20.2.3, "CacheManager service configuration". 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 (regardless 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>
```

20.2.2. 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.

20.2.3. CacheManager service configuration

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The **`CacheManager`** service, described in Section 18.2.1, “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="useLockStriping">false</property>
  <property name="cacheMode">REPL_ASYNC</property>

  <!--
    Number of milliseconds to wait until all responses for a
    synchronous call have been received. Make this longer
    than lockAcquisitionTimeout.
```

```
  -->
  <property name="syncReplTimeout">17500</property>
  <!-- Max number of milliseconds to wait for a lock acquisition -->
  <property name="lockAcquisitionTimeout">15000</property>
  <!-- The max amount of time (in milliseconds) we wait until the
  state (ie. the contents of the cache) are retrieved from
  existing members at startup. -->
  <property name="stateRetrievalTimeout">60000</property>

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

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

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

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

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

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

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

      <!-- Do not change these -->
      <property name="autoDataGravitation">false</property>
      <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 (a.k.a. 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.

20.3. 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.

20.4. 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 **HASessionStateService** bean to manage distributed session states for clustered EJB 2.x stateful session beans. In this section, we cover both the session bean configuration and the **HASessionStateService** bean configuration.

20.4.1. 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 **HASessionStateService** to be used by this bean.

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

20.4.2. 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.

20.4.3. The HASessionStateService configuration

The **`HASessionStateService`** bean is defined in the **`<profile>/deploy/cluster/ha-legacy-jboss-beans.xml`** file.

```
<bean name="HASessionStateService"
      class="org.jboss.ha.hasessionstate.server.HASessionStateService">

  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
  (name="jboss:service=HASessionState",
  exposedInterface=org.jboss.ha.hasessionstate.server.
  HASessionStateServiceMBean.class,
  registerDirectly=true)</annotation>

  <!--  Partition used for group RPCs -->
  <property name="HAPartition"><inject bean="HAPartition"/></property>

  <!-- JNDI name under which the service is bound -->
  <property name="jndiName">/HASessionState/Default</property>
  <!-- Max delay before cleaning unreclaimed state.
       Defaults to 30*60*1000 => 30 minutes -->
  <property name="beanCleaningDelay">0</property>

</bean>
```

The configuration attributes in the **`HASessionStateService`** bean are listed below.

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

20.4.4. Handling Cluster Restart

We have covered the HA smart client architecture in Section 17.2.1, “Client-side interceptor architecture”. The default HA smart proxy client can only failover as long as one node in the cluster exists. If there is a complete cluster shutdown, the proxy becomes orphaned and loses knowledge of the available nodes in the cluster. There is no way for the proxy to recover from this. The proxy needs to look up a fresh set of targets out of JNDI/HA-JNDI when the nodes are restarted.

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

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

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.proxy.ejb.RetryInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </home>
        <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>
```

20.4.5. 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 VM is possible.

2. If the retryEnv field is null, it will check for any environment properties bound to a ThreadLocal by the org.jboss.naming.NamingContextFactory class. To use this class as your naming context factory, 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 Chapter 19, *Clustered JNDI Services* for more on multicast discovery of HA-JNDI.

20.4.6. 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 a previous 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. The SingleRetryInterceptor is now part of the default client interceptor stacks for clustered EJBs.

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

CHAPTER 21. 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.

21.1. 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.

21.1.1. Configure the distributed cache

To avoid round trips to the database, you can use a cache for your entities. JBoss EJB 3.0 entity beans are implemented by Hibernate, which has support for a second-level cache. The second-level cache provides the following functionalities:

- If you persist a cache-enabled entity bean instance to the database via the entity manager, the entity will be inserted into the cache.
- If you update an entity bean instance, and save the changes to the database via the entity manager, the entity will be updated in the cache.
- If you remove an entity bean instance from the database via the entity manager, the entity will be removed from the cache.
- If loading a cached entity from the database via the entity manager, and that entity does not exist in the database, it will be inserted into the cache.

As well as a region for caching entities, the second-level cache also contains regions for caching collections, queries, and timestamps. The Hibernate setup used for the JBoss EJB 3.0 implementation uses JBoss Cache as its underlying second-level cache implementation.

Configuration of a the second-level cache is done via your EJB3 deployment's **`persistence.xml`**, like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/persistence"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="tempdb" transaction-type="JTA">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.use_second_level_cache"
value="true"/>
      <property name="hibernate.cache.use_query_cache" value="true"/>
      <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
      <!-- region factory specific properties -->
      <property name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
      <property name="hibernate.cache.region.jbc2.cfg.entity" value="mvcc-
entity"/>
```

```
      <property name="hibernate.cache.region.jbc2.cfg.collection"
value="mvcc-entity"/>
    </properties>
  </persistence-unit>
</persistence>
```

hibernate.cache.use_second_level_cache

Enables second-level caching of entities and collections.

hibernate.cache.use_query_cache

Enables second-level caching of queries.

hibernate.cache.region.factory_class

Defines the **`RegionFactory`** implementation that dictates region-specific caching behavior. Hibernate ships with 2 types of JBoss Cache-based second-level caches: shared and multiplexed.

A shared region factory uses the same Cache for all cache regions - much like the legacy CacheProvider implementation in older Hibernate versions.

Hibernate ships with 2 shared region factory implementations:

org.hibernate.cache.jbc2.SharedJBossCacheRegionFactory

Uses a single JBoss Cache configuration, from a newly instantiated CacheManager, for all cache regions.

Table 21.1. Additional properties for SharedJBossCacheRegionFactory

Property	Default	Description
hibernate.cache.region.jbc2.cfg.shared	treecache.xml	The classpath or filesystem resource containing the JBoss Cache configuration settings.
hibernate.cache.region.jbc2.cfg.jgroups.stacks	org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	The classpath or filesystem resource containing the JGroups protocol stack configurations.

org.hibernate.cache.jbc2.JndiSharedJBossCacheRegionFactory

Uses a single JBoss Cache configuration, from an existing CacheManager bound to JNDI, for all cache regions.

Table 21.2. Additional properties for JndiSharedJBossCacheRegionFactory

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

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

Table 21.3. Common properties for multiplexed region factory implementations

Property	Default	Description
hibernate.cache.region.jbc2.cfg.entity	optimistic-entity	The JBoss Cache configuration used for the entity cache region. Alternative configurations: mvcc-entity, pessimistic-entity, mvcc-entity-repeatable, optimistic-entity-repeatable, pessimistic-entity-repeatable
hibernate.cache.region.jbc2.cfg.collection	optimistic-entity	The JBoss Cache configuration used for the collection cache region. The collection cache region typically uses the same configuration as the entity cache region.
hibernate.cache.region.jbc2.cfg.query	local-query	The JBoss Cache configuration used for the query cache region. By default, cached query results are not replicated. Alternative configurations: replicated-query
hibernate.cache.region.jbc2.cfg.ts	timestamps-cache	The JBoss Cache configuration used for the timestamp cache region. If query caching is used, the corresponding timestamp cache must be replicating, even if the query cache is non-replicating. The timestamp cache region must never share the same cache as the query cache.

Hibernate ships with 2 shared region factory implementations:

org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory

Uses separate JBoss Cache configurations, from a newly instantiated CacheManager, per cache region.

Table 21.4. Additional properties for MultiplexedJBossCacheRegionFactory

Property	Default	Description
hibernate.cache.region.jbc2.configs	org/hibernate/cache/jbc2/builder/jbc2-configs.xml	The classpath or filesystem resource containing the JBoss Cache configuration settings.

Property	Default	Description
hibernate.cache.region.jbc2.cfg.jgroups.stacks	org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	The classpath or filesystem resource containing the JGroups protocol stack configurations.

org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory

Uses separate JBoss Cache configurations, from a JNDI-bound CacheManager, see Section 18.2.1, "The JBoss Enterprise Application Platform CacheManager Service" , per cache region.

Table 21.5. Additional properties for JndiMultiplexedJBossCacheRegionFactory

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

Now, we have JBoss Cache configured to support distributed caching of EJB 3.0 entity beans. We still have to configure individual entity beans to use the cache service.

21.1.2. Configure the entity beans for cache

Next we need to configure which entities to cache. The default is to not cache anything, even with the settings shown above. We use the **@org.hibernate.annotations.Cache** annotation to tag entity beans that needs to be cached.

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

A very simplified rule of thumb is that you will typically want to do caching for objects that rarely change, and which are frequently read. You can fine tune the cache for each entity bean in the appropriate JBoss Cache configuration file, e.g. **jboss-cache-manager-jboss-beans.xml**. For instance, you can specify the size of the cache. If there are too many objects in the cache, the cache can evict the oldest or least used objects, depending on configuration, to make room for new objects. Assuming the region_prefix specified in **persistence.xml** was **myprefix**, the default name of the cache region for the **com.mycompany.entities.Account** entity bean would be **/myprefix/com/mycompany/entities/Account**.

```
<bean name="..." class="org.jboss.cache.config.Configuration">
   ... ...
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
```

```
      <property name="wakeupInterval">5000</property>
      <!--  Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <!-- Evict LRU node once we have more than this number of
nodes -->
              <property name="maxNodes">10000</property>
              <!-- And, evict any node that hasn't been accessed in this
many seconds -->
              <property name="timeToLiveSeconds">1000</property>
              <!-- Don't evict a node that's been accessed within this
many seconds.
                   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">/myprefix/com/mycompany/entities/Account</property>
            <property name="evictionAlgorithmConfig">
              <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                <property name="maxNodes">10000</property>
                <property name="timeToLiveSeconds">5000</property>
                <property name="minTimeToLiveSeconds">120</property>
              </bean>
            </property>
          </bean>
           ... ...
        </list>
      </property>
    </bean>
  </property>
</bean>
```

If you do not specify a cache region for an entity bean class, all instances of this class will be cached using the **`defaultEvictionRegionConfig`** as defined above. The @Cache annotation exposes an optional attribute "region" that lets you specify the cache region where an entity is to be stored, rather than having it be automatically be created from the fully-qualified class name of the entity class.

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

The eviction configuration would then become:

-

```
<bean name="..." class="org.jboss.cache.config.Configuration">
   ... ...
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeupInterval">5000</property>
      <!--  Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">5000</property>
              <property name="timeToLiveSeconds">1000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
      </property>
      <property name="evictionRegionConfigs">
        <list>
          <bean class="org.jboss.cache.config.EvictionRegionConfig">
            <property name="regionName">/myprefix/Account</property>
            <property name="evictionAlgorithmConfig">
              <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                <property name="maxNodes">10000</property>
                <property name="timeToLiveSeconds">5000</property>
                <property name="minTimeToLiveSeconds">120</property>
              </bean>
            </property>
          </bean>
           ... ...
        </list>
      </property>
    </bean>
  </property>
</bean>
```

21.1.3. Query result caching

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

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

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

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

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

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

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

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

```
<bean name="..." class="org.jboss.cache.config.Configuration">
    ... ...
   <property name="evictionConfig">
       <bean class="org.jboss.cache.config.EvictionConfig">
         <property name="wakeupInterval">5000</property>
         <!--  Overall default -->
         <property name="defaultEvictionRegionConfig">
            <bean class="org.jboss.cache.config.EvictionRegionConfig">
               <property name="regionName">/</property>
               <property name="evictionAlgorithmConfig">
                  <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                     <property name="maxNodes">5000</property>
                     <property name="timeToLiveSeconds">1000</property>
                     <property name="minTimeToLiveSeconds">120</property>
                  </bean>
               </property>
            </bean>
         </property>
         <property name="evictionRegionConfigs">
            <list>
               <bean class="org.jboss.cache.config.EvictionRegionConfig">
                  <property
name="regionName">/myprefix/Account</property>
                  <property name="evictionAlgorithmConfig">
                     <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                        <property name="maxNodes">10000</property>
                        <property
name="timeToLiveSeconds">5000</property>
                        <property
name="minTimeToLiveSeconds">120</property>
                     </bean>
                  </property>
```

```
               </bean>
               <bean class="org.jboss.cache.config.EvictionRegionConfig">
                  <property
name="regionName">/myprefix/org/hibernate/cache/StandardQueryCache</proper
ty>
                  <property name="evictionAlgorithmConfig">
                     <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                        <property name="maxNodes">100</property>
                        <property name="timeToLiveSeconds">600</property>
                        <property
name="minTimeToLiveSeconds">120</property>
                     </bean>
                  </property>
               </bean>
            </list>
         </property>
      </bean>
   </property>
</bean>
```

The @NamedQuery.hints attribute shown above takes an array of vendor-specific @QueryHints as a value. Hibernate accepts the "org.hibernate.cacheRegion" query hint, where the value is the name of a cache region to use instead of the default /org/hibernate/cache/StandardQueryCache. For example:

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

The related eviction configuration:

```
<bean name="..." class="org.jboss.cache.config.Configuration">
    ... ...
   <property name="evictionConfig">
       <bean class="org.jboss.cache.config.EvictionConfig">
         <property name="wakeupInterval">5000</property>
         <!--  Overall default -->
         <property name="defaultEvictionRegionConfig">
            <bean class="org.jboss.cache.config.EvictionRegionConfig">
               <property name="regionName">/</property>
               <property name="evictionAlgorithmConfig">
```

```
                  <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                     <property name="maxNodes">5000</property>
                     <property name="timeToLiveSeconds">1000</property>
                     <property name="minTimeToLiveSeconds">120</property>
                  </bean>
               </property>
            </bean>
         </property>
         <property name="evictionRegionConfigs">
            <list>
               <bean class="org.jboss.cache.config.EvictionRegionConfig">
                  <property
name="regionName">/myprefix/Account</property>
                  <property name="evictionAlgorithmConfig">
                     <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                        <property name="maxNodes">10000</property>
                        <property
name="timeToLiveSeconds">5000</property>
                        <property
name="minTimeToLiveSeconds">120</property>
                     </bean>
                  </property>
               </bean>
               <bean class="org.jboss.cache.config.EvictionRegionConfig">
                  <property
name="regionName">/myprefix/Queries</property>
                  <property name="evictionAlgorithmConfig">
                     <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                        <property name="maxNodes">100</property>
                        <property name="timeToLiveSeconds">600</property>
                        <property
name="minTimeToLiveSeconds">120</property>
                     </bean>
                  </property>
               </bean>
                ... ...
            </list>
         </property>
      </bean>
   </property>
</bean>
```

21.2. 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 special case situation of read-only, or one read-write node with read-only nodes synchronized 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.

NOTE

If you are using Bean Managed Persistence (BMP), you are going to have to implement synchronization on your own.

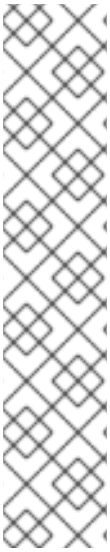
CHAPTER 22. HTTP SERVICES

HTTP session replication is used to replicate the state associated with web client sessions to other nodes in a cluster. Thus, in the event one of your nodes crashes, another node in the cluster will be able to recover. Two distinct functions must be performed:

- Session state replication
- Load-balancing HTTP Requests

State replication is directly handled by JBoss. When you run JBoss in the `all` configuration, session state replication is enabled by default. Just configure your web application as `<distributable>` in its `web.xml` (see Section 22.2, “Configuring HTTP session state replication”), deploy it, and its session state is automatically replicated across all JBoss instances in the cluster.

However, load-balancing is a different story; it is not handled by JBoss itself and requires an external load balancer. This function could be provided by specialized hardware switches or routers (Cisco LoadDirector for example) or by specialized software running on commodity hardware. As a very common scenario, we will demonstrate how to set up a software load balancer using Apache httpd and mod_jk.



NOTE

A load-balancer tracks HTTP requests and, depending on the session to which the request is linked, it dispatches the request to the appropriate node. This is called load-balancing with sticky-sessions or session affinity: once a session is created on a node, every future request will also be processed by that same node. Using a load-balancer that supports sticky-sessions but not configuring your web application for session replication allows you to scale very well by avoiding the cost of session state replication: each request for a session will always be handled by the same node. But in case a node dies, the state of all client sessions hosted by this node (the shopping carts, for example) will be lost and the clients will most probably need to login on another node and restart with a new session. In many situations, it is acceptable not to replicate HTTP sessions because all critical state is stored in a database or on the client. In other situations, losing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

22.1. 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.

22.1.1. 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 the latest stable 2.2.x version. We will assume, for the next sections, that you have installed Apache in the **APACHE_HOME** directory.

Next, download mod_jk binaries. Several versions of mod_jk exist as well. We strongly advise the use of mod_jk 1.2.x, as both earlier versions of mod_jk, and mod_jk2, are deprecated, unsupported and no further development is going on in the community. The mod_jk 1.2.x binary can be downloaded from

http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/. Rename the downloaded file to **mod_jk.so** and copy it under **APACHE_HOME/modules/**.

22.1.2. 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 serve 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.

22.1.3. 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.ping_mode=A
worker.node1.lbfactor=1

# 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.ping_mode=A
worker.node2.lbfactor=1

# 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 (that is, JBoss Enterprise Application Platform instances) node1 and node2 listening on port 8009.

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

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

The **ping_mode** attribute enables CPing/CPong. It determines when established connections are probed to determine whether they are still working. In this case, **ping_mode** is set to **A**, which means that the connection is probed once after connecting to the backend, before sending each request to the backend, and at regular intervals during the internal maintenance cycle.

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.

22.1.4. Configuring JBoss to work with mod_jk

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

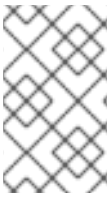
On each clustered JBoss node, we have to name the node according to the name specified in **workers.properties**. For instance, on JBoss instance node1, edit the **JBOSS_HOME/server/all/deploy/jbossweb.sar/server.xml** file (replace **/all** with your own server name if necessary). Locate the **<Engine>** 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 protocol="AJP/1.3" port="8009" address="${jboss.bind.address}"
   redirectPort="8443" />
```

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

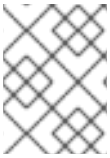


NOTE

For more updated information on using mod_jk 1.2 with JBoss AS, please refer to the JBoss wiki page at **http://www.jboss.org/community/wiki/UsingModjk12WithJBoss**.

22.1.5. Configuring the NSAPI connector on Solaris

This section shows you how to configure the NSAPI connector to use a JBoss Enterprise Platform as a worker node for a Sun Java System Web Server (SJWS) master node.



NOTE

Sun Java System Web Server has recently been renamed to the Oracle iPlanet Web Server.

In this section, all of the server instances are on the same machine. To use different machines for each instance, use the **-b** switch to bind your instance of JBoss Enterprise Platform to a public IP address. Remember to edit the **workers.properties** file on the SJWS machine to reflect these changes in IP address.

22.1.5.1. Prerequisites

This section assumes that:

- Your worker node(s) are already installed with a JBoss Enterprise Platform 5.1 or later. The Native components are not a requirement of the NSAPI connector. Refer to the *Installation Guide* for assistance with this prerequisite.
- Your master node is already installed with any of the following technology combinations, and the appropriate Native binary for its operating system and architecture. Refer to the *Installation Guide* for assistance with this prerequisite.

- Solaris 9 x86 with Sun Java System Web Server 6.1 SP12
- Solaris 9 SPARC 64 with Sun Java System Web Server 6.1 SP12
- Solaris 10 x86 with Sun Java System Web Server 7.0 U8
- Solaris 10 SPARC 64 with Sun Java System Web Server 7.0 U8

22.1.5.2. Configure JBoss Enterprise Platform as a Worker Node

This section shows you how to safely configure your JBoss Enterprise Platform instance as a worker node for use with Sun SJWS.

Procedure 22.1. Configure a JBoss Enterprise Platform instance as a worker node

1. **Create a server profile for each worker node**
   Make a copy of the server profile that you wish to configure as a worker node. (This procedure uses the **default** server profile.)

   ```
   [user@workstation jboss-eap-5.1]$ cd jboss-as/server
   [user@workstation server]$ cp -r default/ default-01
   [user@workstation server]$ cp -r default/ default-02
   ```

2. **Give each instance a unique name**
   Edit the following line in the **deploy/jbossweb.sar/server.xml** file of each new worker instance:

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

   Add a unique **jvmRoute** value, as shown. This value is the identifier for this node in the cluster.

   For the **default-01** server profile:

   ```
   <Engine name="jboss.web" defaultHost="localhost"
   jvmRoute="worker01">
   ```

   For the **default-02** server profile:

   ```
   <Engine name="jboss.web" defaultHost="localhost"
   jvmRoute="worker02">
   ```

3. **Enable session handling**
   Edit the following line in the **deployers/jbossweb.deployer/META-INF/war-deployers-jboss-beans.xml** file of each worker node:

   ```
   <property name="useJK">false</property>
   ```

   This property controls whether special session handling is used to coordinate with mod_jk and other connector variants. Set this property to **true** in both worker nodes:

   ```
   <property name="useJK">true</property>
   ```

4. **Start your worker nodes**
   Start each worker node in a separate command line interface. Ensure that each node is bound to a different IP address with the **-b** switch.

```
[user@workstation jboss-eap-5.1]$ ./jboss-as/bin/run.sh -b 127.0.0.1
-c default-01
```

```
[user@workstation jboss-eap-5.1]$ ./jboss-as/bin/run.sh -b
127.0.0.100 -c default-02
```

22.1.5.3. Configure Sun Java System Web Server for Clustering

The procedures in the following sections assume that the contents of the Native zip appropriate for your operating system and architecture have been extracted to **/tmp/connectors/jboss-ep-native-5.1/**. This path is referred to as *NATIVE* in the procedures that follow. These procedures also assume that the **/tmp/connectors** directory is used to store logs, properties files and NSAPI locks.

These procedures also assume that your installation of Sun Java System Web Server is in one of the following locations, depending on your version of Solaris:

- for Solaris 9 x86 or SPARC 64: **/opt/SUNWwbsrv61/**
- for Solaris 10 x86 or SPARC 64: **/opt/SUNWwbsrv70/**

This path is referred to as *SJWS* in the procedures that follow.

Procedure 22.2. Initial clustering configuration

1. **Disable servlet mappings**
   Under *Built In Servlet Mappings* in the ***SJWS/PROFILE*/config/default-web.xml** file, disable the mappings for the following servlets, as shown in the code sample:
   - default
   - invoker
   - jsp

```
<!-- ==================== Built In Servlet Mappings
===================== -->

<!-- The servlet mappings for the built in servlets defined above. -
->

<!-- The mapping for the default servlet -->
<!--servlet-mapping>
 <servlet-name>default</servlet-name>
 <url-pattern>/</url-pattern>
</servlet-mapping-->

<!-- The mapping for the invoker servlet -->
<!--servlet-mapping>
 <servlet-name>invoker</servlet-name>
 <url-pattern>/servlet/*</url-pattern>
```

```
</servlet-mapping-->

<!-- The mapping for the JSP servlet -->
<!--servlet-mapping>
 <servlet-name>jsp</servlet-name>
 <url-pattern>*.jsp</url-pattern>
</servlet-mapping-->
```

2. **Load the required modules and properties**
   Append the following lines to the ***SJWS/PROFILE*/config/magnus.conf** file:

```
Init fn="load-modules" funcs="jk_init,jk_service"
shlib="NATIVE/lib/nsapi_redirector.so" shlib_flags="(global|now)"
Init fn="jk_init" worker_file="/tmp/connectors/workers.properties"
log_level="debug" log_file="/tmp/connectors/nsapi.log"
shm_file="/tmp/connectors/jk_shm"
```

   These lines define the location of the **nsapi_redirector.so** module used by the **jk_init** and **jk_service** functions, and the location of the **workers.properties** file, which defines the worker nodes and their attributes.

NOTE

The **lib** directory in the ***NATIVE*/lib/nsapi_redirector.so** path applies only to 32-bit machines. On 64-bit machines, this directory is called **lib64**.

22.1.5.3.1. Configure a basic cluster with NSAPI

Use the following procedure to configure a basic cluster, where requests for particular paths are forwarded to particular worker nodes. In Procedure 22.3, “Configure a basic cluster with NSAPI” , worker02 serves the **/nc** path, while worker01 serves **/status** and all other paths defined in the first part of the **obj.conf** file.

Procedure 22.3. Configure a basic cluster with NSAPI

1. **Define the paths to serve via NSAPI**
   Edit the ***SJWS/PROFILE*/config/obj.conf** file. Define paths that should be served via NSAPI at the end of the **default** Object definition, as shown:

```
<Object name="default">
      [...]
   NameTrans fn="assign-name" from="/status" name="jknsapi"
   NameTrans fn="assign-name" from="/images(|/*)" name="jknsapi"
   NameTrans fn="assign-name" from="/css(|/*)" name="jknsapi"
   NameTrans fn="assign-name" from="/nc(|/*)" name="jknsapi"
   NameTrans fn="assign-name" from="/jmx-console(|/*)"
name="jknsapi"
</Object>
```

   You can map the path of any application deployed on your JBoss Enterprise Platform instance in this **obj.conf** file. In the example code, the **/nc** path is mapped to an application deployed under the name **nc**.

2. **Define the worker that serves each path**
   Edit the ***SJWS/PROFILE*/config/obj.conf** file and add the following **jknsapi** Object definition after the **default** Object definition.

```
<Object name="jknsapi">
   ObjectType fn=force-type type=text/plain
   Service fn="jk_service" worker="worker01" path="/status"
   Service fn="jk_service" worker="worker02" path="/nc(/*)"
   Service fn="jk_service" worker="worker01"
</Object>
```

   This **jknsapi** Object defines the worker nodes used to serve each path that was assigned to **name="jknsapi"** in the **default** Object.

   In the example code, the third Service definition does not specify a **path** value, so the worker node defined (**worker01**) serves all of the paths assigned to **jknsapi** by default. In this case, the first Service definition in the example code, which assigns the **/status** path to **worker01**, is superfluous.

3. **Define the workers and their attributes**
   Create a **workers.properties** file in the location you defined in Step 2. Define the list of worker nodes and each worker node's properties in this file, like so:

```
# An entry that lists all the workers defined
worker.list=worker01, worker02

# Entries that define the host and port associated with these
workers
worker.worker01.host=127.0.0.1
worker.worker01.port=8009
worker.worker01.type=ajp13

worker.worker02.host=127.0.0.100
worker.worker02.port=8009
worker.worker02.type=ajp13
```

22.1.5.3.2. Configure a Load-balanced Cluster with NSAPI

Procedure 22.4. Configure a load-balancing cluster with NSAPI

1. **Define the paths to serve via NSAPI**
   Edit the ***SJWS/PROFILE*/config/obj.conf** file. Define paths that should be served via NSAPI at the end of the **default** Object definition, as shown:

```
<Object name="default">
      [...]
   NameTrans fn="assign-name" from="/status" name="jknsapi"
   NameTrans fn="assign-name" from="/images(|/*)" name="jknsapi"
   NameTrans fn="assign-name" from="/css(|/*)" name="jknsapi"
   NameTrans fn="assign-name" from="/nc(|/*)" name="jknsapi"
   NameTrans fn="assign-name" from="/jmx-console(|/*)"
name="jknsapi"
   NameTrans fn="assign-name" from="/jkmanager/*" name="jknsapi"
</Object>
```

You can map the path of any application deployed on your JBoss Enterprise Platform instance in this **obj.conf** file. In the example code, the **/nc** path is mapped to an application deployed under the name **nc**.

2. **Define the worker that serves each path**
   Edit the ***SJWS/PROFILE*/config/obj.conf** file and add the following **jknsapi** Object definition after the **default** Object definition.

```
<Object name="jknsapi">
   ObjectType fn=force-type type=text/plain
   Service fn="jk_service" worker="status" path="/jkmanager(/*)"
   Service fn="jk_service" worker="router"
</Object>
```

   This **jknsapi** Object defines the worker nodes used to serve each path that was assigned to **name="jknsapi"** in the **default** Object.

3. **Define the workers and their attributes**
   Create a **workers.properties** file in the location you defined in Step 2. Define the list of worker nodes and each worker node's properties in this file, like so:

```
# The advanced router LB worker
worker.list=router,status

# Define a worker using ajp13
worker.worker01.port=8009
worker.worker01.host=127.0.0.1
worker.worker01.type=ajp13
worker.worker01.ping_mode=A
worker.worker01.socket_timeout=10
worker.worker01.lbfactor=3

# Define another worker using ajp13
worker.worker02.port=8009
worker.worker02.host=127.0.0.100
worker.worker02.type=ajp13
worker.worker02.ping_mode=A
worker.worker02.socket_timeout=10
worker.worker02.lbfactor=1

# Define the LB worker
worker.router.type=lb
worker.router.balance_workers=worker01,worker02

# Define the status worker
worker.status.type=status
```

22.1.5.3.3. Restart Sun Java System Web Server

Once your Sun Java System Web Server instance is configured, restart it so that your changes take effect.

For Sun Java System Web Server 6.1:

```
SJWS/PROFILE/stop
SJWS/PROFILE/start
```

For Sun Java System Web Server 7.0:

```
SJWS/PROFILE/bin/stopserv
SJWS/PROFILE/bin/startserv
```

22.2. 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 22.1.3, “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, if a server node fails or is shut down, the load balancer can fail over the next client request to any server node and obtain the same session state.

22.2.1. Enabling session replication in your application

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

```
<?xml version="1.0"?>
<web-app  xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                              http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd"
          version="2.4">

    <distributable/>

</web-app>
```

You can futher configure session replication using the **replication-config** element in the **jboss-web.xml** file. However, the **replication-config** element only needs to be set if one or more of the default values described below is unacceptable. Here is an example:

```
<!DOCTYPE jboss-web PUBLIC
    -//JBoss//DTD Web Application 5.0//EN
    http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd>

<jboss-web>

   <replication-config>
      <cache-name>custom-session-cache</cache-name>
      <replication-trigger>SET</replication-trigger>
      <replication-granularity>ATTRIBUTE</replication-granularity>
      <replication-field-batch-mode>true</replication-field-batch-mode>
```

```
      <use-jk>false</use-jk>
      <max-unreplicated-interval>30</max-unreplicated-interval>
      <snapshot-mode>INSTANT</snapshot-mode>
      <snapshot-interval>1000</snapshot-interval>
      <session-notification-
policy>com.example.CustomSessionNotificationPolicy</session-notification-
policy>
   </replication-config>

</jboss-web>
```

All of the above configuration elements are optional and can be ommitted if the default value is acceptable. A couple are commonly used; the rest are very infrequently changed from the defaults. We'll cover the commonly used ones first.

The **`replication-trigger`** element determines when the container should consider that session data must be replicated across the cluster. The rationale for this setting is that after a mutable object stored as a session attribute is accessed from the session, in the absence of a **`setAttribute`** call the container has no clear way to know if the object (and hence the session state) has been modified and needs to be replicated. This element has 3 valid values:

- **SET_AND_GET** is conservative but not optimal (performance-wise): it will always replicate session data even if its content has not been modified but simply accessed. This setting made (a little) sense in JBoss Enterprise Application Platform 4 since using it was a way to ensure that every request triggered replication of the session's timestamp. Since setting **`max_unreplicated_interval`** to 0 accomplishes the same thing at much lower cost, using **SET_AND_GET** makes no sense with Enterprise Application Platform 5.

- **SET_AND_NON_PRIMITIVE_GET** is conservative but will only replicate if an object of a non-primitive type has been accessed (i.e. the object is not of a well-known immutable JDK type such as **`Integer`**, **`Long`**, **`String`**, etc.) This is the default value.

- **SET** assumes that the developer will explicitly call **`setAttribute`** on the session if the data needs to be replicated. This setting prevents unnecessary replication and can have a major beneficial impact on performance, but requires very good coding practices to ensure **`setAttribute`** is always called whenever a mutable object stored in the session is modified.

In all cases, calling **`setAttribute`** marks the session as needing replication.

The **`replication-granularity`** element determines the granularity of what gets replicated if the container determines session replication is needed. The supported values are:

- **SESSION** indicates that the entire session attribute map should be replicated when any attribute is considered modified. Replication occurs at request end. This option replicates the most data and thus incurs the highest replication cost, but since all attributes values are always replicated together it ensures that any references between attribute values will not be broken when the session is deserialized. For this reason it is the default setting.

- **ATTRIBUTE** indicates that only attributes that the session considers to be potentially modified are replicated. Replication occurs at request end. For sessions carrying large amounts of data, parts of which are infrequently updated, this option can significantly increase replication performance. However, it is not suitable for applications that store objects in different attributes that share references with each other (e.g. a **`Person`** object in the "husband" attribute sharing with another **`Person`** in the "wife" attribute a reference to an **`Address`** object). This is because if the attributes are separately replicated, when the session is deserialized on remote nodes the shared references will be broken.

- **FIELD** is useful if the classes stored in the session have been bytecode enhanced for use by POJO Cache. If they have been, the session management layer will detect field level changes within objects stored to the session, and will replicate only those changes. This is the most performant setting. Replication is only for individual changed data fields inside session attribute objects. Shared object references will be preserved across the cluster. Potentially most performant, but requires changes to your application (this will be discussed later).

The other elements under the **replication-config** element are much less frequently used.

The **cacheName** element indicates the name of the JBoss Cache configuration that should be used for storing distributable sessions and replicating them around the cluster. This element lets web applications that require different caching characteristics specify the use of separate, differently configured, JBoss Cache instances. In JBoss Enterprise Application Platform 4 the cache to use was a server-wide configuration that could not be changed per web application. The default value is **standard-session-cache** if the **replication-granularity** is not **FIELD**, **field-granularity-session-cache** if it is. See Section 22.2.3, "Configuring the JBoss Cache instance used for session state replication" for more details on JBoss Cache configuration for web tier clustering.

The **replication-field-batch-mode** element indicates whether all replication messages associated with a request will be batched into one message. This is applicable only if **replication-granularity** is **FIELD**. If **replication-field-batch-mode** is set to **true**, fine-grained changes made to objects stored in the session attribute map will replicate only when the HTTP request is finished; otherwise they replicate as they occur. Setting this to **false** is not advised. Default is **true**.

The **useJK** element indicates whether the container should assume that a JK-based software load balancer (e.g. mod_jk, mod_proxy, mod_cluster) is being used for load balancing for this web application. If set to **true**, the container will examine the session ID associated with every request and replace the **jvmRoute** portion of the session ID if it detects a failover.

The default value is **null** (i.e. unspecified). In this case the session manager will use the presence or absence of a **jvmRoute** configuration on its enclosing JBoss Web **Engine** (see Section 22.1.4, "Configuring JBoss to work with mod_jk") to determine whether JK is used.

You need only set this to **false** for web applications whose URL cannot be handled by the JK load balancer.

The **max-unreplicated-interval** element configures the maximum interval between requests, in seconds, after which a request will trigger replication of the session's timestamp regardless of whether the request has otherwise made the session dirty. Such replication ensures that other nodes in the cluster are aware of the most recent value for the session's timestamp and won't incorrectly expire an unreplicated session upon failover. It also results in correct values for **HttpSession.getLastAccessedTime()** calls following failover.

A value of **0** means the timestamp will be replicated whenever the session is accessed. A value of **-1** means the timestamp will be replicated only if some other activity during the request (e.g. modifying an attribute) has resulted in other replication work involving the session. A positive value greater than the **HttpSession.getMaxInactiveInterval()** value will be treated as probable misconfiguration and converted to **0**; i.e. replicate the metadata on every request. Default value is **60**.

The **snapshot-mode** element configures when sessions are replicated to the other nodes. Possible values are **INSTANT** (the default) and **INTERVAL**.

The typical value, **INSTANT**, replicates changes to the other nodes at the end of requests, using the request processing thread to perform the replication. In this case, the **snapshot-interval** property is ignored.

With **INTERVAL** mode, a background task is created that runs every **snapshot-interval** milliseconds, checking for modified sessions and replicating them.

Note that this property has no effect if **replication-granularity** is set to **FIELD**. If it is **FIELD**, **instant** mode will be used.

The **snapshot-interval** element defines how often (in milliseconds) the background task that replicates modified sessions should be started for this web application. Only meaningful if **snapshot-mode** is set to **interval**.

The **session-notification-policy** element specifies the fully qualified class name of the implementation of the **ClusteredSessionNotificationPolicy** interface that should be used to govern whether servlet specification notifications should be emitted to any registered **HttpSessionListener**, **HttpSessionAttributeListener** and/or **HttpSessionBindingListener**.

Event notifications that may make sense in a non-clustered environment may or may not make sense in a clustered environment; see https://jira.jboss.org/jira/browse/JBAS-5778 for an example of why a notification may not be desired. Configuring an appropriate **ClusteredSessionNotificationPolicy** gives the application author fine-grained control over what notifications are issued.

In previous releases, the default value if not explicitly set is the **LegacyClusteredSessionNotificationPolicy**, which implements the behavior in previous JBoss versions. In JBoss Enterprise Application Platform 5, this was changed to **IgnoreUndeployLegacyClusteredSessionNotificationPolicy**, which implements the same behavior except during undeployment, during which no **HttpSessionListener** and **HttpSessionAttributeListener** notifications are sent.

22.2.2. HttpSession Passivation and Activation

Passivation is the process of controlling memory usage by removing relatively unused sessions from memory while storing them in persistent storage. If a passivated session is requested by a client, it can be "activated" back into memory and removed from the persistent store. JBoss Enterprise Application Platform 5 supports passivation of HttpSessions from web applications whose **web.xml** includes the **distributable** tag (i.e. clustered web applications).

Passivation occurs at three points during the lifecycle of a web application:

- When the container requests the creation of a new session. If the number of currently active sessions exceeds a configurable limit, an attempt is made to passivate sessions to make room in memory.
- Periodically (by default every ten seconds) as the JBoss Web background task thread runs.
- When the web application is deployed and a backup copy of sessions active on other servers is acquired by the newly deploying web application's session manager.

A session will be passivated if one of the following holds true:

- The session has not been in use for longer than a configurable maximum idle time.

- The number of active sessions exceeds a configurable maximum and the session has not been in use for longer than a configurable minimum idle time.

In both cases, sessions are passivated on a Least Recently Used (LRU) basis.

22.2.2.1. Configuring HttpSession Passivation

Session passivation behavior is configured via the **jboss-web.xml** deployment descriptor in your web application's **WEB-INF** directory.

```
<!DOCTYPE jboss-web PUBLIC
    -//JBoss//DTD Web Application 5.0//EN
    http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd>

<jboss-web>

   <max-active-sessions>20</max-active-sessions>
   <passivation-config>
      <use-session-passivation>true</use-session-passivation>
      <passivation-min-idle-time>60</passivation-min-idle-time>
      <passivation-max-idle-time>600</passivation-max-idle-time>
   </passivation-config>


</jboss-web>
```

- **max-active-session**

  Determines the maximum number of active sessions allowed. If the number of sessions managed by the the session manager exceeds this value and passivation is enabled, the excess will be passivated based on the configured **passivation-min-idle-time**. If after passivation is completed (or if passivation is disabled), the number of active sessions still exceeds this limit, attempts to create new sessions will be rejected. If set to **-1** (the default), there is no limit.

- **use-session-passivation**

  Determines whether session passivation will be enabled for the web application. Default is **false**.

- **passivation-min-idle-time**

  Determines the minimum time (in seconds) that a session must have been inactive before the container will consider passivating it in order to reduce the active session count to obey the value defined by **max-active-sessions**. A value of **-1** (the default) disables passivating sessions before **passivation-max-idle-time**. Neither a value of **-1** nor a high value are recommended if **max-active-sessions** is set.

- **passivation-max-idle-time**

  Determines the maximum time (in seconds) that a session can be inactive before the container should attempt to passivate it to save memory. Passivation of such sessions will take place regardless of whether the active session count exceeds **max-active-sessions**. Should be less than the **web.xml session-timeout** setting. A value of **-1** (the default) disables passivation based on maximum inactivity.

The total number of sessions in memory includes sessions replicated from other cluster nodes that are not being accessed on this node. Take this into account when setting **max-active-sessions**. The number of sessions replicated from other nodes will also depend on whether *buddy replication* is enabled.

Say, for example, that you have an eight node cluster, and each node handles requests from 100 users. With *total replication*, each node would store 800 sessions in memory. With *buddy replication* enabled, and the default **numBuddies** setting (**1**), each node will store 200 sessions in memory.

22.2.3. Configuring the JBoss Cache instance used for session state replication

The container for a distributable web application makes use of JBoss Cache to provide HTTP session replication services around the cluster. The container integrates with the CacheManager service to obtain a reference to a JBoss Cache instance (see Section 18.2.1, "The JBoss Enterprise Application Platform CacheManager Service").

The name of the JBoss Cache configuration to use is controlled by the **cacheName** element in the application's **jboss-web.xml** (see Section 22.2.1, "Enabling session replication in your application"). In most cases, though, this does not need to be set as the default values of **standard-session-cache** and **field-granularity-session-cache** (for applications configured for FIELD granularity) are appropriate.

The JBoss Cache configurations in the **CacheManager** service expose a number of options. See Chapter 26, *JBoss Cache Configuration and Deployment*and the JBoss Cache documentation for a more complete discussion. The **standard-session-cache** and **field-granularity-session-cache** configurations are already optimized for the web session replication use case, and most of the settings should not be altered. Administrators may be interested in altering the following settings:

- **cacheMode**

  The default is **REPL_ASYNC**, which specifies that a session replication message sent to the cluster does not wait for responses from other cluster nodes confirming that the message has been received and processed. The alternative mode, **REPL_SYNC**, offers a greater degree of confirmation that session state has been received, but reduces performance significantly. See Section 26.1.2, "Cache Mode" for further details.

- **enabled** property in the **buddyReplicationConfig** section

  Set to **true** to enable buddy replication. See Section 26.1.8, "Buddy Replication". Default is **false**.

- **numBuddies** property in the **buddyReplicationConfig** section

  Set to a value greater than the default (**1**) to increase the number of backup nodes onto which sessions are replicated. Only relevant if buddy replication is enabled. See Section 26.1.8, "Buddy Replication".

- **buddyPoolName** property in the **buddyReplicationConfig** section

  A way to specify a preferred replication group when buddy replication is enabled. JBoss Cache tries to pick a buddy who shares the same pool name (falling back to other buddies if not available). Only relevant if buddy replication is enabled. See Section 26.1.8, "Buddy Replication".

- **multiplexerStack**

Name of the JGroups protocol stack the cache should use. See Section 18.1.1, “The Channel Factory Service”.

- **clusterName**

  Identifying name JGroups will use for this cache's channel. Only change this if you create a new cache configuration, in which case this property should have a different value from all other cache configurations.

If you wish to use a completely new JBoss Cache configuration rather than editing one of the existing ones, please see Section 26.2.1, “Deployment Via the CacheManager Service”.

22.3. USING FIELD-LEVEL REPLICATION

WARNING

This feature is deprecated as of JBoss Enterprise Web Platform 5.1, and will be removed in a future release of JBoss Enterprise Web Platform. Customers are recommended to migrate away from this feature in existing implementations, and not use it in new implementations.

FIELD-level replication only replicates modified data fields inside objects stored in the session. It can reduce the data traffic between clustered nodes, and hence improve the performance of the whole cluster. To use FIELD-level replication, you must first prepare (that is, bytecode enhance) your Java class to allow the session cache to detect when fields in cached objects have been changed and need to be replicated.

First, you need to identify the classes that you need to prepare. You can identify these classes by using annotations, like so:

```
@org.jboss.cache.pojo.annotation.Replicable
public class Address
{
...
}
```

If you annotate a class with **@Replicable**, then all of its subclasses will be automatically annotated as well. Similarly, you can annotate an interface with **@Replicable** and all of its implementing classes will be annotated. For example:

```
@org.jboss.cache.aop.InstanceOfAopMarker
public class Person
{
...
}

public class Student extends Person
```

```
{
...
}
```

There is no need to annotate **Student**. POJO Cache will recognize it as @Replicable because it is a sub-class of **Person**.

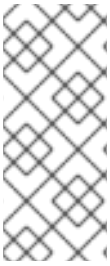
JBoss Enterprise Application Platform 5 requires JDK 5 at runtime, but some users may still need to build their projects using JDK 1.4. In this case, annotating classes can be done via JDK 1.4 style annotations embedded in JavaDocs. For example:

```
/**
 * Represents a street address.
 * @org.jboss.cache.pojo.annotation.Replicable
 */
public class Address
{
...
}
```

Once you have annotated your classes, you will need to perform a pre-processing step to bytecode enhance your classes for use by POJO Cache. You need to use the JBoss AOP pre-compiler **annotationc** and post-compiler **aopc** to process the above source code before and after they are compiled by the Java compiler. The **annotationc** step is only need if the JDK 1.4 style annotations are used; if JDK 5 annotations are used it is not necessary. Here is an example of how to invoke those commands from command line.

```
$ annotationc [classpath] [source files or directories]
$ javac -cp [classpath] [source files or directories]
$ aopc [classpath] [class files or directories]
```

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 of how to build, deploy, and validate a FIELD-level replicated web application from this page: http://www.jboss.org/community/wiki/httpsessionfieldlevelexample. The example bundles the pre- and post-compile tools so you do not need to download JBoss AOP separately.

Finally, let's see an example on how to use FIELD-level replication on those data classes. First, we see some servlet code that reads some data from the request parameters, creates a couple of objects and stores them in the session:

```
Person husband = new Person(getHusbandName(request),
getHusbandAge(request)); Person wife = new
Person(getWifeName(request), getWifeAge(request)); Address addr = new
Address();
addr.setPostalCode(getPostalCode(request));

husband.setAddress(addr);
```

```
wife.setAddress(addr); // husband and wife share the same address!

session.setAttribute("husband", husband); // that's it.
session.setAttribute("wife", wife); // that's it.
```

Later, a different servlet could update the family's postal code:

```
Person wife = (Person)session.getAttribute("wife");
wife.getAddress().setPostalCode(getPostalCode(request));
// this will update and replicate the postal code
```

Notice that in there is no need to call **`session.setAttribute()`** after you make changes to the data object, and all changes to the fields are automatically replicated across the cluster.

Besides plain objects, you can also use regular Java collections of those objects as session attributes. POJO Cache automatically figures out how to handle those collections and replicate field changes in their member objects.

22.4. USING CLUSTERED SINGLE SIGN-ON (SSO)

JBoss supports clustered single sign-on, allowing a user to authenticate to one web application and to be recognized on all web applications that are deployed on the same virtual host, whether or not they are deployed on that same machine or on another node in the cluster. Authentication replication is handled by JBoss Cache. Clustered single sign-on support is a JBoss-specific extension of the non-clustered **`org.apache.catalina.authenticator.SingleSignOn`** valve that is a standard part of Tomcat and JBoss Web. Both the non-clustered and clustered versions allow users to sign on to any one of the web apps associated with a virtual host and have their identity recognized by all other web 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.

22.4.1. 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 Section 18.2.1, “The JBoss Enterprise Application Platform CacheManager Service”) 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`**, 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 22.4.4, "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`**.

22.4.2. SSO Behavior

The user will not be challenged as long as they access only unprotected resources in any of the web applications on the virtual host.

Upon access to a protected resource in any web 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.

22.4.3. 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).

22.4.4. 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" />
```

CHAPTER 23. JBOSS MESSAGING CLUSTERING NOTES

The most current information about using JBoss Messaging in a clustered environment is always available from the relevant *JBoss Messaging User Guide* at http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/.

CHAPTER 24. CLUSTERED DEPLOYMENT OPTIONS

24.1. CLUSTERED SINGLETON SERVICES

A clustered singleton service (also known as a HA singleton) is a service that is deployed on multiple nodes in a cluster, but is providing its service on only one of the nodes. The node running the singleton service is typically called the master node.

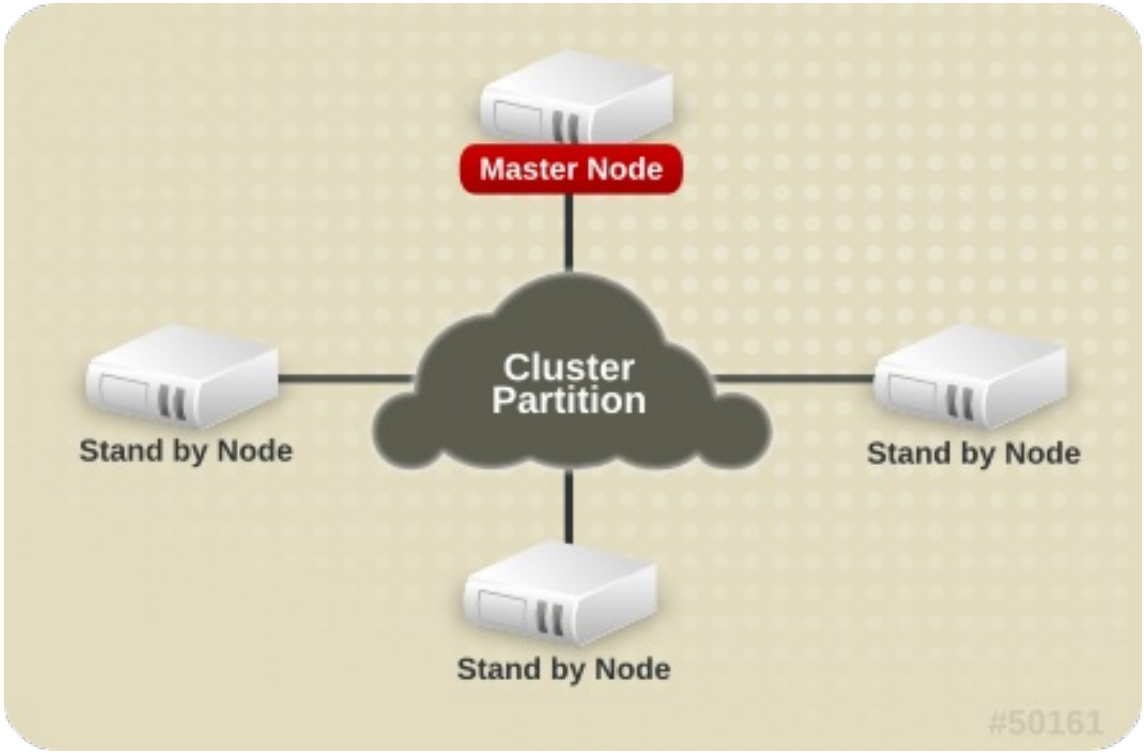


Figure 24.1. Topology before the Master Node fails

When the master fails or is shut down, another master is selected from the remaining nodes and the service is restarted on the new master. Thus, other than a brief interval when one master has stopped and another has yet to take over, the service is always being provided by one but only one node.

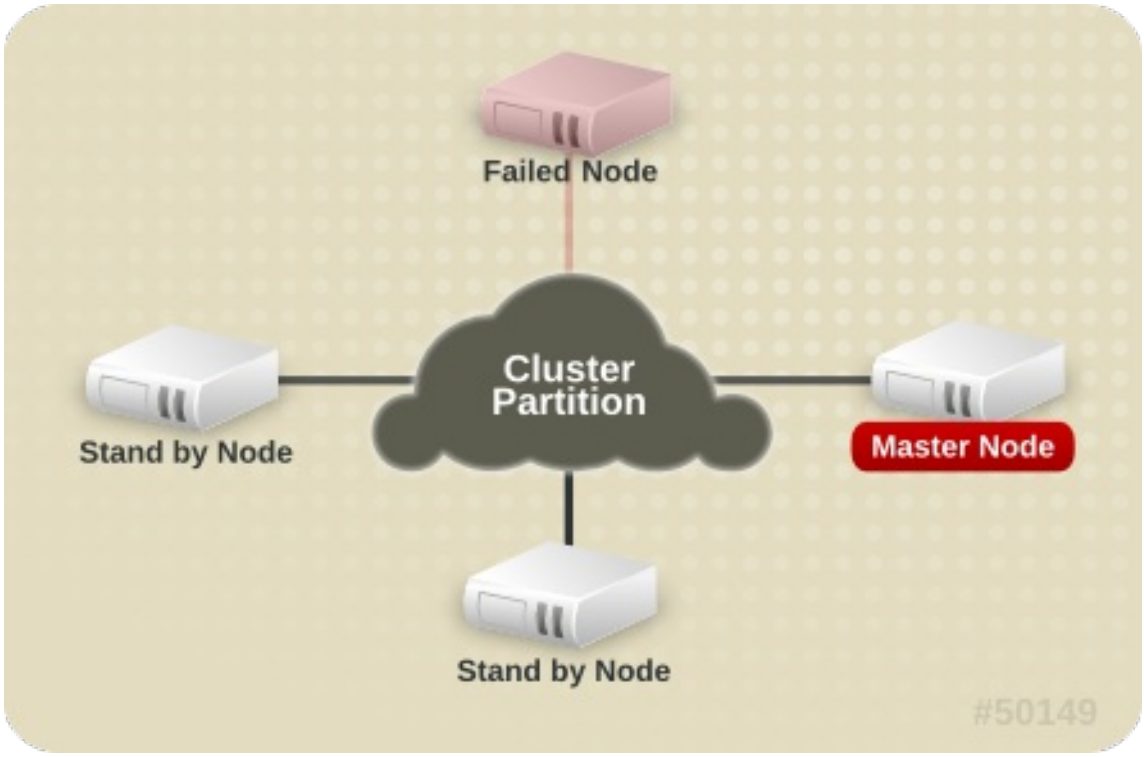


Figure 24.2. Topology after the Master Node fails

24.1.1. 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.

24.1.1.1. 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`** nor **`farm`** directories, 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 **`HASingletonDeployer`** bean (which itself is deployed via the deploy/deploy-hasingleton-jboss-beans.xml file). The HASingletonDeployer service is itself an HA Singleton, one whose provided service, when it becomes master, is to deploy the contents of deploy-hasingleton; and whose service, when it stops being the master (typically at server shutdown), is to undeploy the contents of **`deploy-hasingleton`**.

So, by placing your deployments in **`deploy-hasingleton`** you know that they will be deployed only on the master node in the cluster. If the master node cleanly shuts down, they will be cleanly undeployed as part of shutdown. If the master node fails or is shut down, they will be deployed on 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 the complexity of your service's deployment, and the extent of startup activity in which it engages, this could take a while, during which time the service is not being provided.

24.1.1.2. POJO deployments using HASingletonController

If your service is a POJO (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 a POJO that we want to make an HA singleton. The only thing special about it is it needs to expose a public method that can be called when it should begin providing service, and another that can be called when it should stop providing service:

```
public interface HASingletonExampleMBean
{
   boolean isMasterNode();
}
```

```
public class HASingletonExample implements HASingletonExampleMBean
{
   private boolean isMasterNode = false;

   public boolean isMasterNode()
   {
      return isMasterNode;
   }

   public void startSingleton()
   {
      isMasterNode = true;
   }

   public void stopSingleton()
   {
      isMasterNode = false;
   }
}
```

We used **`startSingleton`** and **`stopSingleton`** in the above example, but you could name the methods anything.

Next, we deploy our service, along with an HASingletonController to control it, most likely packaged in a .sar file, with the following **`META-INF/jboss-beans.xml`**:

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
  <!-- This bean is an example of a clustered singleton -->
  <bean name="HASingletonExample"
class="org.jboss.ha.examples.HASingletonExample">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
```

```
      (name="jboss:service=HASingletonExample",

exposedInterface=org.jboss.ha.examples.HASingletonExampleMBean.class)
</annotation>
  </bean>

  <bean name="ExampleHASingletonController"
class="org.jboss.ha.singleton.HASingletonController">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
      (name="jboss:service=ExampleHASingletonController",

exposedInterface=org.jboss.ha.singleton.HASingletonControllerMBean.class,
      registerDirectly=true)</annotation>
    <property name="HAPartition"><inject bean="HAPartition"/></property>
    <property name="target"><inject bean="HASingletonExample"/></property>
    <property name="targetStartMethod">startSingleton</property>
    <property name="targetStopMethod">stopSingleton</property>
  </bean>
</deployment>
```

Voila! A clustered singleton service.

The primary advantage of this approach over deploy-ha-singleton. is that the above example can be placed in **deploy** or **farm** and thus can be hot deployed and farmed deployed. Also, if our example service had complex, time-consuming startup requirements, those could potentially be implemented in create() or start() methods. JBoss will invoke create() and start() as soon as the service is deployed; it doesn't wait until the node becomes the master node. So, the service could be primed and ready to go, just waiting for the controller to implement startSingleton() at which point it can immediately provide service.

Although not demonstrated in the example above, the **HASingletonController** can support an optional argument for either or both of the target start and stop methods. These are specified using the **targetStartMethodArgument** and **TargetStopMethodArgument** properties, respectively. Currently, only string values are supported.

24.1.1.3. 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 service:

```
<depends>HASingletonDeployerBarrierController</depends>
```

The way it works is that a BarrierController is deployed along with the HASingletonDeployer and listens for JMX notifications from it. A BarrierController is a relatively simple MBean that can subscribe to receive any JMX notification in the system. It uses the received notifications to control the lifecycle of a dynamically created MBean called the Barrier. The Barrier is instantiated, registered and brought to the CREATE state when the BarrierController is deployed. After that, the BarrierController starts and stops the Barrier when matching JMX notifications are received. Thus, other services need only depend on the Barrier bean using the usual <depends> tag, and they will be started and stopped in tandem with the Barrier. When the BarrierController is undeployed the Barrier is also destroyed.

This provides an alternative to the deploy-hasingleton approach in that we can use farming to distribute the service, while content in deploy-hasingleton must be copied manually on all nodes.

On the other hand, the barrier-dependent service will be instantiated/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.

24.1.2. 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 service maintains an attribute called the CurrentView, which is basically an ordered list of the current members of the cluster. As nodes join and leave the cluster, JGroups ensures that each surviving member of the cluster 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 uses the **`HAPartition`**'s **`HASingletonElectionPolicy`** to determine if they are the master, as explained in the Section 24.1.2.1, “HA singleton election policy” .

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.

24.1.2.1. 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 Enterprise Application Platform ships with two election policies:

`HASingletonElectionPolicySimple`

This policy selects a master node based relative age. The desired age is configured via the **`position`** property, which corresponds to the index in the list of available nodes. **`position = 0`**, the default, refers to the oldest node; **`position = 1`**, refers to the 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>
```

24.2. FARMING DEPLOYMENT

The easiest way to deploy an application into the cluster is to use the farming service. Using the farming service, you can deploy an application (e.g. EAR, WAR, or SAR; either an archive file or in exploded form) to the **`all/farm/`** directory of any cluster member and the application will be automatically duplicate across all nodes in the same cluster. If a node joins the cluster later, it will pull in all farm deployed applications in the cluster and deploy them locally at start-up time. If you delete the application from a running clustered server node's **`farm/`** directory, the application will be undeployed locally and then removed from all other clustered server nodes' **`farm/`** directories (triggering undeployment).

Farming is enabled by default in the **`all`** configuration in JBoss Enterprise Application Platform and thus requires no manual setup. The required **`farm-deployment-jboss-beans.xml`** and **`timestamps-jboss-beans.xml`** configuration files are located in the **`deploy/cluster`** directory. If you want to enable farming in a custom configuration, simply copy these files to the corresponding JBoss deploy directory ***`$JBOSS_HOME`*`/server/`*`$your_own_config`*`/deploy/cluster`**. Make sure that your custom configuration has clustering enabled.

While there is little need to customize the farming service, it can be customized via the **`FarmProfileRepositoryClusteringHandler`** bean, whose properties and default values are listed below:

```
<bean name="FarmProfileRepositoryClusteringHandler"
      class="org.jboss.profileservice.cluster.repository.
      DefaultRepositoryClusteringHandler">

  <property name="partition"><inject bean="HAPartition"/></property>
  <property name="profileDomain">default</property>
  <property name="profileServer">default</property>
  <property name="profileName">farm</property>
  <property name="immutable">false</property>
  <property name="lockTimeout">60000</property><!-- 1 minute -->
  <property name="methodCallTimeout">60000</property><!-- 1 minute -->
  <property name="synchronizationPolicy"><inject
bean="FarmProfileSynchronizationPolicy"/></property>
</bean>
```

- **partition** is a required attribute to inject the HAPartition service that the farm service uses for intra-cluster communication.
- **profile[Domain|Server|Name]** are all used to identify the profile for which this handler is intended.
- **immutable** indicates whether or not this handler allows a node to push content changes to the cluster. A value of **`true`** is equivalent to setting **`synchronizationPolicy`** to **`org.jboss.system.server.profileservice.repository.clustered.sync.ImmutableSynchronizationPolicy`**.
- **lockTimeout** defines the number of milliseconds to wait for cluster-wide lock acquisition.
- **methodCallTimeout** defines the number of milliseconds to wait for invocations on remote cluster nodes.
- **synchronizationPolicy** decides how to handle content additions, reincarnations, updates, or removals from nodes attempting to join the cluster or from cluster merges. The policy is consulted on the "authoritative" node, i.e. the master node for the service on the cluster. *Reincarnation* refers to the phenomenon where a newly started node may contain an application in its **`farm/`** directory that was previously removed by the farming service but might still exist on the starting node if it was not running when the removal took place. The default synchronization policy is defined as follows:

  ```
  <bean name="FarmProfileSynchronizationPolicy"
        class="org.jboss.profileservice.cluster.repository.
        DefaultSynchronizationPolicy">
    <property name="allowJoinAdditions"><null/></property>
    <property name="allowJoinReincarnations"><null/></property>
    <property name="allowJoinUpdates"><null/></property>
    <property name="allowJoinRemovals"><null/></property>
    <property name="allowMergeAdditions"><null/></property>
    <property name="allowMergeReincarnations"><null/></property>
    <property name="allowMergeUpdates"><null/></property>
    <property name="allowMergeRemovals"><null/></property>
    <property name="developerMode">false</property>
    <property name="removalTrackingTime">2592000000</property><!-- 30
  days -->
    <property name="timestampService"><inject
  bean="TimestampDiscrepancyService"/></property>
  </bean>
  ```

- **allow[Join|Merge][Additions|Reincarnations|Updates|Removals]** define fixed responses to requests to allow additions, reincarnations, updates, or removals from joined or merged nodes.
- **developerMode** enables a lenient synchronization policy that allows all changes. Enabling developer mode is equivalent to setting each of the above properties to `true` and is intended for development environments.
- **removalTrackingTime** defines the number of milliseconds for which this policy should remembered removed items, for use in detecting reincarnations.
- **timestampService** estimates and tracks discrepancies in system clocks for current and past members of the cluster. Default implementation is defined in `timestamps-jboss-beans.xml`.

CHAPTER 25. JGROUPS SERVICES

JGroups provides the underlying group communication support for JBoss Enterprise Application Platform clusters. The interaction of clustered services with JGroups was covered in Section 18.1, "Group Communication with JGroups". This chapter focuses on the details of this interaction, with particular attention to configuration details and troubleshooting tips.

This chapter is not intended as complete JGroups documentation. If you want to know more about JGroups, you can consult:

- The JGroups project documentation at http://jgroups.org/ug.html
- The JGroups wiki pages at jboss.org, rooted at https://www.jboss.org/community/wiki/JGroups

The first section of this chapter covers the many JGroups configuration options in detail. JBoss Enterprise Application Platform ships with a set of default JGroups configurations. Most applications will work with the default configurations out of the box. You will only need to edit these configurations when you deploy an application with special network or performance requirements.

25.1. CONFIGURING A JGROUPS CHANNEL'S PROTOCOL STACK

The JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster. Communication occurs over a communication channel. The channel built up from a stack of network communication *protocols*, each of which is responsible for adding a particular capability to the overall behavior of the channel. Key capabilities provided by various protocols include transport, cluster discovery, message ordering, lossless message delivery, detection of failed peers, and cluster membership management services.

Figure 25.1, "Protocol stack in JGroups" shows a conceptual cluster with each member's channel composed of a stack of JGroups protocols.

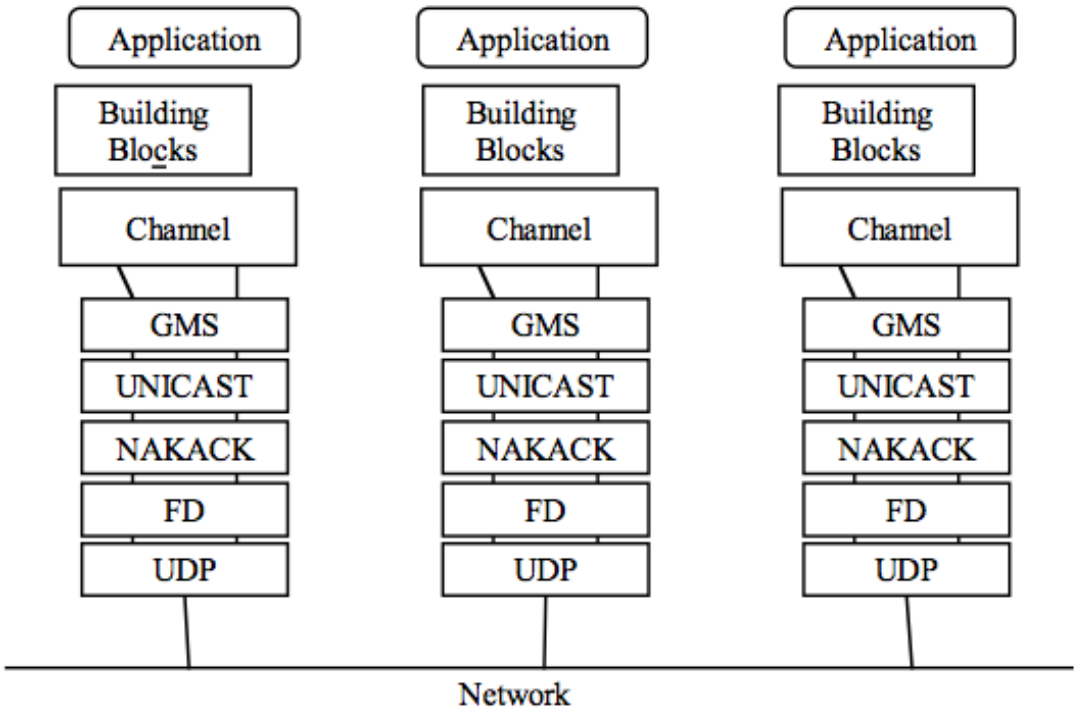


Figure 25.1. Protocol stack in JGroups

This section of the chapter covers some of the most commonly used protocols, according to the type of behaviour they add to the channel. We discuss a few key configuration attributes exposed by each protocol, but since these attributes should be altered only by experts, this chapter focuses on familiarizing users with the purpose of various protocols.

The JGroups configurations used in JBoss Enterprise Application Platform appear as nested elements in the ***$JBOSS_HOME*/server/all/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** file. This file is parsed by the **ChannelFactory** service, which uses the contents to provide correctly configured channels to the clustered services that require them. See Section 18.1.1, "The Channel Factory Service" for more on the **ChannelFactory** service.

The following is an example protocol stack configuration from **jgroups-channelfactory-stacks.xml**:

```
<stack name="udp-async"
           description="Same as the default 'udp' stack above, except
message bundling
                        is enabled in the transport protocol
(enable_bundling=true).
                        Useful for services that make high-volume
asynchronous
                        RPCs (e.g. high volume JBoss Cache instances
configured
                        for REPL_ASYNC) where message bundling may
improve performance.">
        <config>
          <UDP
             singleton_name="udp-async"
             mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
             mcast_addr="${jboss.partition.udpGroup:228.11.11.11}"
             tos="8"
             ucast_recv_buf_size="20000000"
             ucast_send_buf_size="640000"
             mcast_recv_buf_size="25000000"
             mcast_send_buf_size="640000"
             loopback="true"
             discard_incompatible_packets="true"
             enable_bundling="true"
             max_bundle_size="64000"
             max_bundle_timeout="30"
             ip_ttl="${jgroups.udp.ip_ttl:2}"
             thread_naming_pattern="cl"
             timer.num_threads="12"
             enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"

diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
             diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

             thread_pool.enabled="true"
             thread_pool.min_threads="8"
             thread_pool.max_threads="200"
             thread_pool.keep_alive_time="5000"
             thread_pool.queue_enabled="true"
             thread_pool.queue_max_size="1000"
             thread_pool.rejection_policy="discard"
```

```
            oob_thread_pool.enabled="true"
            oob_thread_pool.min_threads="8"
            oob_thread_pool.max_threads="200"
            oob_thread_pool.keep_alive_time="1000"
            oob_thread_pool.queue_enabled="false"
            oob_thread_pool.rejection_policy="discard"/>
          <PING timeout="2000" num_initial_members="3"/>
          <MERGE2 max_interval="100000" min_interval="20000"/>
          <FD_SOCK/>
          <FD timeout="6000" max_tries="5" shun="true"/>
          <VERIFY_SUSPECT timeout="1500"/>
          <BARRIER/>
          <pbcast.NAKACK use_mcast_xmit="true" gc_lag="0"
                   retransmit_timeout="300,600,1200,2400,4800"
                   discard_delivered_msgs="true"/>
          <UNICAST timeout="300,600,1200,2400,3600"/>
          <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                   max_bytes="400000"/>
          <VIEW_SYNC avg_send_interval="10000"/>
          <pbcast.GMS print_local_addr="true" join_timeout="3000"
                   shun="true"
                   view_bundling="true"
                   view_ack_collection_timeout="5000"
                   resume_task_timeout="7500"/>
          <FC max_credits="2000000" min_threshold="0.10"
              ignore_synchronous_response="true"/>
          <FRAG2 frag_size="60000"/>
          <!-- pbcast.STREAMING_STATE_TRANSFER/ -->
          <pbcast.STATE_TRANSFER/>
          <pbcast.FLUSH timeout="0" start_flush_timeout="10000"/>
        </config>
    </stack>
```

The **<config>** element contains all the configuration data for JGroups. This information is used to configure a JGroups *channel*, which is conceptually similar to a socket, and manages communication between peers in a cluster. Each element within the **<config>** element defines a particular JGroups *protocol*. Each protocol performs one function. The combination of these functions defines the characteristics of the channel as a whole. The next few sections describe common protocols and explain the options available to each.

25.1.1. Common Configuration Properties

The following property is exposed by all of the JGroups protocols discussed below:

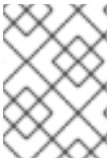
- **stats** whether the protocol should gather runtime statistics on its operations that can be exposed via tools like the AS's JMX console or the JGroups Probe utility. What, if any, statistics are gathered depends on the protocol. Default is **true**.

NOTE

All of the protocols in the versions of JGroups used in JBoss Application Server 3.x and 4.x exposed **down_thread** and **up_thread** attributes. The JGroups version included in JBoss Application Server 5 and later no longer uses those attributes, and a **WARN** message will be written to the server log if they are configured for any protocol.

25.1.2. Transport Protocols

The transport protocols send and receive messages to and from the network. They also manage the thread pools used to deliver incoming messages to addresses higher in the protocol stack. JGroups supports **UDP**, **TCP** and **TUNNEL** as transport protocols.



NOTE

The **UDP**, **TCP**, and **TUNNEL** protocols are mutually exclusive. You can only have one transport protocol in each JGroups **Config** element

25.1.2.1. UDP configuration

UDP is the preferred transport protocol for JGroups. UDP uses multicast (or, in an unusual configuration, multiple unicasts) to send and receive messages. If you choose UDP as the transport protocol for your cluster service, you need to configure it in the **UDP** sub-element in the JGroups **config** element. Here is an example.

```
            <UDP
               singleton_name="udp-async"
               mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
               mcast_addr="${jboss.partition.udpGroup:228.11.11.11}"
               tos="8"
               ucast_recv_buf_size="20000000"
               ucast_send_buf_size="640000"
               mcast_recv_buf_size="25000000"
               mcast_send_buf_size="640000"
               loopback="true"
               discard_incompatible_packets="true"
               enable_bundling="true"
               max_bundle_size="64000"
               max_bundle_timeout="30"
               ip_ttl="${jgroups.udp.ip_ttl:2}"
               thread_naming_pattern="cl"
               timer.num_threads="12"
               enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"

diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
               diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

               thread_pool.enabled="true"
               thread_pool.min_threads="8"
               thread_pool.max_threads="200"
               thread_pool.keep_alive_time="5000"
               thread_pool.queue_enabled="true"
               thread_pool.queue_max_size="1000"
               thread_pool.rejection_policy="discard"

               oob_thread_pool.enabled="true"
               oob_thread_pool.min_threads="8"
               oob_thread_pool.max_threads="200"
               oob_thread_pool.keep_alive_time="1000"
               oob_thread_pool.queue_enabled="false"
               oob_thread_pool.rejection_policy="discard"/>
```

JGroups transport configurations have a number of attributes available. First we look at the attributes available to the **UDP** protocol, followed by the attributes that are also used by the **TCP** and **TUNNEL** transport protocols.

The attributes particular to the **UDP** protocol are:

- **ip_mcast** specifies whether or not to use IP multicasting. The default is **true**. If set to **false**, multiple unicast packets will be sent instead of one multicast packet. Any packet sent via **UDP** protocol are UDP datagrams.

- **mcast_addr** specifies the multicast address (class D) for communicating with the group (i.e., the cluster). The standard protocol stack configurations in JBoss AS use the value of system property **jboss.partition.udpGroup**, if set, as the value for this attribute. Using the **-u** command line switch when starting JBoss Application Server sets that value. See Section 25.6.2, "Isolating JGroups Channels" for information about using this configuration attribute to ensure that JGroups channels are properly isolated from one another. If this attribute is omitted, the default value is **228.11.11.11**.

- **mcast_port** specifies the port to use for multicast communication with the group. See Section 25.6.2, "Isolating JGroups Channels" for how to use this configuration attribute to ensure JGroups channels are properly isolated from one another. If this attribute is omitted, the default is **45688**.

- **mcast_send_buf_size**, **mcast_recv_buf_size**, **ucast_send_buf_size** and **ucast_recv_buf_size** define the socket send and receive buffer sizes that JGroups will request from the operating system. A large buffer size helps to ensure that packets are not dropped due to buffer overflow. However, socket buffer sizes are limited at the operating system level, so obtaining the desired buffer may require configuration at the operating system level. See Section 25.6.2.3, "Improving UDP Performance by Configuring OS UDP Buffer Limits" for further details.

- **bind_port** specifies the port to which the unicast receive socket should be bound. The default is **0**; i.e. use an ephemeral port.

- **port_range** specifies the number of ports to try if the port identified by **bind_port** is not available. The default is **1**, which specifies that only **bind_port** will be tried.

- **ip_ttl** specifies time-to-live (TTL) 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.

- **tos** specifies the traffic class for sending unicast and multicast datagrams.

The attributes that are common to all transport protocols, and thus have the same meanings when used with **TCP** or **TUNNEL**, are:

- **singleton_name** provides a unique name for this transport protocol configuration. Used by the application server's **ChannelFactory** to support sharing of a transport protocol instance by different channels that use the same transport protocol configuration. See Section 18.1.2, "The JGroups Shared Transport".

- **bind_addr** specifies the interface on which to receive and send messages. By default, JGroups uses the value of system property **jgroups.bind_addr**. This can also be set with the **-b** command line switch. See Section 25.6, "Other Configuration Issues" for more on binding JGroups sockets.

- **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 sends UDP packets via all available network interface controllers, if your machine has multiple network interface controllers available. This means that the same multicast message is sent N times, so use with care.

- **receive_interfaces** specifies a list of of interfaces on which to receive multicasts. The multicast receive socket will listen on all of these interfaces. This is a comma-separated list of IP addresses or interface names, for example, `192.168.5.1,eth1,127.0.0.1`.

- **send_interfaces** specifies a list of of interfaces via which to send multicasts. The multicast sender socket will send on all of these interfaces. This is a comma-separated list of IP addresses or interface names, for example, `192.168.5.1,eth1,127.0.0.1`.This means that the same multicast message is sent N times, so use with care.

- **enable_bundling** specifies whether to enable message bundling. If `true`, the tranpsort protocol queues outgoing messages until `max_bundle_size` bytes have accumulated, or `max_bundle_time` milliseconds have elapsed, whichever occurs first. Then the transport protocol bundles queued messages into one large message and sends it. The messages are unbundled at the receiver. The default is `false`.

  Message bundling can have significant performance benefits for channels that are used for high volume sending of messages where the sender does not block waiting for a response from recipients (for example, a JBoss Cache instance configured for **REPL_ASYNC**.) It can add considerable latency to applications where senders need to block waiting for responses, so it is not recommended for certain situations, such as where a JBoss Cache instance is configured for **REPL_SYNC**.

- **loopback** specifies whether the thread sending a message to the group should itself carry the message back up the stack for delivery. (Messages sent to the group are always delivered to the sending node as well.) If `false`, the sending thread does not carry the message; the transport protocol waits to read the message off the network and uses one of the message delivery pool threads for delivery. The default is `false`, but `true` is recommended to ensure that the channel receives its own messages, in case the network interface goes down.

- **discard_incompatible_packets** specifies whether to discard packets sent by peers that use a different version of JGroups. Each message in the cluster is tagged with a JGroups version. If `discard_incompatible_packets` is set to `true`, messages received from different versions of JGroups will be silently discarded. Otherwise, a warning will be logged. *In no case will the message be delivered.* The default value is `false`.

- **enable_diagnostics** specifies that the transport should open a multicast socket on address `diagnostics_addr` and port `diagnostics_port` to listen for diagnostic requests sent by the JGroups Probe utility.

- The various **thread_pool** attributes configure the behavior of the pool of threads JGroups uses to carry ordinary incoming messages up the stack. The various attributes provide the constructor arguments for an instance of `java.util.concurrent.ThreadPoolExecutorService`. In the example above, the pool will have a minimum or *core size* of 8 threads, and a maximum size of 200. If more than 8 pool threads have been created, a thread returning from carrying a message will wait for up to 5000 milliseconds to be assigned a new message to carry, after which it will terminate. If no threads

are available to carry a message, the (separate) thread reading messages off the socket will place messages in a queue; the queue will hold up to 1000 messages. If the queue is full, the thread reading messages off the socket will discard the message.

- The various **oob_thread_pool** attributes are similar to the **thread_pool** attributes in that they configure a **java.util.concurrent.ThreadPoolExecutorService** used to carry incoming messages up the protocol stack. In this case, the pool is used to carry a special type of message known as an Out-Of-Band (OOB) message. OOB messages are exempt from the ordered-delivery requirements of protocols like NAKACK and UNICAST and thus can be delivered up the stack even if NAKACK or UNICAST are queueing up messages from a particular sender. OOB messages are often used internally by JGroups protocols and can be used by applications as well. For example, when JBoss Cache is in **REPL_SYNC** mode, it uses OOB messages for the second phase of its two-phase-commit protocol.

25.1.2.2. 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 singleton_name="tcp"
        start_port="7800" end_port="7800"/>
```

The following attributes are specific to the **TCP** element:

- **start_port** and **end_port** define the range of TCP ports to which the server should bind. The server socket is bound to the first available port beginning with **start_port**. If no available port is found (for example, because the ports are in use by other sockets) before the **end_port**, the server throws an exception. If no **end_port** is provided, or **end_port** is lower than **start_port**, no upper limit is applied to the port range. If **start_port** is equal to **end_port**, JGroups is forced to use the specified port, since **start_port** fails if the specified port in not available. The default value is **7800**. If set to **0**, the operating system will select a port. (This will only work for **MPING** or **TCPGOSSIP** discovery protocols. **TCCPING** requires that nodes and their required ports are listed.)

- **bind_port** in TCP acts as an alias for **start_port**. If configured internally, it sets **start_port**.

- **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.

NOTE

All of the attributes common to all protocols discussed in the UDP protocol section also apply to TCP.

25.1.2.3. TUNNEL configuration

The **TUNNEL** protocol uses an external router process to send messages. The external router is a Java process that runs the **org.jgroups.stack.GossipRouter** main class. 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 set up communication with nodes behind firewalls. A node can establish a TCP connection to the **GossipRouter** through the firewall (you can use port 80). This connection is also 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 within the JGroups **<config>** element, like so:

```
<TUNNEL  singleton_name="tunnel"
            router_port="12001"
            router_host="192.168.5.1"/>
```

The available attributes in the **TUNNEL** element are listed below.

- **router_host** specifies the host on which the GossipRouter is running.

- **router_port** specifies the port on which the GossipRouter is listening.

- **reconnect_interval** specifies the interval of time (in milliseconds) for which **TUNNEL** will attempt to connect to the **GossipRouter** if the connection is not established. The default value is **5000**.

NOTE

All of the attributes common to all protocols discussed in the UDP protocol section also apply to **TUNNEL**.

25.1.3. Discovery Protocols

When a channel on a node first connects, it must determine which other nodes are running compatible channels, and which of these nodes is currently acting as the *coordinator* (the node responsible for letting new nodes join the group). Discovery protocols are used to find active nodes in the cluster and to determine which is the coordinator. This information is then provided to the group membership protocol (GMS), which communicates with the coordinator's GMS to add the newly-connecting node to the group. (For more information about group membership protocols, see Section 25.1.6, “Group Membership (GMS)”.)

Discovery protocols also assist merge protocols (see Section 25.5, “Merging (MERGE2)”) to detect cluster-split situations.

The discovery protocols sit on top of the transport protocol, so you can choose to use different discovery protocols depending on your transport protocol. These are also configured as sub-elements in the JGroups **<config>** element.

25.1.3.1. 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="3"/>
```

Here is another example PING configuration for contacting a Gossip Router.

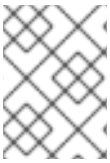
```
<PING gossip_host="localhost"
      gossip_port="1234"
       timeout="2000"
      num_initial_members="3"/>
```

The available attributes in the **PING** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **gossip_host** specifies the host on which the GossipRouter is running.
- **gossip_port** specifies the port on which the GossipRouter is listening on.
- **gossip_refresh** specifies the interval (in milliseconds) for the lease from the GossipRouter. The default is 20000.
- **initial_hosts** is a comma-separated list of addresses or ports (for example, **host1[12345],host2[23456]**) which are pinged for discovery. Default is **null**, meaning multicast discovery should be used. If **initial_hosts** is specified, you must list all possible

cluster members, not just a few well-known hosts, or **MERGE2** cluster split discovery will not work reliably.

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.

25.1.3.2. 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"
       num_initial_members="3"
       initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"/>
```

The available attributes in the **TCPGOSSIP** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-separated list of addresses/ports (for example, **host1[12345],host2[23456]**) of **GossipRouter**s to register

25.1.3.3. TCPPING

The TCPPING protocol takes a set of known members and pings 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"
     num_initial_members="3"/
     initial_hosts="hosta[2300],hostb[3400],hostc[4500]"
     port_range="3">
```

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 (for example, **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]**, and **hostc[4502]**. This configuration option allows for multiple possible ports on the same host to be pinged without having to spell out all possible combinations. If in your TCP protocol configuration your **end_port** is greater than your **start_port**, we recommend using a TCPPING **port_range** equal to the difference, to ensure a node is pinged no matter which port it is bound to within the allowed range.

25.1.3.4. MPING

MPING uses IP multicast to discover the initial membership. Unlike the other discovery protocols, which delegate the sending and receiving of discovery messages on the network to the transport protocol, **MPING** opens its own sockets to send and receive multicast discovery messages. As a result it can be used with all transports, but it is most often used with **TCP**. **TCP** usually requires **TCPPING**, which must explicitly list all possible group members. **MPING** does not have this requirement, and is typically used where **TCP** is required for regular message transport, and UDP multicasting is allowed for discovery.

```
<MPING timeout="2000"
    num_initial_members="3"
    bind_to_all_interfaces="true"
    mcast_addr="228.8.8.8"
    mcast_port="7500"
    ip_ttl="8"/>
```

The available attributes in the **MPING** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default 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. By default JGroups uses the value of the system property **jgroups.bind_addr**, which can be set with the **-b** command line switch. See Section 25.6, “Other Configuration Issues” for more on binding JGroups sockets.
- **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.

25.1.4. Failure Detection Protocols

The failure detection protocols are used to detect failed nodes. Once a failed node is detected, a *suspect verification* phase can occur. If the node is still considered dead after this phase is complete, the cluster updates its membership view so that further messages are not sent to the failed node. The service using JGroups is informed that the node is no longer part of the cluster. Failure detection protocols are configured as sub-elements in the JGroups **<config>** element.

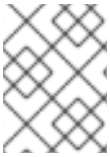
25.1.4.1. FD

FD is a failure detection protocol based on 'heartbeat' messages. This protocol requires that each node periodically ping its neighbour. If the neighbour fails to respond, the calling node sends a **SUSPECT** message to the cluster. The current group coordinator can optionally verify that the suspected node is dead (**VERIFY_SUSPECT**). If the node is still considered dead after this verification step, the coordinator updates the cluster's membership view. The following is an example of **FD** configuration:

```
<FD timeout="6000"
    max_tries="5"
    shun="true"/>
```

The available attributes in the **FD** element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for 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 forbidden from sending messages to the group without formally rejoining. A shunned node would need to rejoin the cluster via the discovery process. JGroups allows applications to configure a channel such that, when a channel is shunned, the process of rejoining the cluster and transferring state takes place automatically. (This is default behavior for JBoss Application Server.)



NOTE

Regular traffic from a node is proof of life, so heartbeat messages are only sent when no regular traffic is detected on the node for a long period of time.

25.1.4.2. 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, with the final member connecting to the first, forming a ring. Node B becomes suspected when its neighbour, Node A, detects an abnormally closed TCP socket, presumably due to a crash in Node B. (When nodes intend to leave the group, they inform their neighbours so that they do not become suspected.)

The simplest **FD_SOCK** configuration does not take any attribute. You can declare an empty **FD_SOCK** element in the JGroups **<config>** element.

```
<FD_SOCK/>
```

The attributes available to the **FD_SOCK** element are listed below.

- **bind_addr** specifies the interface to which the server socket should be bound. By default, JGroups uses the value of the system property **jgroups.bind_addr**. This system property can be set with the **-b** command line switch. For more information about binding JGroups sockets, see Section 25.6, “Other Configuration Issues”.

25.1.4.3. 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"/>
```

The available attributes in the **VERIFY_SUSPECT** element are listed below.

- **timeout** specifies how long to wait for a response from the suspected member before considering it dead.

25.1.4.4. 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).

A failure detection layer is intended to report real failures promptly, while avoiding 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/>
<FD timeout="6000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500"/>
```

In this example, a member becomes suspected when the neighbouring socket has been closed abnormally, in a process crash, for instance, since the operating system closes all sockets. However, if a host or switch crashes, the sockets will not be closed. **FD** will suspect the neighbour after sixty seconds (**6000** milliseconds). Note that if this example system were stopped in a breakpoint in the debugger, the node being debugged will be suspected once the **timeout** has elapsed.

A combination of **FD** and **FD_SOCK** provides a solid failure detection layer, which is why this technique is used across the JGroups configurations included with JBoss Application Server.

25.1.5. Reliable Delivery Protocols

Reliable delivery protocols within the JGroups stack ensure that messages are actually delivered, and delivered in the correct order (First In, First Out, or FIFO) to the destination node. The basis for reliable message delivery is positive and negative delivery acknowledgments (ACK and NAK). In **ACK** mode, the sender resends the message until acknowledgment is received from the receiver. In **NAK** mode, the receiver requests retransmission when it discovers a gap.

25.1.5.1. UNICAST

The **UNICAST** protocol is used for unicast messages. It uses positive acknowlegements (**ACK**). It is configured as a sub-element under the JGroups **config** element. If the JGroups stack is configured with the 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="300,600,1200,2400,3600"/>
```

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 has not received an **ACK** after 100 milliseconds the first time, and the second time it waits for 200 milliseconds before resending, and so on. A low value for the first timeout allows for prompt retransmission of dropped messages, but means that messages may be transmitted more than once if they have not actually been lost (that is, the message has been sent, but the **ACK** has not been received before the timeout). High values (**1000, 2000, 3000**) can improve performance if the network is tuned such that UDP datagram loss is infrequent. High values on networks with frequent losses will be harmful to performance, since later messages will not be delivered until lost messages have been retransmitted.

25.1.5.2. NAKACK

The **NAKACK** protocol is used for multicast messages. It uses negative acknowlegements (**NAK**). Under this protocol, each message is tagged with a sequence number. The receiver keeps track of the received sequence numbers and delivers the messages in order. When a gap in the series of received sequence numbers is detected, the receiver schedules a task to periodically ask the sender to retransmit the missing message. The task is cancelled if the missing message is received. **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"/>
```

The configurable attributes in the **pbcast.NAKACK** element are as follows.

- **retransmit_timeout** specifies the series of timeouts (in milliseconds) after which retransmission is requested if a missing message has not yet been received.
- **use_mcast_xmit** determines whether the sender should send the retransmission to the entire cluster rather than just to the node requesting it. This is useful when the *sender*'s network layer tends to drop packets, avoiding the need to individually retransmit to each node.
- **max_xmit_size** specifies the maximum size (in bytes) for a bundled retransmission, if multiple messages are reported missing.
- **discard_delivered_msgs** specifies whether to discard delivered messages on the receiver nodes. By default, nodes save delivered messages so any node can retransmit a lost message in case the original sender has crashed or left the group. However, if we only ask the sender to resend its messages, we can enable this option and discard delivered messages.
- **gc_lag** specifies the number of messages to keep in memory for retransmission, even after the periodic cleanup protocol (see Section 25.4, “Distributed Garbage Collection (STABLE)”) indicates all peers have received the message. The default value is **20**.

25.1.6. 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 any interested services like JBoss Cache or HAPartition, 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"
    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 number of milliseconds to wait after a failed JOIN before trying again.
- **print_local_addr** specifies whether to dump the node's own address to the standard output when started.
- **shun** specifies whether a node should shun (that is, disconnect) itself if it receives a cluster view in which it is not a member node.
- **disable_initial_coord** specifies whether to prevent this node from becoming the cluster

coordinator during the initial connection of the channel. This flag does not prevent a node becoming the coordinator after the initial channel connection, if the current coordinator leaves the group.

- **view_bundling** specifies whether multiple JOIN or LEAVE requests arriving at the same time are bundled and handled together at the same time, resulting in only one new view that incorporates all changes. This is is more efficient than handling each request separately.

25.1.7. 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 out-of-memory conditions or dropped packets that have to be retransmitted. In JGroups, 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 send 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="2000000"
    min_threshold="0.10"
    ignore_synchronous_response="true"/>
```

The configurable attributes in the **FC** element are as follows.

- **max_credits** specifies the maximum number of credits (in bytes). This value should be smaller than the JVM heap size.
- **min_credits** specifies the minimum number of bytes that must be received before the receiver will send more credits to the sender.
- **min_threshold** specifies the percentage of the **`max_credits`** that should be used to calculate **`min_credits`**. Setting this overrides the **`min_credits`** attribute.
- **ignore_synchronous_response** specifies whether threads that have carried messages up to the application should be allowed to carry outgoing messages back down through FC without blocking for credits. *Synchronous response* refers to the fact that these messages are generally responses to incoming RPC-type messages. Forbidding JGroups threads to carry messages up to block in FC can help prevent certain deadlock scenarios, so we recommend setting this to **`true`**.

NOTE

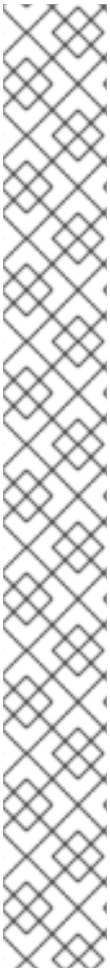
FC is required for group communication where group messages must be sent at the highest speed that the slowest receiver can handle. For example, say we have a cluster comprised of nodes A, B, C and D. D is slow (perhaps overloaded), while the rest are fast. When A sends a group message, it does so via TCP connections: A-A (theoretically), A-B, A-C and A-D.

Say A sends 100 million messages to the cluster. TCP's flow control applies to A-B, A-C and A-D individually, but not to A-BCD as a group. Therefore, A, B and C will receive the 100 million messages, but D will receive only 1 million. (This is also why **NAKACK** is required, even though TCP handles its own retransmission.)

JGroups must buffer all messages in memory in case an original sender *S* dies and a node requests retransmission of a message sent by *S*. Since all members buffer all messages that they receive, stable messages (messages seen by every node) must sometimes be purged. (The purging process is managed by the **STABLE** protocol. For more information, see Section 25.4, “Distributed Garbage Collection (STABLE)” .)

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, even with TCP we need to FC to ensure we send messages at a rate the slowest receiver (D) can handle.



NOTE

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 uses JGroups to make synchronous group RPC calls. 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.)

25.2. FRAGMENTATION (FRAG2)

This protocol fragments messages that are larger than a certain size, and reassembles them at the receiver's side. It works for both unicast and multicast messages. It is configured with the **FRAG2** sub-element in the JGroups **config** element. Here is an example configuration:

```
<FRAG2 frag_size="60000"/>
```

The configurable attributes in the FRAG2 element are as follows.

- **frag_size** specifies the maximum message size (in bytes) before fragmentation occurs. Messages larger than this size are fragmented. For stacks that use the UDP transport, this value must be lower than 64 kilobytes (the maximum UDP datagram size). For TCP-based stacks, it must be lower than the value of **max_credits** in the FC protocol.

NOTE

TCP protocol already provides fragmentation, but a JGroups fragmentation protocol is still required if FC is used. The reason for this is that if you send a message larger than **FC.max_credits**, the FC protocol will block forever. So, **frag_size** within FRAG2 must always be set to a value lower than that of **FC.max_credits**.

25.3. 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/>
```

25.4. DISTRIBUTED GARBAGE COLLECTION (STABLE)

In a JGroups cluster, all nodes must store all messages received for potential retransmission in case of a failure. However, if we store all messages forever, we will run out of memory. The distributed garbage collection service periodically purges messages that have been seen by all nodes, removing them 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"
    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. Set this to **0** to disable interval-based garbage collection.

- **max_bytes** specifies the maximum number of bytes received before the cluster triggers a garbage collection run. Set to **0** to disable garbage collection based on the bytes received.

- **stability_delay** specifies the maximum time period (in milliseconds) of a random delay

introduced before a node sends its **STABILITY** message at the end of a garbage collection run. The delay gives other nodes concurrently running a **STABLE** task a chance to send first. If used together with **max_bytes**, this attribute should be set to a small number.

NOTE

Set the **max_bytes** attribute when you have a high traffic cluster.

25.5. MERGING (MERGE2)

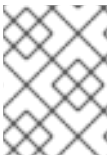
When a network error occurs, the cluster might be partitioned into several different partitions. JGroups has a MERGE service that allows the coordinators in partitions to communicate with each other and form a single cluster back again. The merging service is configured in the **MERGE2** sub-element under the JGroups **Config** element. Here is an example configuration.

```
<MERGE2 max_interval="10000"
    min_interval="2000"/>
```

The configurable attributes in the **MERGE2** element are as follows.

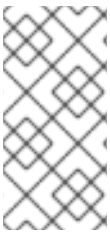
- **max_interval** specifies the maximum number of milliseconds to wait before sending a MERGE message.
- **min_interval** specifies the minimum number of milliseconds to wait before sending a MERGE message.

JGroups chooses a random value between **min_interval** and **max_interval** to periodically send the MERGE message.



NOTE

The application state maintained by the application using a channel is not merged by JGroups during a merge. This must be done by the application.



NOTE

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 may not detect all sub-groups, and may miss those comprised solely of unlisted members.

25.6. OTHER CONFIGURATION ISSUES

25.6.1. 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 is 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 the system property **jgroups.bind_addr** (or a deprecated earlier name for the same thing, **bind.address**) has been set. The system property

has a higher priority level than the XML property. If JBoss Application Server is started with the **-b** (or **--host**) switch, the application server will set **jgroups.bind_addr** to the specified value. If **-b** is not set, the application server will bind most services to **localhost** by default.

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 various **bind_addr** attributes to the desired interfaces.

25.6.2. Isolating JGroups Channels

Within JBoss Application Server, there are a number of services that independently create JGroups channels — possibly multiple different JBoss Cache services (used for **HttpSession** replication, EJB3 stateful session bean replication and EJB3 entity replication), two JBoss Messaging channels, and **HAPartition**, the general purpose clustering service 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 and, for UDP-based channels, its multicast address and port. Isolating a JGroups channel means ensuring that different

channels use different values for the group name, the multicast address and, in some cases, the multicast port.

25.6.2.1. Isolating sets of Application Server instances from each other

This section addresses the issue of having multiple independent clusters running within the same environment. For example, you might have a production cluster, a staging cluster, and a QA cluster, or multiple clusters in a QA test lab or development team environment.

To isolate JGroups clusters from other clusters on the network, you must:

- Make sure the channels in the various clusters use different group names. This can be controlled with the command line arguments used to start JBoss; see Section 25.6.2.2.1, "Changing the Group Name" for more information.
- Make sure the channels in the various clusters use different multicast addresses. This is also easy to control with the command line arguments used to start JBoss.
- If you are not running on Linux, Windows, Solaris or HP-UX, you may also need to ensure that the channels in each cluster use different multicast ports. This is more difficult than using different group names, although it can still be controlled from the command line. See Section 25.6.2.2.3, "Changing the Multicast Port" . Note that using different ports should not be necessary if your servers are running on Linux, Windows, Solaris or HP-UX.

25.6.2.2. Isolating Channels for Different Services on the Same Set of AS Instances

This section addresses the usual case: a cluster of three machines, each of which has, for example, an HAPartition deployed alongside JBoss Cache for web session clustering. The HAPartition channels should not communicate with the JBoss Cache channels. Ensuring proper isolation of these channels is straightforward, and is usually handled by the application server without any alterations on the part of the user.

To isolate channels for different services from each other on the same set of application server instances, each channel must have its own group name. The configurations that ship with JBoss Application Server ensure that this is the case. However, if you create a custom service that uses JGroups directly, you must use a unique group name. If you create a custom JBoss Cache configuration, ensure that you provide a unique value in the **`clusterName`** configuration property.

In releases prior to JBoss Application Server 5, different channels running in the same application server also had to use unique multicast ports. With the JGroups shared transport introduced in JBoss AS 5 (see Section 18.1.2, "The JGroups Shared Transport"), it is now common for multiple channels to use the same tranpsort protocol and its sockets. This makes configuration easier, which is one of the main benefits of the shared transport. However, if you decide to create your own custom JGroups protocol stack configuration, be sure to configure its transport protocols with a multicast port that is different from the ports used in other protocol stacks.

25.6.2.2.1. Changing the Group Name

The group name for a JGroups channel is configured via the service that starts the channel. For all the standard clustered services, we make it easy for you to create unique groups names by simply using the **`-g`** (or **`--partition`**) switch when starting 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,

```
<property name="clusterName">${jboss.partition.name:DefaultPartition}-
SFSBCache</property>
```

25.6.2.2.2. Changing the multicast address and port

The **-u** (or **--udp**) command line switch may be used to control the multicast address used by the JGroups channels opened by all standard AS services.

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the **jboss.partition.udpGroup** system property, which is referenced in all of the standard protocol stack configurations in JBoss AS:

```
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}" ....
```

NOTE

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.

25.6.2.2.3. Changing the Multicast Port

On some operating systems (Mac OS X for example), using different **-g** and **-u** values is not sufficient to isolate clusters; the channels running in the different clusters must also use different multicast ports. Unfortunately, setting the multicast ports is not as simple as **-g** and **-u**. By default, a JBoss AS instance running the **all** configuration will use up to two different instances of the JGroups UDP transport protocol, and will therefore open two multicast sockets. You can control the ports those sockets use by using system properties on the command line. For example,

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all \\
        -Djboss.jgroups.udp.mcast_port=12345 -
Djboss.messaging.datachanneludpport=23456
```

The **jboss.messaging.datachanneludpport** property controls the multicast port used by the **MPING** protocol in JBoss Messaging's **DATA** channel. The **jboss.jgroups.udp.mcast_port** property controls the multicast port used by the UDP transport protocol shared by all other clustered services.

The set of JGroups protocol stack configurations included in the **$JBOSS_HOME/server/all/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** file includes a number of other example protocol stack configurations that the standard JBoss AS distribution doesn't actually use. Those configurations also use system properties to set any multicast ports. So, if you reconfigure some AS service to use one of those protocol stack configurations, use the appropriate system property to control the port from the command line.

NOTE

It should be sufficient to just change the address, but unfortunately the handling of multicast sockets is one area where the JVM fails to hide operating system behavior differences from the application. The `java.net.MulticastSocket` class provides different overloaded constructors. On some operating systems, if you use one constructor variant, packets addressed to a particular multicast port are delivered to all listeners on that port, regardless of the multicast address on which they are listening. We refer to this as the *promiscuous traffic* problem. On most operating systems that exhibit the promiscuous traffic problem (Linux, Solaris and HP-UX) JGroups can use a different constructor variant that avoids the problem. However, on some operating systems with the promiscuous traffic problem (Mac OS X), multicast does not work properly if the other constructor variant is used. So, on these operating systems the recommendation is to configure different multicast ports for different clusters.

25.6.2.3. Improving UDP Performance by Configuring OS UDP Buffer Limits

By default, the JGroups channels in JBoss Enterprise Application Platform use the UDP transport protocol to take advantage of IP multicast. However, one disadvantage of UDP is it does not come with the reliable delivery guarantees provided by TCP. The protocols discussed in Section 25.1.5, “Reliable Delivery Protocols” allow JGroups to guarantee delivery of UDP messages, but those protocols are implemented in Java, not at the operating system network layer. For peak performance from a UDP-based JGroups channel it is important to limit the need for JGroups to retransmit messages by limiting UDP datagram loss.

One of the most common causes of lost UDP datagrams is an undersized receive buffer on the socket. The UDP protocol's `mcast_recv_buf_size` and `ucast_recv_buf_size` configuration attributes are used to specify the amount of receive buffer JGroups *requests* from the operating system, but the actual size of the buffer the operating system provides is limited by operating system-level maximums. These maximums are often very low:

Table 25.1. Default Max UDP Buffer Sizes

Operating System	Default Max UDP Buffer (in bytes)
Linux	131071
Windows	No known limit
Solaris	262144
FreeBSD, Darwin	262144
AIX	1048576

The command used to increase the above limits is operating system-specific. The table below shows the command required to increase the maximum buffer to 25 megabytes. In all cases, root privileges are required:

Table 25.2. Commands to Change Max UDP Buffer Sizes

Operating System	Command
Linux	**sysctl -w net.core.rmem_max=26214400**
Solaris	**ndd -set /dev/udp udp_max_buf 26214400**
FreeBSD, Darwin	**sysctl -w kern.ipc.maxsockbuf=26214400**
AIX	**no -o sb_max=8388608** (AIX will only allow 1 megabyte, 4 megabytes or 8 megabytes).

25.6.3. JGroups Troubleshooting

25.6.3.1. 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:

```
[lib]$ java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -
mcast_addr 224.10.10.10 -port 5555
```

Then in another window start **McastSenderTest**:

```
[lib]$ 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.

25.6.3.2. 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.

CHAPTER 26. 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 the Red Hat Documentation portal.

See also Section 18.2, “Distributed Caching with JBoss Cache” for information on how the standard JBoss Enterprise Application Platform clustered services use JBoss Cache.

26.1. 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.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/.

Most JBoss Cache configuration examples in this section use the JBoss Microcontainer schema for building up an **`org.jboss.cache.config.Configuration`** object graph from XML. JBoss Cache has its own custom XML schema, but the standard JBoss Enterprise 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.

26.1.1. Editing the CacheManager Configuration

As discussed in Section 18.2.1, “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 26.2.1, “Deployment Via the CacheManager Service”.

The CacheManager is configured via the **`$JBOSS_HOME/server/$PROFILE/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>

   <property name="useLockStriping">false</property>

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

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

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

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

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

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

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

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

         <!-- Do not change these -->
         <property name="autoDataGravitation">false</property>
         <property name="dataGravitationRemoveOnFind">true</property>
         <property name="dataGravitationSearchBackupTrees">true</property>

         <property name="buddyLocatorConfig">
            <bean
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
               <!-- The number of backup nodes we maintain -->
               <property name="numBuddies">1</property>
               <!-- Means that each node will *try* to select a buddy on
                    a different physical host. If not able to do so
```

```
                    though, it will fall back to colocated nodes. -->
               <property name="ignoreColocatedBuddies">true</property>
            </bean>
         </property>
      </bean>
   </property>
   <property name="cacheLoaderConfig">
      <bean class="org.jboss.cache.config.CacheLoaderConfig">
            <!-- Do not change these -->
            <property name="passivation">true</property>
            <property name="shared">false</property>

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

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

Basically, the XML specifies the creation of an **`org.jboss.cache.config.Configuration`** java bean and the setting of a number of properties on that bean. Most of the properties are of simple types,

but some, such as **buddyReplicationConfig** and **cacheLoaderConfig** take various types java beans as their values.

Next we'll look at some of the key configuration options.

26.1.2. 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 two-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.

26.1.3. 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.

26.1.4. 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-version 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.

26.1.5. 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 26.2.1, "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 26.2.3, "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 26.2.2, "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>
```

26.1.6. 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 18.2.1, "The JBoss Enterprise Application Platform CacheManager Service"). The EJB container will configure eviction itself using the values included in each bean's configuration.

26.1.7. 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.

26.1.7.1. 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">${jboss.server.data.dir}${/}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.

26.1.8. 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.

26.2. 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.

26.2.1. 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 18.2.1, "The JBoss Enterprise Application Platform CacheManager Service"). End user applications can do the same thing; here's how.

Section 26.1.1, "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 26.1.1, "Editing the CacheManager Configuration" for an example configuration.

26.2.1.1. 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");
   }
}
```

26.2.2. Deployment Via a `-service.xml` File

As in JBoss Enterprise Application Platform 4.x, you can also deploy a JBoss Cache instance as an MBean service via a **`-service.xml`** file. The primary difference from JBoss Enterprise Application Platform 4.x is the value of the **code** attribute in the **`mbean`** element. In JBoss Enterprise Application Platform 4.x, this was **`org.jboss.cache.TreeCache`**; in JBoss Enterprise Application Platform 5.x 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.CacfheJmxWrapper"
         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.

26.2.3. 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 26.1.1, "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=ExampleCacheJmxWrapper",

exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
                  registerDirectly=true)
```

```
      </annotation>

      <property name="configuration"><inject bean="ExampleCacheConfig"/>
</property>

   </bean>

   <bean name="ExampleService" class="org.foo.ExampleService">
      <property name="cache"><inject bean="ExampleCacheJmxWrapper"
property="cache"/></property>
   </bean>

</deployment>
```

Here the "ExampleCacheJmxWrapper" bean handles the task of creating the cache from the configuration. **`CacheJmxWrapper`** is a JBoss Cache class that provides an MBean interface for a cache. Adding an <annotation> element binds the JBoss Microcontainer **`@JMX`** annotation to the bean; that in turn results in JBoss Enterprise 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. APPENDICES

APPENDIX A. VENDOR-SPECIFIC DATASOURCE DEFINITIONS

This appendix includes datasource definitions for databases supported by JBoss Enterprise Application Platform.

A.1. DEPLOYER LOCATION AND NAMING

All database deployers should be saved to the ***$JBOSS_HOME*/server/default/deploy/oracle-ds.xml** directory on the server. Each deployer file needs to end with the suffix **-ds.xml**. For instance, an Oracle datasource deployer might be named **oracle-ds.xml**. If files are not named properly, the are not found by the server.

A.2. DB2

Example A.1. DB2 Local-XA

Copy the ***$db2_install_dir*/java/db2jcc.jar** and ***$db2_install_dir*/java/db2jcc_license_cu.jar** files into the ***$jboss_install_dir*/server/default/lib** directory. The **db2java.zip** file, which is part of the legacy CLI driver, is normally not required when using the DB2 Universal JDBC driver included in DB2 v8.1 and later.

```
<datasources>

  <local-tx-datasource>
    <jndi-name>DB2DS</jndi-name>
    <!-- Use the syntax 'jdbc:db2:yourdatabase' for jdbc type 2
connection -->
    <!-- Use the syntax 'jdbc:db2://serveraddress:port/yourdatabase' for
jdbc type 4 connection -->
    <connection-
url>jdbc:db2://serveraddress:port/yourdatabase</connection-url>
    <driver-class>com.ibm.db2.jcc.DB2Driver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

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

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.2. DB2 XA

Copy the ***$db2_install_dir*/java/db2jcc.jar** and ***$db2_install_dir*/java/db2jcc_license_cu.jar** files into the ***$jboss_install_dir*/server/default/lib** directory.

The **db2java.zip** file is required when using the DB2 Universal JDBC driver (type 4) for XA on DB2 v8.1 fixpak 14 (and the corresponding DB2 v8.2 fixpak 7).

```
<datasources>
  <!--
      XADatasource for DB2 v8.x (app driver)
  -->

  <xa-datasource>
    <jndi-name>DB2XADS</jndi-name>

    <xa-datasource-class>com.ibm.db2.jcc.DB2XADataSource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">your_server_address</xa-
datasource-property>
    <xa-datasource-property name="PortNumber">your_server_port</xa-
datasource-property>
    <xa-datasource-property name="DatabaseName">your_database_name</xa-
datasource-property>
    <!-- DriverType can be either 2 or 4, but you most likely want to
use the JDBC type 4 as it doesn't require a DB" client -->
    <xa-datasource-property name="DriverType">4</xa-datasource-
property>
    <!-- If driverType 4 is used, the following two tags are needed -->
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>

    <xa-datasource-property name="User">your_user</xa-datasource-
property>
    <xa-datasource-property name="Password">your_password</xa-
datasource-property>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2</type-mapping>
    </metadata>
  </xa-datasource>

</datasources>
```

Example A.3. DB2 on AS/400

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

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

<!-- $Id: db2-400-ds.xml,v 1.1.4.2 2004/10/27 18:44:10 pilhuhn Exp $ -->

<!-- You need the jt400.jar that is delivered with IBM iSeries Access or
the
OpenSource Project jtopen.

[systemname] Hostame of the iSeries
[schema]     Default schema is needed so jboss could use metadat to
test if the tables exists
-->

<datasources>
  <local-tx-datasource>
    <jndi-name>DB2-400</jndi-name>
    <connection-url>jdbc:as400://[systemname]/[schema];extended
dynamic=true;package=jbpkg;package cache=true;package
library=jboss;errors=full</connection-url>
    <driver-class>com.ibm.as400.access.AS400JDBCDriver</driver-class>
    <user-name>[username]</user-name>
    <password>[password]</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
  <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
    -->
    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2/400</type-mapping>
    </metadata>

  </local-tx-datasource>

</datasources>
```

Example A.4. DB2 on AS/400 "native"

The *Native* JDBC driver is shipped as part of the IBM Developer Kit for Java (57xxJV1). It is implemented by making native method calls to the SQL *CLI (Call Level Interface)*, and it only runs on the i5/OS JVM. The class name to register is **`com.ibm.db2.jdbc.app.DB2Driver`**. The URL subprotocol is **db2**. See JDBC FAQKS at http://www-03.ibm.com/systems/i/software/toolbox/faqjdbc.html#faqA1 for more information.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
========================================================================= --
>
<!--
-->
<!--  JBoss Server Configuration
-->
<!--
-->
<!--
========================================================================= --
>
<!-- $Id: db2-400-ds.xml,v 1.1.4.2 2004/10/27 18:44:10 pilhuhn Exp $
-->
<!-- You need the jt400.jar that is delivered with IBM iSeries Access or
the
OpenSource Project jtopen.
[systemname] Hostame of the iSeries
[schema]     Default schema is needed so jboss could use metadat to
test if the tables exists  -->
<datasources>
  <local-tx-datasource>
    <jndi-name>DB2-400</jndi-name>
    <connection-url>jdbc:db2://[systemname]/[schema];extended
dynamic=true;package=jbpkg;package cache=true;package
library=jboss;errors=full</connection-url>
    <driver-class>com.ibm.db2.jdbc.app.DB2Driver</driver-class>
    <user-name>[username]</user-name>
    <password>[password]</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>        --
>

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
  <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>        -->
    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>DB2/400</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Tips

- This driver is sensitive to the job's CCSID, but works fine with **`CCSID=37`**.

- **[systemname]** must be defined as entry **WRKRDBDIRE** like ***local**.

A.3. ORACLE

Example A.5. Oracle Local-TX Datasource

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

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

<!-- $Id: oracle-ds.xml,v 1.6 2004/09/15 14:37:40 loubyansky Exp $ -->
<!--
========================================================================= -->
<!--  Datasource config for Oracle originally from Steven Coy
-->
<!--
========================================================================= -->


<datasources>
  <local-tx-datasource>
    <jndi-name>OracleDS</jndi-name>
    <connection-
url>jdbc:oracle:thin:@youroraclehost:1521:yoursid</connection-url>
    <!--
 See on WIKI page below how to use Oracle's thin JDBC driver to connect
with enterprise RAC.
    -->
    <!--
 Here are a couple of the possible OCI configurations.
 For more information, see
http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/java.9
20/a96654/toc.htm

<connection-url>jdbc:oracle:oci:@youroracle-tns-name</connection-url>
or
<connection-url>jdbc:oracle:oci:@(description=(address=
(host=youroraclehost)(protocol=tcp)(port=1521))(connect_data=
(SERVICE_NAME=yourservicename)))</connection-url>

Clearly, its better to have TNS set up properly.
    -->
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
```

```
    <user-name>x</user-name>
    <password>y</password>

    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>

    <!-- Uses the pingDatabase method to check a connection is still
valid before handing it out from the pool -->
    <!--valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker
</valid-connection-checker-class-name-->
    <!-- Checks the Oracle error codes and messages for fatal errors -->
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</excep
tion-sorter-class-name>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

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

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>Oracle9i</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.6. Oracle XA Datasource

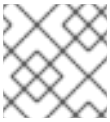
```
<?xml version="1.0" encoding="UTF-8"?>

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

<!-- $Id: oracle-xa-ds.xml,v 1.13 2004/09/15 14:37:40 loubyansky Exp $ -
->
```

```
<!--
======================================================================== --
>
<!-- ATTENTION:  DO NOT FORGET TO SET Pad=true IN transaction-
service.xml  -->
<!--
======================================================================== --
>

<datasources>
  <xa-datasource>
    <jndi-name>XAOracleDS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>
    <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-
datasource-class>
    <xa-datasource-property name="URL">jdbc:oracle:oci8:@tc</xa-
datasource-property>
    <xa-datasource-property name="User">scott</xa-datasource-property>
    <xa-datasource-property name="Password">tiger</xa-datasource-
property>
    <!-- Uses the pingDatabase method to check a connection is still
valid before handing it out from the pool -->
    <!--valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker
</valid-connection-checker-class-name-->
    <!-- Checks the Oracle error codes and messages for fatal errors -->
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</excep
tion-sorter-class-name>
    <!-- Oracles XA datasource cannot reuse a connection outside a
transaction once enlisted in a global transaction and vice-versa -->
    <no-tx-separate-pools></no-tx-separate-pools>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>Oracle9i</type-mapping>
    </metadata>
  </xa-datasource>

  <mbean
code="org.jboss.resource.adapter.jdbc.vendor.OracleXAExceptionFormatter"
         name="jboss.jca:service=OracleXAExceptionFormatter">
    <depends optional-attribute-
name="TransactionManagerService">jboss:service=TransactionManager</depen
ds>
  </mbean>

</datasources>
```

Example A.7. Oracle's Thin JDBC Driver with Enterprise RAC

The extra configuration to use Oracle's Thin JDBC driver to connect with Enterprise RAC involves the <connection-url>. The two hostnames provide load balancing and failover to the underlying physical database.

```
...
<connection-url>jdbc:oracle:thin:@(description=(address_list=
(load_balance=on)(failover=on)(address=(protocol=tcp)(host=xxxxhost1)
(port=1521))(address=(protocol=tcp)(host=xxxxhost2)(port=1521)))
(connect_data=(service_name=xxxxsid)(failover_mode=(type=select)
(method=basic))))</connection-url>
...
```



NOTE

This example has only been tested against Oracle 10g.

A.3.1. Changes in Oracle 10g JDBC Driver

It is no longer necessary to enable the **Pad** option in your **jboss-service.xml** file. Further, you no longer need the <no-tx-seperate-pool/>.

A.3.2. Type Mapping for Oracle 10g

You need to specify Oracle9i type mapping for Oracle 10g datasource configurations.

Example A.8. Oracle9i Type Mapping

```
....
<metadata>
  <type-mapping>Oracle9i</type-mapping>
</metadata>
....
```

A.3.3. Retrieving the Underlying Oracle Connection Object

Example A.9. Oracle Connection Object

```
 Connection conn = myJBossDatasource.getConnection();
WrappedConnection wrappedConn = (WrappedConnection)conn;
Connection underlyingConn = wrappedConn.getUnderlyingConnection();
OracleConnection oracleConn = (OracleConnection)underlyingConn;
```

A.4. SYBASE

Example A.10. Sybase Datasource

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

```
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/SybaseDB</jndi-name>
    <!-- Sybase jConnect URL for the database.
  NOTE: The hostname and port are made up values. The optional
  database name is provided, as well as some additinal Driver
  parameters.
    -->
    <connection-url>jdbc:sybase:Tds:host.at.some.domain:5000/db_name?
JCONNECT_VERSION=6</connection-url>
    <driver-class>com.sybase.jdbc2.jdbc.SybDataSource</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter</excep
tion-sorter-class-name>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

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

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

</datasources>
```

[1]

A.5. MICROSOFT SQL SERVER

To evaluate those drivers, you can use a simple JSP page to query the **pubs** database shipped with Microsoft SQL Server.

Move the WAR archive located in files/mssql-test.zip to the **/deploy**, start the server, and navigate your web browser to http://localhost:8080/test/test.jsp.

Example A.11. Local-TX Datasource Using DataDirect Driver

This example uses the DataDirect Connect for JDBC drivers from http://www.datadirect.com.

```
 <datasources>
  <local-tx-datasource>
    <jndi-name>MerliaDS</jndi-name>
    <connection-
```

```
url>jdbc:datadirect:sqlserver://localhost:1433;DatabaseName=jboss</conne
ction-url>
    <driver-class>com.ddtek.jdbc.sqlserver.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>sa</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.12. Local-TX Datasource Using Merlia Driver

This example uses the Merlia JDBC Driver drivers from http://www.inetsoftware.de.

```
 <datasources>
  <local-tx-datasource>
    <jndi-name>MerliaDS</jndi-name>
    <connection-url>jdbc:inetdae7:localhost:1433?
database=pubs</connection-url>
    <driver-class>com.inet.tds.TdsDataSource</driver-class>
    <user-name>sa</user-name>
    <password>sa</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.13. XA Datasource Using Merlia Driver

This example uses the Merlia JDBC Driver drivers from http://www.inetsoftware.de.

```
 <datasources>
  <xa-datasource>
    <jndi-name>MerliaXADS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>
    <xa-datasource-class>com.inet.tds.DTCDataSource</xa-datasource-
class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
```

```
property>
    <user-name>sa</user-name>
    <password>sa</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>

</datasources>
```

A.5.1. Microsoft JDBC Drivers

The Microsoft JDBC driver for MS SQL Server comes now in two flavors:

- SQL Server 2000 Driver for JDBC Service Pack 3 which can be used with SQL Server 2000
- Microsoft SQL Server 2005 JDBC Driver which be used with either SQL Server 2000 or 2005. This version contains numerous fixes and has been certified for JBoss Hibernate. This driver runs under JDK 5.

Make sure to read the `release.txt` included in the driver distribution to understand the differences between these drivers, especially the new package name introduced with 2005 and the potential conflicts when using both drivers in the same app server.

Example A.14. Microsoft SQL Server 2000 Local-TX Datasource

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

<datasources>
  <local-tx-datasource>
    <jndi-name>MSSQL2000DS</jndi-name>
    <connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;Databa
seName=pubs</connection-url>
    <driver-class>com.microsoft.jdbc.sqlserver.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.15. Microsoft SQL Server 2005 Local-TX Datasource

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


<datasources>
  <local-tx-datasource>
    <jndi-name>MSSQL2005DS</jndi-name>
    <connection-
url>jdbc:sqlserver://localhost:1433;DatabaseName=pubs</connection-url>
    <driver-class>com.microsoft.sqlserver.jdbc.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.16. Microsoft SQL Server 2005 XA Datasource

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

<datasources>
  <xa-datasource>
    <jndi-name>MSSQL2005XADS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>
    <xa-datasource-
class>com.microsoft.sqlserver.jdbc.SQLServerXADataSource</xa-datasource-
class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
    <xa-datasource-property name="SelectMethod">cursor</xa-datasource-
property>
    <xa-datasource-property name="User">sa</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-
property>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>

</datasources>
```

A.5.2. JSQL Drivers

Example A.17. JSQL Driver

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

<datasources>
  <local-tx-datasource>
    <jndi-name>JSQLDS</jndi-name>
    <connection-
url>jdbc:JSQLConnect://localhost:1433/databaseName=testdb</connection-
url>
    <driver-class>com.jnetdirect.jsql.JSQLDriver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

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

  </local-tx-datasource>

</datasources>
```

A.5.3. jTDS JDBC Driver

jTDS is an open source 100% pure Java (type 4) JDBC 3.0 driver for Microsoft SQL Server (6.5, 7, 2000 and 2005) and Sybase (10, 11, 12, 15). jTDS is based on FreeTDS and is currently the fastest production-ready JDBC driver for microsoft SQL Server and Sybase. jTDS is 100% JDBC 3.0 compatible, supporting forward-only and scrollable/updateable ResultSets, concurrent (completely independent) Statements and implementing all the **`DatabaseMetaData`** and **`ResultSetMetaData`** methods.

Download jTDS from http://jtds.sourceforge.net/.

Example A.18. jTDS Local-TX Datasource

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


<datasources>
  <local-tx-datasource>
    <jndi-name>jtdsDS</jndi-name>
    <connection-
url>jdbc:jtds:sqlserver://localhost:1433;databaseName=pubs</connection-
url>
    <driver-class>net.sourceforge.jtds.jdbc.Driver</driver-class>
```

```
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- optional parameters -->
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
    <min-pool-size>10</min-pool-size>
    <max-pool-size>30</max-pool-size>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <new-connection-sql>select 1</new-connection-sql>
    <check-valid-connection-sql>select 1</check-valid-connection-sql>
    <set-tx-query-timeout></set-tx-query-timeout>
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Example A.19. jTDS XA Datasource

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

<datasources>
  <xa-datasource>
    <jndi-name>jtdsXADS</jndi-name>
    <xa-datasource-class>net.sourceforge.jtds.jdbcx.JtdsDataSource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
    <xa-datasource-property name="User">sa</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-
property>

    <!--
  When set to true, emulate XA distributed transaction support. Set to
false to use experimental
  true distributed transaction support. True distributed transaction
support is only available for
  SQL Server 2000 and requires the installation of an external stored
procedure in the target server
  (see the README.XA file in the distribution for details).
    -->
    <xa-datasource-property name="XaEmulation">true</xa-datasource-
property>

    <track-connection-by-tx></track-connection-by-tx>

    <!-- optional parameters -->
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
    <min-pool-size>10</min-pool-size>
```

```
    <max-pool-size>30</max-pool-size>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <new-connection-sql>select 1</new-connection-sql>
    <check-valid-connection-sql>select 1</check-valid-connection-sql>
    <set-tx-query-timeout></set-tx-query-timeout>
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>

</datasources>
```

A.5.4. "Invalid object name 'JMS_SUBSCRIPTIONS' Exception

If you receive an exception like the one in Example A.20, "JMS_SUBSCRIPTIONS Exception" during startup, specify a **SelectMethod** in the connection URL, as shown in Example A.21, "Specifying a SelectMethod".

Example A.20. JMS_SUBSCRIPTIONS Exception

```
   17:17:57,167 WARN  [ServiceController] Problem starting service
jboss.mq.destination:name=testTopic,service=Topic
   org.jboss.mq.SpyJMSException: Error getting durable subscriptions for
topic TOPIC.testTopic; - nested throwable: (java.sql.SQLException:
[Microsoft][SQLServer 2000 Driver for JDBC][SQLServer]Invalid object
name 'JMS_SUBSCRIPTIONS'.)
   at
org.jboss.mq.sm.jdbc.JDBCStateManager.getDurableSubscriptionIdsForTopic(
JDBCStateManager.java:290)
   at
org.jboss.mq.server.JMSDestinationManager.addDestination(JMSDestinationM
anager.java:656)
```

Example A.21. Specifying a SelectMethod

```
   <connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;Databa
seName=jboss</connection-url>
```

A.6. MYSQL DATASOURCE

A.6.1. Installing the Driver

Procedure A.1. Installing the Driver

1. Download the driver from http://www.mysql.com/products/connector/j/. Make sure to choose the driver based on your version of MySQL.

2. Expand the driver ZIP or TAR file, and locate the **.jar** file.
3. Move the **.jar** file into ***$JBOSS_HOME*/server/*config_name*/lib**.
4. Copy the ***$JBOSS_HOME*docs/examples/jca/mysql-ds.xml** example datasource deployer file to ***$JBOSS_HOME*/server/*config_name*/deploy/**, for use as a template.

A.6.2. MySQL Local-TX Datasource

Example A.22. MySQL Local-TX Datasource

This example uses a database hosted on **localhost**, on port 3306, with **autoReconnect** enabled. This is not a recommended configuration, unless you do not need any Transactions support.

```
<datasources>
  <local-tx-datasource>

    <jndi-name>MySqlDS</jndi-name>

    <connection-url>jdbc:mysql://localhost:3306/database</connection-
url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>

    <user-name>username</user-name>
    <password>secret</password>

    <connection-property name="autoReconnect">true</connection-
property>

    <!-- Typemapping for JBoss 4.0 -->
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>

  </local-tx-datasource>
</datasources>
```

A.6.3. MySQL Using a Named Pipe

Example A.23. MySQL Using a Named Pipe

This example uses a database hosted locally, but uses a named pipe instead of TCP/IP.

```
<datasources>
  <local-tx-datasource>

    <jndi-name>MySQLDS</jndi-name>
    <connection-url>jdbc:mysql://./database</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>

    <user-name>username</user-name>
    <password>secret</password>
```

```
    <connection-property
name="socketFactory">com.mysql.jdbc.NamedPipeSocketFactory</connection-
property>

    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>

  </local-tx-datasource>
</datasources>
```

A.7. POSTGRESQL

Example A.24. PostgreSQL Local-TX Datasource

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

<datasources>
  <local-tx-datasource>
    <jndi-name>PostgresDS</jndi-name>
    <connection-url>jdbc:postgresql://[servername]:[port]/[database
name]</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

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

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>PostgreSQL 7.2</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Example A.25. PostgreSQL XA Datasource

This configuratino works for PostgreSQL 8.x and later.

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

<datasources>
  <xa-datasource>
```

```
    <jndi-name>PostgresDS</jndi-name>

    <xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">[servername]</xa-
datasource-property>
    <xa-datasource-property name="PortNumber">5432</xa-datasource-
property>

    <xa-datasource-property name="DatabaseName">[database name]</xa-
datasource-property>
    <xa-datasource-property name="User">[username]</xa-datasource-
property>
    <xa-datasource-property name="Password">[password]</xa-datasource-
property>

    <track-connection-by-tx></track-connection-by-tx>
  </xa-datasource>
</datasources>
```

A.8. INGRES

Example A.26. Ingres Datasource

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

<datasources>
  <local-tx-datasource>
    <jndi-name>IngresDS</jndi-name>
    <use-java-context>false</use-java-context>
    <driver-class>com.ingres.jdbc.IngresDriver</driver-class>
    <connection-url>jdbc:ingres://localhost:II7/testdb</connection-url>
    <datasource-class>com.ingres.jdbc.IngresDataSource</datasource-
class>
    <datasource-property name="ServerName">localhost</datasource-
property>
    <datasource-property name="PortName">II7</datasource-property>
    <datasource-property name="DatabaseName">testdb</datasource-
property>
    <datasource-property name="User">testuser</datasource-property>
    <datasource-property name="Password">testpassword</datasource-
property>
    <new-connection-sql>select count(*) from iitables</new-connection-
sql>

    <check-valid-connection-sql>select count(*) from iitables</check-
valid-connection-sql>
    <metadata>
      <type-mapping>Ingres</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

> [2]

---

[1] Source: *http://community.jboss.org/wiki/SetUpASybaseDatasource*

[2] Source: *http://community.ingres.com*

APPENDIX B. LOGGING INFORMATION AND RECIPES

B.1. LOG LEVEL DESCRIPTIONS

Table B.1, "**log4j** Log Level Definitions" lists the typical meanings for different log levels in **log4j**. Your application may interpret these levels differently, depending on your choices.

Table B.1. log4j Log Level Definitions

log4j Level	JDK Level	Description
FATAL		The Application Service is likely to crash.
ERROR	SEVERE	A definite problem exists.
WARN	WARNING	Likely to be a problem, but may be recoverable.
INFO	INFO	Low-volume detailed logging. Something of interest, but not a problem.
DEBUG	FINE	Low-volume detailed logging. Information that is probably not of interest.
	FINER	Medium-volume detailed logging.
TRACE	FINEST	High-volume detailed logging.

NOTE

The more verbose logging levels are not appropriate for production systems, because of the high level of output they generate.

Example B.1. Restricting Logged Information to a Specific Log Level

```
<!-- Show the evolution of the DataSource pool in the logs
[inUse/Available/Max]-->
<category
name="org.jboss.resource.connectionmanager.JBossManagedConnectionPool">
  <priority value="TRACE" class="org.jboss.logging.XLevel"></priority>
</category>
```

B.2. SEPARATE LOG FILES PER APPLICATION

To segregate logging output per application, assign **log4j** categories to specific appenders. This is typically done in the `conf/log4j.xml` deployment descriptor.

Example B.2. Filtering App1 Log Output to a Separate File

```
 <appender name="App1Log" class="org.apache.log4j.FileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"></errorHandler>
  <param name="Append" value="false"/>
  <param name="File" value="${jboss.server.home.dir}/log/app1.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
  </layout>
</appender>

...

<category name="com.app1">
  <appender-ref ref="App1Log"></appender-ref>
</category>
<category name="com.util">
  <appender-ref ref="App1Log"></appender-ref>
</category>
```

Example B.3. Using TCLMCFilter

Enterprise Platform 5.1 includes the new class **`jboss.logging.filter.TCLMCFilter`**, which allows you to filter based on the deployment URL.

```
 <appender name="App1Log" class="org.apache.log4j.FileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"></errorHandler>
  <param name="Append" value="false"/>
  <param name="File" value="${jboss.server.home.dir}/log/app1.log"/>
  <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
  </layout>
  <filter class="org.jboss.logging.filter.TCLMCFilter">
  <param name="AcceptOnMatch" value="true"/>
  <param name="DeployURL" value="app1.ear"/>
  </filter>

  <!-- end the filter chain here -->
  <filter class="org.apache.log4j.varia.DenyAllFilter"></filter>

</appender>
```

B.3. REDIRECTING CATEGORY OUTPUT

When you increase the level of logging for one or more categories, it is often useful to redirect the output to a seperate file for easier investigation. To do this you add an **`appender-ref`** to the category.

Example B.4. Adding an `appender-ref`

```
 <appender name="JSR77" class="org.apache.log4j.FileAppender">
  <param name="File" value="${jboss.server.home.dir}/log/jsr77.log"/>
  ...
</appender>

<!-- Limit the JSR77 categories -->
<category name="org.jboss.management" additivity="false">
  <priority value="DEBUG"></priority>
  <appender-ref ref="JSR77"></appender-ref>
</category>
```

All **`org.jboss.management`** output goes to the **`jsr77.log`** file. The **`additivity`** attribute controls whether output continues to go to the root category appender. If **`false`**, output only goes to the appenders referred to by the category.