Red Hat Application Server

JOnAS User Guide

Red Hat Application Server: JOnAS User Guide
Copyright © 1999-2005 by ObjectWeb Consortium

Table of Contents

I. Introductory JOnAS Documentation

The chapters in this section contain introductory information for all JOnAS users.

Table of Contents

redhat.

Chapter 1.

Java Open Application Server (JOnAS)

This chapter provides an overview of Red Hat Application Server and the JOnAS platform.

1.1. Introduction to Red Hat Application Server

Red Hat Application Server is a middleware platform—it is layered between the operating system and applications. This middleware links systems and resources that are scattered across the network.

Red Hat Application Server comprises a runtime system and associated development libraries for creating and deploying Java-based Web applications with dynamic content (for example, dynamic Web sites, portal servers, and content management systems). These applications might retrieve, display, or update data in database management systems such as PostgreSQL or Oracle, or they might communicate with standard application software, such as ERP systems, or with proprietary legacy applications.

Red Hat Application Server is a robust platform for the development and deployment of Web applications written in Java and built with JSP, servlet, and Enterprise JavaBeans (EJB) technologies. It has been built to standard protocols and APIs that have emerged from Java, J2EE, Web Services, SOAP (Single Object Access Protocol), XML (Extensible Markup Language), and CORBA (Common Object Request Broker Architecture) standards groups. Developers build their applications using these standards, while Red Hat's middleware infrastructure ensures compatibility with the guidelines set forth by the J2EE specifications.

1.1.1. J2EE

The Sun J2EE specification (http://java.sun.com/j2ee/), together with related specifications such as EJB (http://java.sun.com/products/ejb/) and JMS (http://java.sun.com/products/jms/), define an architecture and interfaces for developing and deploying distributed Internet Java server applications based on a multi-tier architecture. This specification facilitates and standardizes the development, deployment, and assembling of application components that will be deployable on J2EE platforms. These applications are typically web-based, transactional, database-oriented, multi-user, secure, scalable, and portable.

More precisely, the Sun J2EE specification describes two kinds of information:

- The first is the runtime environment, called a J2EE server, that provides the execution environment and the required system services, such as the transaction service, the persistence service, the Java Message Service (JMS), and the security service.
- The second is programmer and user information that explains how an application component should be developed, deployed, and used.

Not only will an application component be independent of the platform and operating system (because it is written in Java), it will also be independent of the J2EE platform.

A typical J2EE application is composed of:

- Presentation components, also called *web components*, that define the application Web interface. These are servlets (http://java.sun.com/products/servlet/) and JSPs (http://java.sun.com/products/jsp/).
- Enterprise components, the *Enterprise JavaBeans (EJB)*, that define the application business logic and application data.

The J2EE server provides containers for hosting web and enterprise components. The container provides the component with life-cycle management and interfaces the components with the services provided by the J2EE server:

- The *web container* handles servlet and JSP components.

- The *EJB container* handles the Enterprise JavaBeans components.

- A J2EE server can also provide an environment for deploying Java clients (accessing EJBs); this is called a *client container*.
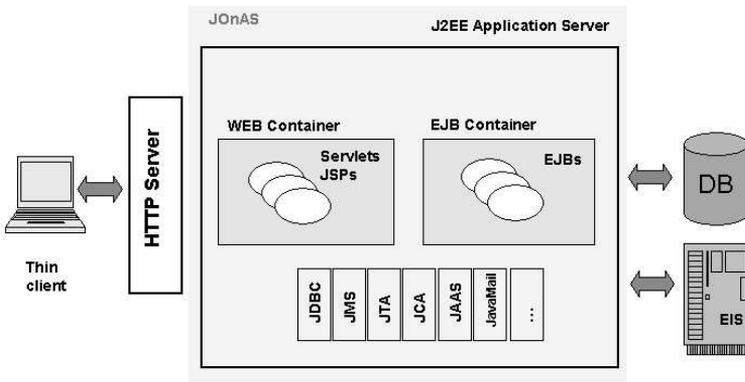
Figure 1-1. J2EE Architecture

1.2. JOnAS Features

JOnAS is a pure Java, open source, application server. Its high degree of modularity enables it to be used as:

- A J2EE server, for deploying and running EAR applications (that is, applications composed of both web and EJB components)

- An EJB container, for deploying and running EJB components (for example, for applications without web interfaces or when using JSP/servlet engines that are not integrated as a JOnAS container)

- A Web container, for deploying and running JSPs and servlets (for example, for applications without EJB components).

1.2.1. System Requirements

JOnAS is available for JDK 1.4. It has been used on many operating systems (Linux, AIX, Windows, Solaris, HP-UX, etc.) and with different databases (Oracle, PostgreSQL, MySQL, SQL server, Access, DB2, Versant, Informix, Interbase, etc.).

1.2.2. Java Standard Conformance

JOnAS supports the deployment of applications conforming to J2EE 1.4 specification. Its current integration of Tomcat as a Web container ensures conformity to Servlet 2.4 and JSP 2.0 specifications. The JOnAS server relies on or implements the following Java APIs: EJB 2.1, JCA 1.5, JDBC 3.0, JTA 1.0.1, JMS 1.1, JMX 1.2, JNDI 1.2.1, JAAS 1.0, JACC 1.0, and JavaMail 1.3.

1.2.3. Key Features

JOnAS provides the following important advanced features:

- Management: JOnAS server management uses JMX and provides a servlet-based management console.
- Services: JOnAS's service-based architecture provides for high modularity and configurability of the server. It allows the developer to apply a component-model approach at the middleware level, and makes the integration of new modules easy (for example, for open source contributors). It also provides a way to start only the services needed by a particular application, thus saving valuable system resources. You can manage JOnAS services through JMX.
- Scalability: JOnAS integrates several optimization mechanisms for increasing server scalability. This includes a pool of stateless session beans, a pool of message-driven beans, a pool of threads, a cache of entity beans, activation/passivation of entity beans, a pool of connections (for JDBC and JMS), and storage access optimizations (shared flag, isModified).
- Clustering: JOnAS clustering solutions, both at the WEB and EJB levels, provide load balancing, high availability, and failover support.
- Distribution: JOnAS works with several distributed processing environments, due to the integration of ObjectWeb's CAROL (Common Architecture for RMI ObjectWeb Layer) project (http://www.objectweb.org/carol/index.html), which enables simultaneous support of several communication protocols:
  - `CMI` (Cluster Method Invocation), the "Cluster aware" distribution protocol of JOnAS
  - `RMI` (Java Remote Method Invocation) using the Sun proprietary protocol JRMP (Java Remote Method Protocol)
  - `RMI` on IIOP (Java Remote Method Invocation over Internet Inter-Orb Protocol).

- Support of "Web Services": Due to the integration of AXIS, JOnAS allows J2EE components to access "Web Services" (that is, to be "Web Services" clients), and allows J2EE components to be deployed as "Web Services" endpoints. Standard Web Services clients and endpoints deployment, as specified in J2EE 1.4, is supported.
- Support of JDO: By integrating the ObjectWeb implementation of JDO (Java Data Objects) (http://java.sun.com/products/jdo), SPEEDO (http://speedo.objectweb.org), and its associated J2EE CA Resource Adapter, JOnAS provides the capability of using JDO within J2EE components.

Three critical J2EE aspects were implemented early on in the JOnAS server:

- J2EE CA: JOnAS applications can easily access Enterprise Information Systems (EIS). By supporting the Java Connector Architecture, JOnAS allows deployment of any J2EE CA-compliant Resource Adapter (connector), which makes the corresponding EIS available from the J2EE application components. Moreover, resource adapters will become the standard way to plug JDBC drivers (and JMS implementation, with J2EE 1.4) into J2EE platforms. A JDBC Resource Adapter available with JOnAS provides JDBC PreparedStatement pooling and can be used in place of the JOnAS DBM service. A JORAM JMS Resource adapter is also available.

- JMS (Java Messaging Service): JMS implementations can be easily plugged into JOnAS. They run as a JOnAS service either in the same JVM (Java Virtual Machine) or in a separate JVM, and JOnAS provides administration facilities that hide the JMS proprietary administration APIs. Currently, three JMS implementations can be used: the JORAM open-source JMS implementation from ObjectWeb (http://joram.objectweb.org/), SwiftMQ (http://www.swiftmq.com/), and Websphere MQ. J2EE CA Resource Adapters are also available, providing a more standard way to plug JORAM or SwiftMQ into JOnAS.

- JTA (Java Transaction API): The JOnAS platform supports distributed transactions that involve multiple components and transactional resources. The JTA transactions support is provided by a Transaction Monitor that has been developed on an implementation of the CORBA Object Transaction Service (OTS).

1.3. JOnAS Architecture

JOnAS is designed with services in mind. A service typically provides system resources to containers. Most of the components of the JOnAS application server are pre-defined JOnAS services. However, it is possible and easy for an advanced JOnAS user to define a service and to integrate it into JOnAS. Because J2EE applications do not necessarily need all services, it is possible to define, at JOnAS server configuration time, the set of services that are to be launched at server start.

The JOnAS architecture is illustrated in the following figure, showing WEB and EJB containers relying on JOnAS services (all services are present in this figure). Two thin clients are also shown in this figure, one of which is the JOnAS administration console (called JonasAdmin).
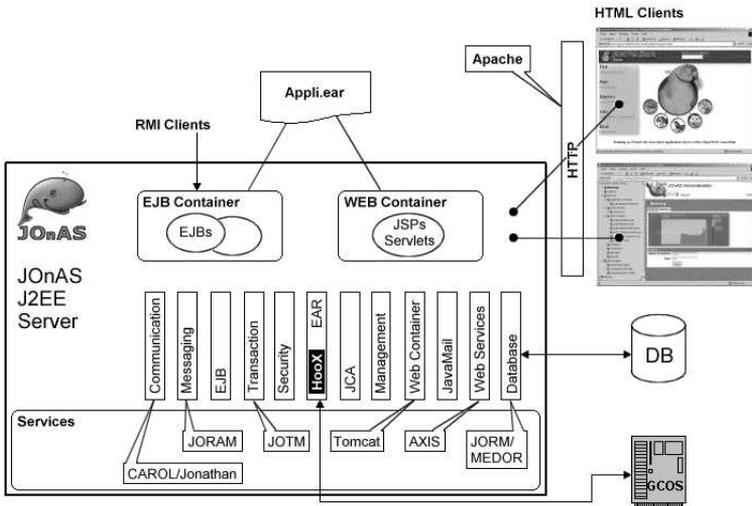
Figure 1-2. J2EE Architecture

1.3.1. Communication and Naming Service

The Communication and Naming Service (also called *Registry*) is used for launching the RMI registry, the CosNaming, and/or the CMI registry, depending on the JOnAS configuration (that is, the CAROL configuration, which specifies which communication protocols are to be used). There are different registry launching modes, such as using the same JVM or not, and launching automatically if not already running. CAROL enables multi-protocol runtime support and deployment, which avoids having to redeploy components when changing the communication protocol.

The Communication and Naming Service provides the JNDI (Java Naming and Directory Interface) API to application components and to other services in order to bind and look up remote objects (for example, EJB Homes) and resource references (JDBC DataSource, Mail, and JMS connection factories, etc.).

1.3.2. EJB Container Service

The EJB Container Service is in charge of loading the EJB components and their containers. EJB containers consist of a set of Java classes that implement the EJB specification and a set of interposition classes that interface the EJB components with the services provided by the JOnAS application server. Interposition classes are specific to each EJB component and are generated by the deployment tool called GenIC.

JOnAS configuration provides a means for specifying that this service be launched during JOnAS initialization.

Enterprise JavaBeans (EJB) are software components that implement the business logic of an application (while the servlets and JSPs implement the presentation). There are three types of Enterprise JavaBeans:

- Session beans are objects associated with only one client; they are short-lived (one method call or a client session) and represent the current state of the client session. They can be transaction-aware, stateful, or stateless.

- Entity beans are objects that represent data in a database. They can be shared by several clients and are identified by means of a primary key. The EJB container is responsible for managing the persistence of such objects. The persistence management of such an object is entirely transparent to the client that will use it, and may or may not be transparent to the bean provider who develops it. This depends on if it is one of the following:

  - For an Enterprise Bean with Container-Managed Persistence, the bean provider does not develop any data access code; persistence management is delegated to the container. The mapping between the bean and the persistent storage is provided in the deployment descriptor, in an application server-specific way.

  - For an Enterprise Bean with Bean-Managed Persistence, the bean provider writes the database access operations in the methods of the Enterprise Bean that are specified for data creation, load, store, retrieval, and remove operations.

- Message-driven Beans are objects that can be considered as message listeners. They execute on receipt of a JMS Java Message Service message; they are transaction-aware and stateless. They implement some type of asynchronous EJB method invocation. (Refer to http://java.sun.com/products/jms/.)

JOnAS configuration provides a means for specifying a set of EJB-JAR files to be loaded. EJB-JAR files can also be deployed at server runtime using the JOnAS administration tools.

For implementing Container-Managed Persistence of EJB 2.0 and EJB 2.1 (CMP2), JOnAS relies on the ObjectWeb JORM (Java Object Repository Mapping http://jorm.objectweb.org/) and MEDOR (Middleware Enabling Distributed Object Requests http://medor.objectweb.org/) frameworks. JORM

supports complex mappings of EJBs to database tables, as well as several types of persistency support (relational databases, object databases, LDAP repositories, etc.).

JOnAS also implements the Timer Service features as specified in EJB 2.1.

1.3.3. WEB Container Service

The WEB Container Service is in charge of running a servlet/JSP engine in the JVM of the JOnAS server and of loading web applications (WAR files) within this engine. Currently, this service can be configured to use Tomcat (see http://jakarta.apache.org/tomcat/). Servlet/JSP engines are integrated within JOnAS as *web containers*, that is, containers that provide the web components with access to the system resources (of the application server) and to EJB components, in a J2EE-compliant way.

JOnAS configuration provides a means for specifying that the WEB Container Service be launched during JOnAS initialization. Additionally, JOnAS configuration provides a means for specifying a set of WAR files to be loaded. Such WAR files can also be deployed at server runtime using the JOnAS administration tools. User management for Tomcat and JOnAS has been unified. The class-loading delegation policy (priority to the Webapp classloader or to the parent classloader) can be configured.

Servlet (http://java.sun.com/products/servlet/) and JSP (http://java.sun.com/products/jsp/) are technologies for developing dynamic web pages. The servlet approach allows the development of Java classes (HTTP servlets) that generate HTML pages and that can be invoked through HTTP requests. Typically, servlets access the information system using Java APIs (such as JDBC or the APIs of EJB components) in order to build the content of the HTML page they will generate in response to the HTTP request. The JSP technology is a complement of the servlet technology. A JSP is an HTML page containing Java code within particular XML-like tags; this Java code is in charge of generating the dynamic content of the HTML page.

Servlets and JSPs are considered as J2EE application components, responsible for the application presentation logic. Such application components can access resources provided by the J2EE server (such as JDBC datasources, JMS connection factories, EJBs, mail factories). For EJB components, the actual assignment of these resources is performed at component deployment time and is specified in the deployment descriptor of each component, because the component code uses logical resource names.

1.3.4. EAR Service

The EAR Service is used for deploying complete J2EE applications, that is, applications packaged in EAR files, which themselves contain EJB-JAR files and/or WAR files. This service handles the EAR files and delegates the deployment of the WAR files to the WEB Container service and the EJB-JAR files to the EJB Container service. It handles creating the appropriate class loaders, as defined in the J2EE specification, in order for the J2EE application to execute properly.

For deploying J2EE applications, JOnAS must be configured to launch the EAR service and to specify the set of EAR files to be loaded. EAR files can also be deployed at server runtime using the JOnAS administration tools.

1.3.5. Transaction Service

The Transaction Service encapsulates a Java Transaction Monitor called JOTM, a project from ObjectWeb (http://jotm.objectweb.org). It is a mandatory service that handles distributed transactions. It provides transaction management for EJB components as defined in their deployment descriptors. It handles two-phase commit protocol against any number of Resource Managers (XA Resources). For J2EE, a transactional resource can be a JDBC connection, a JMS session, or a J2EE CA Resource Adapter connection. The transactional context is implicitly propagated with the distributed requests. The Transaction Monitor can be distributed across one or more JOnAS servers; thus a transaction

may involve several components located on different JOnAS servers. This service implements the JTA 1.0.1 specification, thus allowing transactions from application components or from application clients to be explicitly started and terminated. Starting transactions from application components is allowed only from Web components, session beans, or message-driven beans. Restricting transactions to only these two types of beans is called *Bean-managed transaction demarcation*.

One of the main advantages of the EJB support for transactions is its declarative aspect, which means that transaction control is no longer hard-coded in the server application, but is configured at deployment time. This is known as *container-managed transaction demarcation*. With container-managed transaction demarcation, the transactional behavior of an Enterprise Bean is defined at configuration time and is part of the deployment descriptor of the bean. The EJB container is responsible for providing the transaction demarcation for the enterprise beans according to the value of transactional attributes associated with EJB methods. These attributes can be one of the following:

- `NotSupported`: If the method is called within a transaction, this transaction is suspended during the time of the method execution.

- `Required`: If the method is called within a transaction, the method is executed in the scope of this transaction; otherwise, a new transaction is started for the execution of the method and committed before the method result is sent to the caller.

- `RequiresNew`: The method is always executed within the scope of a new transaction. The new transaction is started for the execution of the method and committed before the method result is sent to the caller. If the method is called within a transaction, this transaction is suspended before the new one is started, and resumed when the new transaction has completed.

- `Mandatory`: The method should always be called within the scope of a transaction; otherwise, the container throws the `TransactionRequired` exception.

- `Supports`: The method is invoked within the caller transaction scope. If the caller does not have an associated transaction, the method is invoked without a transaction scope.

- `Never`: With this attribute the client is required to call the method without a transaction context, otherwise the container throws the `java.rmi.RemoteException` exception.

The ObjectWeb project JOTM (Java Open Transaction Manager), is actually based on the transaction service of earlier JOnAS versions. It will be enhanced to provide advanced transaction features, such as nested transactions and "Web Services" transactions (an implementation of DBTP is available).

1.3.6. Database Service

This service is responsible for handling Datasource objects. A Datasource is a standard JDBC administrative object for handling connections to a database. The database service creates and loads such datasources on the JOnAS server. DataSources to be created and deployed can be specified at JOnAS configuration time, or they can be created and deployed at server runtime using the JOnAS administration tools. The database service is also responsible for connection pooling; it manages a pool of database connections to be used by the application components, thus avoiding many physical connection creations, which are time-consuming operations. The database service can now be replaced by the JDBC Resource Adapter, to be deployed by the J2EE CA resource service, which additionally provides JDBC PreparedStatement pooling.

1.3.7. Security Service

The Security Service implements the authorization mechanisms for accessing J2EE components, as specified in the J2EE specification.

- EJB security is based on the concept of *roles*. The methods can be accessed by a given set of roles. In order to access the methods, you *must* be in at least one role of this set.

  The mapping between roles and methods (permissions) is done in the deployment descriptor using the `security-role` and `method-permission` elements. Programmatic security management is also possible using two methods of the EJBContext interface in order to enforce or complement security check in the bean code: `getCallerPrincipal()` and `isCallerInRole (String roleName)`. The role names used in the EJB code (in the `isCallerInRole` method) are, in fact, references to actual security roles, which makes the EJB code independent of the security configuration described in the deployment descriptor. The programmer makes these role references available to the bean deployer or application assembler by way of the `security-role-ref` elements included in the `session` or `entity` elements of the deployment descriptor.

- Web security uses the same mechanisms; however, permissions are defined for URL patterns instead of EJB methods. Therefore, the security configuration is described in the Web deployment descriptor. Programmatically, the caller role is accessible within a web component via the `isUserInRole (String roleName)` method.

In JOnAS, the mapping between roles and user identification is done in the user identification repository. When using Tomcat for user authentication, this user identification repository can be stored either in files, in a JNDI repository (such as LDAP), or in a relational database. This is achieved through a JOnAS implementation of the Realm for each Web container and through the JAAS (Java Authentication and Authorization Service) login modules for Java clients.

Realms use authentication resources provided by JOnAS, which enable you to rely on files, LDAP, or JDBC. These realms are in charge of propagating the security context to the EJB container during EJB calls. JAAS login modules are provided for user authentication of Web Container and Java clients. Certificate-based authentication is also available, with the CRLLoginModule login module for certificate revocation.

JOnAS also implements the Java Authorization Contract for Containers (JACC 1.0) specification, allowing you to manage authorizations as Java security permissions and to plug in any security policy provider.

1.3.8. Messaging Service

Asynchronous EJB-method invocation is possible on Message-driven Bean components. A Message-driven Bean is an EJB component that can be considered to be a JMS (Java Message Service) MessageListener; that is, a service that processes JMS messages asynchronously (see http://java.sun.com/products/jms). It is associated with a JMS destination. Its `onMessage` method is activated on the reception of messages sent by a client application to this destination. It is also possible for any EJB component to use the JMS API within the scope of transactions managed by the application server.

For supporting Message-driven Beans and JMS operations coded within application components, the JOnAS application server relies on a JMS implementation. JOnAS makes use of a third-party JMS implementation; currently the JORAM open-source software is integrated and delivered with JOnAS, the SwiftMQ product can also be used, and other JMS provider implementations can easily be integrated (see http://joram.objectweb.org/ and http://www.swiftmq.com/). JORAM provides several noteworthy features, particularly:

- Reliability (with a persistent mode)

- Distribution (transparently to the JMS client, it can run as several servers, thus allowing load balancing)

- The choice of TCP or SOAP as the communication protocol for messages.

The JMS service is in charge of launching (or establishing a connection to) the integrated JMS server, which may or may not run in the same JVM as JOnAS. It also provides connection pooling and thread pooling (for Message-driven Beans). Through this service, JOnAS provides facilities to create JMS-administered objects such as the connection factories and the destinations, either at server-launching time or at runtime using the JOnAS administration tools.

Note that the same function of JMS implementation integration can now be achieved through a Resource Adapter, to be deployed by the J2EE CA Resource Service. Such a Resource Adapter, J2EE CA 1.5, is provided for JORAM.

1.3.9. J2EE CA Resource Service

The J2EE Connector Architecture (J2EE CA) allows the connection of different Enterprise Information Systems (EIS) to a J2EE application server. It is based on the Resource Adapter (RA); this is an architecture component—comparable to a software driver—that connects the EIS, the application server, and the enterprise application (J2EE components). The RA is generally provided by an EIS vendor and provides a Java interface (the Common Client Interface or CCI) to the J2EE components for accessing the EIS (this can also be a specific Java interface). The RA also provides standard interfaces for plugging into the application server, allowing them to collaborate to keep all system-level mechanisms (transactions, security, and connection management) transparent from the application components.
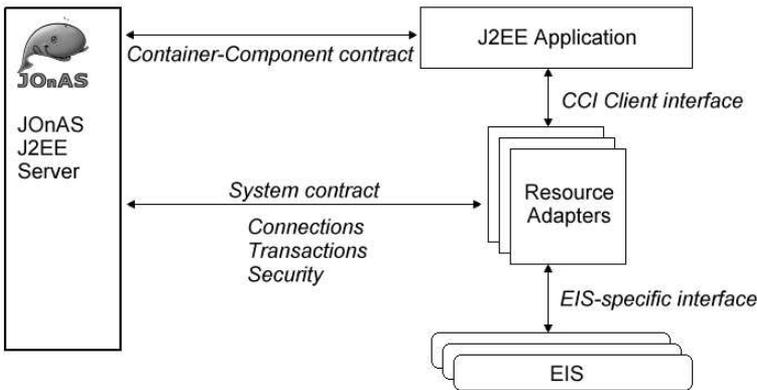
Figure 1-3. JCA Architecture

The application performs "business logic" operations on the EIS data using the RA client API (CCI), while transactions, connections (including pooling), and security on the EIS are managed by the application server through the RA (system contract).

The JOnAS Resource service is in charge of deploying J2EE CA-compliant Resource Adapters (connectors), packaged as RAR files, on the JOnAS server. RAR files can also be included in EAR files, in which case the connector will be loaded by the application classloader. Once Resource Adapters are deployed, a connection factory instance is available in the JNDI namespace to be looked up by application components.

A J2EE CA 1.0 Resource Adapter for JDBC is available with JOnAS. It can replace the current JOnAS database service for plugging JDBC drivers and managing connection pools. It also provides JDBC

PreparedStatement pooling.

A J2EE CA 1.5 Resource Adapter for JMS is available with JOnAS. It can replace the current JOnAS Messaging service for plugging into JORAM.

1.3.10. Management Service

You require the Management service in order to administer a JOnAS server from the JOnAS administration console. Each server running this service is visible from the administration console.

The Management service is based on JMX (Java Management Extension). Standard MBeans defined within the JOnAS application server expose the management methods of the instrumented JOnAS server objects, such as services, containers, and the server itself. These MBeans implement the management model as specified in the J2EE Management Specification. The Management service runs a JMX server. The MBeans of the JOnAS server are registered within this JMX server.

The JOnAS administration console is a Struts-based Web application (servlet/JSP) that accesses the JMX server to present the managed features within the administration console. Thus, through a simple Web browser, it is possible to manage one or several JOnAS application servers. The administration console provides a means for configuring all JOnAS services (and making the configuration persistent) and for deploying any type of application (EJB-JAR, WAR, EAR) and any type of resource (DataSources, JMS and Mail connection factories, J2EE CA connectors), all without the need to stop or restart the server. The administration console displays information for monitoring the servers and applications. This information includes used memory, used threads, number of EJB instances, and which component currently uses which resources. When Tomcat is used as Web Container, the Tomcat Management is integrated within the JOnAS console. A Management EJB (MEJB) is also delivered, providing access to the management features, as specified in the J2EE Management Specification.

1.3.11. Mail Service

A J2EE application component can send e-mail messages using JavaMail (see http://java.sun.com/products/javamail/). The Mail service of the JOnAS application server provides the resources necessary for the J2EE application components. The Mail service creates mail factories and registers these resources in the JNDI namespace in the same way that the database service or the JMS service creates Datasources or ConnectionFactories and registers these objects in the JNDI namespace. There are two types of mail factories: `javax.mail.Session` and `javax.mail.internet.MimePartDataSource`.

1.3.12. WebServices Service

This service is implemented on top of AXIS and is used for the deployment of Web Services.

1.4. JOnAS Development and Deployment Environment

The JOnAS development and deployment environment comprises the JOnAS configuration and deployment facilities and the JOnAS development environment.

1.4.1. JOnAS Configuration and Deployment Facilities

Once JOnAS has been installed in a directory referenced by the `JONAS_ROOT` environment variable, it is possible to configure servers and to deploy applications into several execution environments. This is achieved using the `JONAS_BASE` environment variable. `JONAS_ROOT` and `JONAS_BASE` can be compared to the `CATALINA_HOME` and `CATALINA_BASE` variables of Tomcat. While `JONAS_ROOT`

is dedicated to JOnAS installation, `JONAS_BASE` is used to specify a particular JOnAS instance configuration. `JONAS_BASE` designates a directory containing a specific JOnAS configuration, and it identifies subdirectories containing the EJB-JAR, WAR, EAR, and RAR files that can be loaded in this application environment. There is an ANT target in the JOnAS `build.xml` file for creating a new `JONAS_BASE` directory structure. Thus, from one JOnAS installation, it is possible to switch from one application environment to another by just changing the value of the `JONAS_BASE` variable.

There are two ways to configure a JOnAS application server and load applications: either by using the administration console or by editing the configuration files. There are also "autoload" directories for each type of application and resource (EJB-JAR, WAR, EAR, RAR) that allow the JOnAS server to automatically load the applications located in these directories when starting.

JOnAS provides several facilities for deployment:

- For writing the deployment descriptors, plugins for Integrated Development Environments (IDE) provide some generation and editing features (Eclipse and JBuilder plugins are available). The NewBean JOnAS built-in tool generates template deployment descriptors. The Xdoclet tool also generates deployment descriptors for JOnAS. The Apollon ObjectWeb project generates Graphical User Interfaces for editing any XML file; it has been used to generate a deployment descriptor editor GUI (see http://debian-sf.objectweb.org/projects/apollon). A deployment tool developed by the ObjectWeb JOnAS community, earsetup (http://sourceforge.net/projects/earsetup), will also be available for working with the JSR88-compliant (J2EE 1.4) deployment APIs provided by the ObjectWeb Ishmael project (see http://sourceforge.net/projects/earsetup/ and http://debian-sf.objectweb.org/projects/ishmael/ respectively).
- Some basic tools for the deployment itself are the JOnAS GenIC command line tool and the corresponding ANT EJB-JAR task. The IDE plugins integrate the use of these tools for deployment operations. The main feature of the Ishmael project will be the deployment of applications on the JOnAS platform.

1.4.2. JOnAS Development Environments

There are many plugins and tools that facilitate the development of J2EE applications to be deployed on JOnAS. IDE plugins for JBuilder Kelly (http://forge.objectweb.org/projects/kelly/), JOPE (http://forge.objectweb.org/projects/jope/), and Lomboz (http://lomboz.objectweb.org) provide the means to develop, deploy, and debug J2EE components on JOnAS. The Xdoclet code generation engine (http://xdoclet.sourceforge.net/) can generate EJB interfaces and deployment descriptors (standard and JOnAS specific ones), taking as input the EJB implementation class containing specific JavaDoc tags. The JOnAS NewBean tool generates templates of interfaces, implementation class, and deployment descriptors for any kind of EJB. Many development tools may work with JOnAS; refer to the JOnAS tools page at http://www.objectweb.org/jonas/tools.html for more details.

In addition, JOnAS is delivered with complete J2EE examples, providing a `build.xml` ANT file with all the necessary targets for compiling, deploying, and installing J2EE applications.

1.5. Clustering and Performance

Clustering for an application server generally makes use of three features: Load Balancing (LB), High Availability (HA), and Failover. Such mechanisms can be provided at the Web-container level by dispatching requests to several Servlet/JSP engine instances, at the EJB-container level by dispatching EJB requests to several EJB container instances, and at the database level by using several databases. A replicated JNDI naming is also necessary.

JOnAS provides Load Balancing, High Availability, and Failover at the WEB container level using the Apache Tomcat mod_jk plugin and an HTTP-in-memory session-replication mechanism based on JGroups. The plugin dispatches HTTP requests from the Apache web server to Tomcat instances

running as JOnAS web containers. Server fluctuations are automatically taken into account. This plugin supports round-robin and weighted round-robin load-balancing algorithms, with a sticky session option.

Load balancing and HA are provided at the EJB container level in JOnAS. Operations invoked on EJB Home interfaces (EJB creation and retrieval) are dispatched on the nodes of the cluster. The mechanism is based on a *clustered-aware* replicated JNDI registry using a Clustered remote Method Invocation protocol (CMI). The stubs contain the knowledge of the cluster and implement the load-balancing policy, which may be round-robin and weighted round-robin. In the near future, a load-balancing mechanism based on the nodes load will be available. Failover at the EJB level will be provided by implementing a stateful Session Bean state replication mechanism.

The JOnAS clustering architecture is illustrated in the following figure.
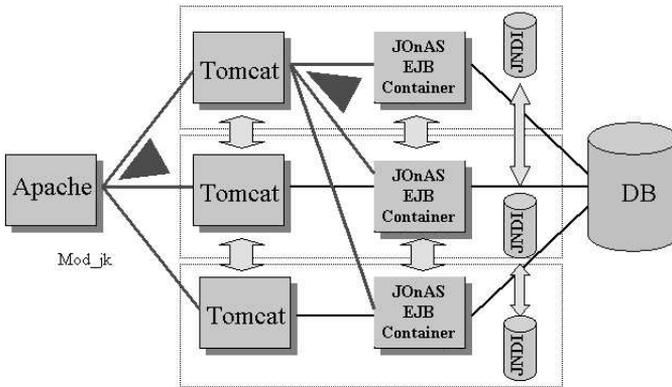
Figure 1-4. Clustered Architecture

Apache is used as the front-end HTTP server; Tomcat is used as the JOnAS web container. The JOnAS servers share the same database. The mod_jk plug-in provides load balancing/high availability at the Servlet/JSP level. Failover is provided through the in-memory, session-replication mechanism. Load balancing/high availability are provided at the EJB level through the CMI protocol associated with the replicated, clustered-aware JNDI registry. Tomcat may or may not run in the same JVM as the EJB container. JOnAS provides some documentation for configuring such an architecture.

The use of the C-JDBC ObjectWeb project offers load balancing and high availability at the database level (see http://www.objectweb.org/c-jdbc/index.html). The use of C-JDBC is transparent to the application (in our case, to JOnAS), because it is viewed as a standard JDBC driver. However, this "driver" implements the cluster mechanisms (reads are load-balanced and writes are broadcasted). The database is distributed and replicated among several nodes, and C-JDBC load balances the queries between these nodes. An evaluation of C-JDBC using the TPC-W benchmark on a 6-node cluster has shown performance scaling linearly up to six nodes.

In addition to clustering solutions, JOnAS provides many intrinsic mechanisms to ensure high scalable and efficiency:

- A pool of stateless Session Bean instances
- A pool of Entity Bean instances, configurable for each Entity Bean within its deployment descriptor
- Activation/passivation of entity beans (passivation can be controlled through the management console)
- Pools of connections, for JDBC, JMS, J2EE CA connectors
- A pool of threads for message-driven beans
- Session Bean timeout can be specified at deployment
- A "shared" flag in the specific deployment descriptor of an Entity Bean that indicates whether the persistent representation of this Entity Bean is shared by several servers/applications, or whether it is dedicated to the JOnAS server where it is loaded. In the latter case, the optimization performed by JOnAS consists of not reloading the corresponding data between transactions.
- The usual EJB 1.1 "isModified" (or "Dirty") mechanism is available, for avoiding storage of unmodified data.

Some benchmarks and JOnAS Use cases have already proven that JOnAS is highly scalable. Refer to the Rubis http://www.cs.rice.edu/CS/Systems/DynaServer/perf_scalability_ejb.pdf results or the OpenUSS Use case (http://openuss.sourceforge.net/openuss/). Rubis is a benchmark for e-commerce J2EE applications, which now belongs to the ObjectWeb JMOB (Java Middleware Open Benchmarking) project (http://www.objectweb.org/jmob/index.html). OpenUSS is an operational university portal that has approximately 20,000 users.

1.6. Future Development

As an open source implementation of an application server, JOnAS is constantly evolving to satisfy user requirements and to follow the related standards. These are the current JOnAS plans:

- JOnAS administration will be enhanced by completing the concept of management domain, and by introducing cluster management facilities.
- Addressing performance issues by developing workbenches and by producing tuning guides.
- Support of "Web Services" and tools for developing those services.

Chapter 2.

Getting Started with JOnAS

This tutorial guides you through running a first example EJB. Guidance is also provided for running a more complex example in which an EJB has access to a database.

You can find additional information about JOnAS configuration in Chapter 3 *JOnAS Configuration*.

2.1. Running the First EJB Application

2.1.1. JOnAS Examples

There are several examples in the JOnAS distribution under `$JONAS_ROOT/examples/src`. You should run the example located in the `$JONAS_ROOT/examples/src/sb/` directory first.

In this example, a Java client accesses a stateful Session Bean and calls the `buy` method of the bean several times inside the scope of transactions.

2.1.2. Building the sb Example

The easiest way to compile this example is to go to the `sb` directory `$JONAS_ROOT/examples/src/sb/` and use the `compile.sh` shell script.

2.1.3. Running the sb Example

This a distributed example in which two processes are involved:

- The JOnAS server, in which beans will be loaded
- The Java client that creates instances of beans and calls business methods on it.

To run this example:

1. Run the JOnAS server:
   service jonas start

   The following message is displayed on the standard output:
   ```
   The JOnAS Server 'jonas' version-number is ready
   ```

2. Make beans available to clients by loading the `jar` containing the sb example:
   jonas admin –a sb.jar

   The following message is displayed on the standard output:
   ```
   message-header: Op available
   ```

3. Run the Java client in another terminal command-line window:
   jclient sb.ClientOp

4. If the following output displays, the first EJB application with JOnAS has run successfully:
   ```
   Create a bean
   Start a first transaction
   First request on the new bean
   Second request on the bean
   Commit the transaction
   Start a second transaction
   ```

```
Rollback the transaction
Request outside any transaction
ClientOp OK. Exiting.
```

5. Before ending this session, be sure to stop the JOnAS server:
   service jonas stop

These instructions are also located in the README file in the working directory.

2.1.4. Understanding Why This Works

- This example demonstrates that the CLASSPATH is correctly set because, when the JOnAS server is launched, the jonas script calls the JOnAS bootstrap class.

  The jclient script is being used to run the client. Note that the bean classes were found in $JONAS_ROOT/examples/classes. If this had not been the case, it would have been necessary to call the jclient script with the -cp "$JONAS_ROOT/ejbjars/sb.jar" option.

- The client has succeeded in contacting the JOnAS server because the client has a distributed reference that was previously registered in the naming service. To do this, server and client use **JNDI** (the Java Naming and Directory Interface). The carol.properties file (which contains the JNDI configuration) is located in the $JONAS_ROOT/conf directory.

  This carol.properties has the JNDI properties set to the default values.

  With these default values, the registry runs on localhost on the default port (1099 for RMI/JRMP).

  By default, the registry is launched in the same JVM as the JOnAS Server.

2.2. More Complex Examples

2.2.1. Other JOnAS Examples

The following examples are located under $JONAS_ROOT/examples/src:

- The eb example ($JONAS_ROOT/examples/src/eb/) uses Entity beans. (See Section 2.2.2 *An Example With Database Access*.)

  The two beans share the same interface (Account): one uses bean-managed persistence (explicit persistence), the other uses container-managed persistence (implicit persistence).

  This is a good example for understanding what must be done, or not done, for persistence, based on the chosen mode of persistence. It provides both a CMP 1.1 and a CMP 2.0 implementation.

- The lb example ($JONAS_ROOT/examples/src/lb/) uses Entity beans with local interfaces.

  A Session Bean, Manager, locally manages an Entity Bean, Manac, which represents an account.

  This is a good example for understanding what must be done for a local client collocated with an Entity Bean providing local interfaces.

- The jms directory ($JONAS_ROOT/examples/src/jms/) contains a stateful Session Bean with methods performing JMS operations and a pure JMS client message receptor.

  A complete description of this example is in Section 26.6 *A JMS EJB Example*.

- The mailsb directory ($JONAS_ROOT/examples/src/mailsb/) contains a **SessionMailer** and **MimePartDSMailer Stateful Session beans** with methods providing a way for building and sending mail.

- The `mdb/samplemdb` directory (`$JONAS_ROOT/examples/src/mdb/samplemdb/`) contains a **Message Driven Bean** that listens to a topic and an MdbClient, which is a pure JMS client, that sends 10 messages on the corresponding topic.

  This is a very good example for understanding how to write and use message-driven beans.

- The `mdb/sampleappli` directory (`$JONAS_ROOT/examples/src/mdb/sampleappli/`) contains the following:

  - Two **Message-Driven beans**, one listening to a topic (`StockHandlerBean`) the other listening to a queue (`OrderBean`)

  - An Entity Bean with container-managed persistence (`StockBean`) and a stateless Session Bean for creating the table used in the database.

  SampleAppliClient sends several messages on the topic. Upon receipt of a message, the **StockHandlerBean** updates the database via the **StockBean** and sends a message to the queue inside a global transaction. All of the EJBs are involved in transactions that may commit or rollback.

- Alarm (http://jonas.objectweb.org/current/examples/alarm/) is an application that watches alarm messages generated asynchronously through JMS. It utilizes the different techniques used in JOnAS:

  - An Entity Bean for AlarmRecord, persistent in a database

  - A Session Bean to allow clients to visualize Alarms received

  - A Message Driven Bean to catch Alarms

  - JMS to use a topic where Alarms are sent and received

  - Tomcat (for JSP pages and servlets)

  - Security.

- Earsample (http://jonas.objectweb.org/current/examples/earsample/) contains a complete J2EE application. This sample is a simple stateful Session Bean that has synchronization and security.

  This bean is accessed from a servlet in which the user is authenticated and JOnAS controls access to the methods of the bean. The servlet performs a variety of lookups (resource-ref, resource-env-ref, env-entry, ejb-ref, and ejb-local-ref) in the java:comp/env environment to illustrate how the uniform naming works in the servlets.

- The `CMP2` directory (http://jonas.objectweb.org/current/examples/cmp2/) contains an example illustrating most of the concepts of CMP 2.0.

- `jaasclient` (http://jonas.objectweb.org/current/examples/jaasclient/) contains an example of a JAAS login module that illustrates the different methods for authentication.

  There are different callback handlers that demonstrate how to enter identification at a command line prompt, either with a dialog or without a prompt (where the client uses its own login/password).

Each directory contains a `README` that explains how to build and run each example.

2.2.2. An Example With Database Access

The **eb** example contains two Entity beans that manage `Account` objects.

The two beans share the same interface (Account); one with bean-managed persistence (BMP, explicit persistence), the other with container-managed persistence (CMP, implicit persistence). The default CMP implementation is CMP 1.1. A CMP 2.0 implementation is also provided and its use is described in the `README`.

Before running this example, perform the steps in:

- Section 2.2.2.1 *Configuring Database Access*
- Section 2.2.2.2 *Creating the Table in the Database*
- Section 2.2.2.3 *Configuring the Classpath*

2.2.2.1. Configuring Database Access

In order to be able to access your relational database, JOnAS will create and use a DataSource object that must be configured according to the database that will be used.

These DataSource objects are configured via properties files. `$JONAS_ROOT/conf` contains templates for configuring DataSource objects for databases such as Oracle and PostgreSQL:

- `$JONAS_BASE/conf/Oracle1.properties`
- `$JONAS_BASE/conf/PostgreSQL1.properties`

Depending on your database, you can customize one of these files with values appropriate for your installation. After doing so, you must update the property `jonas.service.dbm.datasources` in the `jonas.properties` file.

For example, for the `Oracle1.properties` file.

```
jonas.service.dbm.datasources     Oracle1
```

Section 3.5.8 *Configuring the Database Service* provides more details about DataSource objects and their configuration.

2.2.2.2. Creating the Table in the Database

The `$JONAS_ROOT/examples/src/eb` directory contains an SQL script for Oracle: `Account.sql` (`$JONAS_ROOT/examples/src/eb/Account.sql`). If your Oracle server is running and you are using CMP 1.1, you can create the table used by the example. If you are using CMP 2.0, do not create the table.

2.2.2.2.1. Example: Creating the Table in Oracle

sqlplus user/passwd

SQL> @Account.sql

SQL> quit

2.2.2.3. Configuring the Classpath

The JDBC driver classes must be accessible from the classpath. To enable that, update the `config_env` file (http://jonas.objectweb.org/current/bin/unix/config_env).

In this file, set one of the following variables: IDB_CLASSES, ORACLE_CLASSES, or POSTGRE_CLASSES with the appropriate value for your database installation.

2.2.2.4. Building the eb Example

The simplest way to compile this example is to go to the `$JONAS_ROOT/examples/src/eb` directory and use the `compile.sh` shell script (`$JONAS_ROOT/examples/src/eb/compile.sh`).

If the **Ant 1.5** build tool is installed on your machine, you can build the JOnAS examples by using the `build.xml` files located in the `$JONAS_ROOT/examples` or `$JONAS_ROOT/examples/src` directories. To do this, use the `build.sh` shell script.

2.2.2.5. Running the eb Example

Here, again, two processes are involved:

- The JOnAS server in which beans will be loaded
- The Java client that creates instances of beans and calls business methods on it.

To run this example:

1. Run the JOnAS server to make beans available to clients:
   service jonas start

   jonas admin −a eb.jar

   The following messages are displayed on the standard output:
   ```
   The JOnAS Server 'jonas' version-number is ready and running on rmi

   message-header : AccountExpl available

   message-header : AccountImpl available
   ```

2. Run the Java clients in another terminal emulator window:
   jclient eb.ClientAccount AccountImplHome

   jclient eb.ClientAccount AccountExplHome

   The example `eb` has run successfully if the following output displays:
   ```
   Getting a UserTransaction object from JNDI

   Connecting to the AccountHome

   Getting the list of existing accounts in database

   101 Antoine de St Exupery 200.0
   102 alexandre dumas fils 400.0
   103 conan doyle 500.0
   104 alfred de musset 100.0
   105 phileas lebegue 350.0
   106 alphonse de lamartine 650.0

   Creating a new Account in database

   Finding an Account by its number in database

   Starting a first transaction, that will be committed

   Starting a second transaction, that will be rolled back

   Getting the new list of accounts in database

   101 Antoine de St Exupery 200.0
   102 alexandre dumas fils 300.0
   ```

```
103 conan doyle 500.0
104 alfred de musset 100.0
105 phileas lebegue 350.0
106 alphonse de lamartine 650.0
109 John Smith 100.0

Removing Account previously created in database

ClientAccount terminated
```

3. Before ending this session, be sure to stop the JOnAS server:
   service jonas stop

Chapter 3.

JOnAS Configuration

This chapter describes how to configure JOnAS.

3.1. JOnAS Configuration Rules

As described in Chapter 2 *Getting Started with JOnAS*, JOnAS is pre-configured and ready to use directly with `RMI/JRMP` for remote access, if visibility to classes other than those contained in the JOnAS distribution in `$JONAS_ROOT/lib` is not required.

To use `RMI/IIOP` for remote access or to work with additional Java classes (for example, JDBC driver classes), you must perform additional configuration tasks, such as setting a specific port number for the registry.

The JOnAS distribution contains a number of configuration files in `$JONAS_ROOT/conf` directory. These files can be edited to change the default configuration. However, it is recommended that the configuration files needed by a specific application running on JOnAS be placed in a separate location. This is done by using an additional environment variable called `JONAS_BASE`.

3.1.1. JONAS_BASE Environment Variable

⚠️ **Warning**

> JOnAS configuration files are read from the `$JONAS_BASE/conf` directory. If `JONAS_BASE` is not defined, it is automatically initialized to `$JONAS_ROOT`.

There are two ways to use the `JONAS_BASE` environment variable:

1. Perform the following actions:

    a. Create a new directory and initialize `JONAS_BASE` with the path to this directory.

    b. Create the following sub-directories in `$JONAS_BASE`:
    - `conf`
    - `ejbjars`
    - `apps`
    - `webapps`
    - `rars`
    - `logs`

    c. Copy the configuration files located in `$JONAS_ROOT/conf` into `$JONAS_BASE/conf`. Then, modify the configuration files according to the requirements of your application, as explained in the following sections.

2. Perform the following actions:

    a. Initialize `$JONAS_BASE` with a path.

    b. Change to the `$JONAS_ROOT` directory and enter:
    ```
    ant create_jonasbase
    ```

This copies all the required files and creates all the directories.

Note

The `build.xml` files provided with the JOnAS examples support `JONAS_BASE`. If this environment variable is defined prior to building and installing the examples, the generated archives are installed under the appropriate sub-directory of `$JONAS_BASE`. For example, the EJB-JAR files corresponding to the sample examples of `$JONAS_ROOT/examples/src/` are installed in `$JONAS_BASE/ejbjars`.

3.2. Configuring the JOnAS Environment

3.2.1. The JOnAS Configuration File

The JOnAS server is configured via a configuration file named `jonas.properties`. It contains a list of key/value pairs presented in the Java properties file format.

The default configuration is provided in `$JONAS_ROOT/conf/jonas.properties` (refer to `$JONAS_BASE/conf/jonas.properties`). This file, which holds all possible properties with their default values, is mandatory. The JOnAS server looks for this file at start time in the `$JONAS_BASE/conf` directory (`$JONAS_ROOT/conf` if `$JONAS_BASE` is not defined).

Most of the properties are related to the JOnAS services that can be launched in the JOnAS server. These properties are described in detail in Section 3.5 *Configuring JOnAS Services*.

The property `jonas.orb.port` is not related to any service. It identifies the port number on which the remote objects receive calls. Its default value is `0`, which means that an anonymous port is chosen. When the JOnAS server is behind a firewall, this property can be set to a specific port number.

When several JOnAS servers must run simultaneously, it is beneficial to set a different name for each JOnAS server in order to administer these servers.

Also note that it is possible to define configuration properties on the command line:

`java –Dproperty=value`

Use the `jonas check` command to review the JOnAS configuration state. (Refer to Section 6.1 *jonas*.)

3.2.2. Configuration Scripts

The JOnAS distribution contains the `$JONAS_ROOT/bin/unix/setenv` and `$JONAS_ROOT/bin/unix/config_env` configuration scripts.

These configuration scripts set useful environment variables for JAVA setup (`$JAVA` and `$JAVAC`). They add `$JONAS_BASE/conf` to the `$CLASSPATH` if `$JONAS_BASE` is set, otherwise they add `$JONAS_ROOT/conf`. These scripts are called by almost all other scripts (`jclient`, `jonas`, `newbean`, `registry`, `GenIC`).

Therefore, when requiring the visibility of specific `.jar` files, the best practice is to update the `config_env` file. For example, to see some of the JDBC driver classes, one or more of the variables `IDB_CLASSES`, `ORACLE_CLASSES`, and `POSTGRE_CLASSES` must be updated.

Another way to place an additional `.jar` in the classpath of your JOnAS server is to insert it at the end of the `config_env` file:

```
CLASSPATH=<The_Path_To_Your_Jar>$SPS$CLASSPATH
```

Note that an additional environment variable called `XTRA_CLASSPATH` can be defined to load specific classes at JOnAS server start-up. Refer to Chapter 5 *JOnAS Class Loader Hierarchy*.

3.3. Configuring the Communication Protocol and JNDI

3.3.1. Choosing the Protocol

Typically, access to JNDI (the Java Naming and Directory Interface) is bound to a `jndi.properties` file that must be accessible from the classpath. This is somewhat different within JOnAS. Starting with JOnAS 3.1.2, multi-protocol support is provided through the integration of the CAROL component (refer to http://www.objectweb.org/carol). This currently provides support for RMI/JRMP, RMI/IIOP, and CMI (clustered protocol) by changing the configuration. Other protocols may be supported in the future. This configuration is now provided within the `carol.properties` file (that includes what was provided in the `jndi.properties` file). This file is supplied with the JOnAS distribution in the `$JONAS_ROOT/conf` directory (refer to `$JONAS_BASE/conf/carol.properties`).

The following communication protocols are supported:

- RMI/JRMP is the JRE implementation of RMI on the JRMP protocol. This is the default communication protocol.
- **RMI/IIOP** is the JacORB implementation of RMI over the IIOP protocol. The configuration file of JacORB is the `$JONAS_BASE/conf/jacorb.properties` file. As of version 4.3.1, JOnAS uses JacORB, not the JRE ORB implementation. The default iiop model now used is the POA model (http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/rmiiiopexample.html). Thus, GenIC should be relaunched on all previously generated beans.
- **CMI** (Cluster Method Invocation) is the JOnAS communication protocol used for clustered configurations. Note that this protocol is based on JRMP.

The `carol.properties` file contains:

```
# jonas rmi activation (jrmp, iiop,   or cmi)
carol.protocols=jrmp
#carol.protocols=cmi
#carol.protocols=iiop
#carol.protocols=jeremie
# RMI JRMP URL
carol.jrmp.url=rmi://localhost:1099

# RMI IIOP URL
carol.iiop.url=iiop://localhost:2000

# CMI URL
carol.cmi.url=cmi://localhost:2001

# general rules for jndi
carol.jndi.java.naming.factory.url.pkgs=org.objectweb.jonas.naming
```

CAROL can be customized by editing the `$JONAS_BASE/conf/carol.properties` file to:

- Choose the protocol through the `carol.protocols` property.

- Change `localhost` to the hostname where **registry** will be run.

- Change the standard port number.

If the standard port number is changed, **registry** must be run with this port number as the argument, `registry <Your_Portnumber>`, when the registry is not launched inside the JOnAS server. Note that Jeremie is now deprecated and could be removed in a future version.

You can configure JOnAS to use several protocols simultaneously. To do this, just specify a comma-separated list of protocols in the `carol.protocols` property of the `carol.properties` file. For example:

```
# jonas rmi activation (choose from iiop, jrmp, and cmi)
carol.protocols=iiop,jrmp
```

3.3.2. Security and Transaction Context Propagation

JOnAS implements EJB security and transactions by using the communication layer to propagate the security and transaction contexts across method calls. By default, the communication protocol is configured for context propagation. However, this configuration can be changed by disabling the context propagation for security and/or transaction; this is done primarily to increase performance. The context propagation can be configured in the `jonas.properties` file by setting the `jonas.security.propagation` and `jonas.transaction.propagation` properties to `true` or `false`:

```
# Enable the Security context propagation
jonas.security.propagation              true

# Enable the Transaction context propagation
jonas.transaction.propagation           true
```

3.3.3. Multi-protocol Deployment (GenIC)

The JOnAS deployment tool (GenIC) must be told which protocol stubs (for remote invocation) are to be generated. Choosing several protocols will eliminate the need to redeploy the EJBs when switching from one protocol to another. The default is that GenIC generates stubs for `rmi/jrmp`. To change this configuration, call GenIC as described in Section 6.5 *GenIC*. Use the `-protocols` option to specify a comma-separated list of protocols (chosen from `jrmp`, `iiop`, and `cmi`). For example:

```
GenIC -protocols jrmp,iiop
```

This list of protocols can also be specified for Ant EJB tasks (refer to Chapter 27 *Ant EJB Tasks: Using EJB-JAR*):

```
<jonas destdir="${dist.ejbjars.dir}"
    jonasroot="${jonas.root}"
    protocols="jrmp,iiop"
    keepgenerated="true"
    verbose="${verbose}"
    mappernames="${mapper.names}"
    additionalargs="${genicargs}">
</jonas>
```

As of JOnAS 4.3.1, the default IIOP model used is the POA model (http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/rmiiiopexample.html). Thus, GenIC should be relaunched on all previously generated beans.

3.4. Configuring Logging System (monolog)

The logging system is based on Monolog, the standard API for ObjectWeb projects (refer to http://www.objectweb.org/monolog/doc/index.html). Configuring trace messages inside Jonas can be done in two ways:

- Changing the `trace.properties` file to configure the traces statically, before the JOnAS Server is run (refer to `$JONAS_BASE/conf/trace.properties`).
- Using the `jonas admin` command (refer to Section 6.1 *jonas*) to configure the traces dynamically while the JOnAS Server is running.

Note

The SQL requests sent to a database can be easily traced using the JOnAS Logging system and the integrated P6Spy tool. The configuration steps are described in Chapter 4 *Configuring JDBC DataSources*.

3.4.1. trace.properties Syntax

A standard file is provided in `$JONAS_ROOT/conf/trace.properties` (refer to `$JONAS_BASE/conf/trace.properties`). Use the `CLASSPATH` to retrieve this file.

The monolog documentation described in http://www.objectweb.org/monolog/doc/index.html provides more details about how to configure logging. Monolog is built over a standard log API (currently, `log4j`). Loggers can be defined, each one being backed on a handler.

A handler represents an output, is identified by its name, has a type, and has some additional properties. Two handlers have been defined in the `trace.properties` file (refer to `$JONAS_BASE/conf/trace.properties`):

- `tty` is basic, standard output on a console, with no headers.
- `logf` is a handler for printing messages on a file.

Each handler can define the header it will use, the type of logging (console, file, rolling file), and the file name.

Note that if the tag `automatic` is specified as the output filename, JOnAS will replace this tag with a file pointing to `$JONAS_BASE/logs/<jonas_name_server>-<timestamp>.log`.

The `logf` handler, which is bundled with JOnAS, uses this `automatic` tag.

Loggers are identified by names that are structured as a tree. The root of the tree is named `root`.

Each logger is a topical logger (that is, it is associated with a topic). Topic names are usually based on the package name. Each logger can define the handler it will use and the trace level among the following four values:

- `ERROR` errors. Should always be printed.
- `WARN` warning. Should be printed.
- `INFO` informative messages, not typically used in Jonas (for example, test results).
- `DEBUG` debug messages. Should be printed only for debugging.

If nothing has been defined for a logger, it will use the properties defined for its parent.

JOnAS code is written using the monolog API and can use the `tty` handler.

3.4.1.1. Example: Setting the DEBUG Level

To set the DEBUG level for the logger used in the jonas_ejb module:

```
logger.org.objectweb.jonas_ejb.level DEBUG
```

3.4.1.2. Example: Setting the Output Traces to the Console and a File

To set the output of JOnAS traces to both the console and a file named /tmp/jonas/log:

```
handler.logf.output /tmp/jonas.log
logger.org.objectweb.jonas.handler.0 tty
logger.org.objectweb.jonas.handler.1 logf
```

3.4.1.3. Example: Setting the Output Trace to a File

To set the output of JOnAS traces to a file in the $JONAS_BASE/logs/ directory:

```
handler.logf.output automatic
logger.org.objectweb.jonas.handler.0 logf
```

3.5. Configuring JOnAS Services

JOnAS may be viewed as a number of manageable built-in services started at server launching time. JOnAS is also able to launch external services, which can be defined as explained in Chapter 24 *JOnAS Services*.

The following is a list of the JOnAS built-in services:

registry

> The registry binds remote objects and resources that will later be accessed via JNDI. It is automatically launched before all the other services when starting JOnAS.

jmx

> The jmx service enables you to administer the JOnAS servers and the JOnAS services via a JMX-based administration console. jmx launches automatically.

> JOnAS uses MX4J (refer to http://mx4j.sourceforge.net/).

jtm

> The Transaction Manager service is used for the support of distributed transactions. This is the only *mandatory* service for JOnAS.

dbm

> The database service is required by application components that need to access relational databases.

resource

> The resource service is needed for access to *Resource Adapters* conformant to the *J2EE Connector Architecture Specification*.

jms

As of the 4.1 release, a JMS provider can be integrated through the deployment of a resource adapter.

security

The `security` service enforces security at runtime.

ear

The EAR service provides support for J2EE applications.

mail

The `Mail` service is required by e-mail applications.

discovery

The `Discovery` service allows support for administration of *management domains*.

db

The `db` service is used for launching a Java database implementation. For instance, one implementation is done for the HSQLDB Java database.

ejb

The EJB Container service provides support for EJB application components.

web

The WEB Container service provides support for web components (as servlets and JSP). At this time JOnAS provides an implementation of this service for Tomcat.

ws

The WebServices service provides support for WebServices (WSDL publication).

The services available in a JOnAS server are those specified in the JOnAS configuration file. The `jonas.services` property in the `jonas.properties` file must contain a list of the required service names. Currently, these services are started *in the order* in which they appear in the list. Therefore, the following constraints should be considered:

- `jmx` must precede all other services in the list (except `registry`) in order to allow the management of these services.
- `jtm` must precede the `dbm`, `resource`, and `jms` services.
- `security` must be after `dbm`, as it uses datasources.
- The services used by the application components must be listed before the container service used to deploy these components. For example, if the application contains EJBs that need database access, `dbm` must precede `ejb` in the list of required services.

Example:

```
jonas.services registry,jmx,jtm,dbm,security,resource,jms,mail,ejb,ws,web,ear
```

The `registry` can be omitted from the list because this service is automatically launched if it is not already activated by another previously started server. This is also true for `jmx`, since, beginning with JOnAS 3.1, this service is automatically launched after the `registry`.

The `dbm`, `resource`, and `jms` services are listed after the `jtm`.

The application components deployed on this server can use Java Messaging and Java Mail because `jms` and `mail` are listed before `ejb`.

Configuration parameters for services are located in the `jonas.properties` file. They must follow the strict naming convention that a service *XX* will be configured via a set of properties:

```
jonas.service.XX.foo    something
jonas.service.XX.bar    else
```

3.5.1. Configuring the Registry Service

The `registry` service is used for accessing the RMI registry, CMI registry, or CosNaming (iiop), depending on the configuration of communication protocols specified in `carol.properties`. (refer to Section 3.3 *Configuring the Communication Protocol and JNDI*.)

There are several Registry-launching modes based on the value of the JOnAS property `jonas.service.registry.mode`. The possible values of this property are:

`automatic`

  The Registry is launched in the same JVM as JOnAS Server, if not already started. This is the default value.

`collocated`

  The Registry is launched in the same JVM as the JOnAS Server.

`remote`

  The Registry must be launched before the JOnAS Server in a separate JVM. (Refer to Section 6.4 *registry*.)

The port number on which the Registry is launched is defined in the `carol.properties` file.

3.5.2. Configuring the EJB Container Service

The EJB Container service is the primary JOnAS service. It provides EJB containers for EJB components.

You can create an EJB container from an EJB-JAR file in the following ways:

• The corresponding EJB-JAR file name is listed in the `jonas.service.ejb.descriptors` property in the `jonas.properties` file. If the file name does not contain an absolute path, it should be located in the `$JONAS_BASE/ejbjars/` directory. The container is created when the JOnAS server starts.

  Example:
  `jonas.service.ejb.descriptors   Bank.jar`

  In this example the *Container service* creates a container from the EJB-JAR file named `Bank.jar`. JOnAS will search for this file in the `$JONAS_BASE/ejbjars/` directory.

• To automatically create an EJB container at server start-up time, place the EJB-JAR files in an `autoload` directory. The name of this directory is specified using the `jonas.service.ejb.autoloaddir` property in the `jonas.properties` file.

JOnAS also allows for loading unpacked EJB components. The name of the xml file containing the EJB's deployment descriptor must be listed in the `jonas.service.ejb.descriptors` property. Note that the JOnAS server must have access to the component's classes, which may be achieved using the `XTRA_CLASSPATH` environment variable (refer to Chapter 5 *JOnAS Class Loader Hierarchy*).

3.5.3. Automatic Downloading of Stubs from Clients

JOnAS can be configured to allow dynamic Stubs downloads with the property `jonas.service.ejb.codebase.use`. Possible values are `true` and `false`.

☆**Important**

> This functionality works only in local mode (that is, the server and client must be in the same file system). Note that clients must have a SecurityManager set to download Stubs. If you use jclient, see Section 6.2 *jclient* script options.

```
java –Djava.security.manager –Djava.security.policy=/url/to/java.policy
   –jar /path/to/client.jar app-client.jar
```
Example 3-1. Client Container Example:

3.5.4. Configuring the WEB Container Service

The WEB Container service provides WEB containers for the WEB components used in the J2EE applications. JOnAS provides an implementation of the WEB Container service for Tomcat 5.0.x.

If you are not using a JOnAS/WebContainer package, that is, if the environment variable `$JETTY_HOME` or `$CATALINA_HOME` is not set, the web container service cannot be loaded at JOnAS startup and a warning is displayed.

A WEB container is created from a WAR file. If the file name does not contain an absolute path name, it must be located in the `$JONAS_BASE/webapps/` directory.

JOnAS can create WEB containers when the JOnAS server starts by providing the corresponding file names via the `jonas.service.web.descriptors` property in the `jonas.properties` file.

Example:

```
jonas.service.web.descriptors  Bank.war
```

In this example the WEB Container service creates a container from the WAR file named `Bank.war`. It searches for this file in the `$JONAS_BASE/webapps/` directory.

By using `webapp` directories instead of packaging a new WAR file each time, you can improve the development process. You can replace the classes with the new compiled classes, reload the servlets in your browser, and immediately see the changes. This is also true for the JSPs. Note that these reload features can be disabled in the configuration of the Tomcat web container at production time.

Example of using the `jonasAdmin/` webapp directory instead of `jonasAdmin.war`.

1. In the `JONAS_BASE/webapps/autoload` directory, create a directory (for example, `jonasAdmin`): JONAS_BASE/webapps/autoload/jonasAdmin

2. Move the `jonasAdmin.war` file to this directory.

3. Unpack the WAR file to the current directory, then remove the WAR file.

4. At the next JOnAS startup, the `webapp` directory is used instead of the WAR file. Change the JSP and see the changes at the same time.

3.5.5. Configuring the WebServices Service

3.5.5.1. A. Choose a Web Service Engine

At this time, only one implementation for WebServices is available: the Axis implementation. But in the future, a Glue implementation can be made easily.

In `jonas.properties`:

```
#...
# the fully qualifier name of the service class
jonas.service.ws.class org.objectweb.jonas.ws.AxisWSServiceImpl
#...
```

3.5.5.2. B. Choose One or More WSDL Handler(s)

WSDL Handlers are used to locate and publish all your WSDL documents. You can use several WSDL Handlers as long as you define them in the `jonas.properties` file.

Example:

If you want to publish a WSDL into the local file system, use the FileWSDLHandler

In `jonas.properties`:

```
#...
# a list of comma separated WSDL Handlers
    jonas.service.ws.wsdlhandlers file1
# Configuration of the file WSDL Handler
    jonas.service.ws.file1.type file
# Make sure users who run JOnAS have read/write access in this directory
    jonas.service.ws.file1.location /path/to/directory/where/store/wsdls
#...
```

With this configuration, JOnAS will try to read a property file named `file1.properties` (located along with `jonas.properties`). Here is its content:

```
#...

# FileWSDLHandler class
jonas.service.wsdl.class     org.objectweb.jonas.ws.handler.FileWSDLHandler

# Directory where WSDLs will be copied
jonas.service.publish.file.directory    /tmp

# Encoding of the file (In respect with the platform JOnAS is running on)
# If not set default to UTF-8
jonas.service.publish.file.encoding     UTF-8
```

3.5.5.3. Endpoint URL Prefix

The endpoint URL prefix property (`jonas.service.ws.url-prefix`) is used when you want to enforce the URL to be used for your deployed endpoints. This is clearly interesting when there is a cluster of JOnAS instances and an HTTP frontend (Apache HTTP for load balancing). You want all of your web services' endpoints to use the `http://www.mywebserver.com` URL instead of the usual `http://localhost:9000` (which has a meaning only at the local level).

3.5.6. Configuring the EAR Service

The `EAR service` allows deployment of complete J2EE applications (including both EJB-JAR and WAR files packed in an EAR file). This service is based on the `WEB container service` and the `EJB container service`. The `WEB container service` is used to deploy the WARs included in the J2EE application; the *EJB container service* is used to deploy the EJB containers for the EJB-JARs included in the J2EE application.

This service may be configured by setting the `jonas.service.ear.descriptors` property in `jonas.properties` file. This property provides a list of `ears` that must be deployed when JOnAS is launched.

When using relative paths for EAR file names, the files should be located in the `$JONAS_BASE/apps/` directory.

Example:

```
jonas.service.ear.descriptors  Bank.ear
```

In this example the `EAR service` will deploy the EAR file named `Bank.ear`. It will search for this file in the `$JONAS_BASE/apps/` directory.

3.5.7. Configuring the Transaction Service

The *Transaction service* is used by the *Container service* in order to provide transaction management for EJB components as defined in the deployment descriptor. This is a mandatory service.

The Transaction service uses a *Transaction manager* that may be local or may be launched in another JVM (a remote Transaction manager). Typically, when there are several JOnAS servers working together, one Transaction service must be considered as the *master* and the others as *slaves*. The slaves must be configured as if they were working with a remote Transaction manager.

The following is an example of the configuration for two servers: one named TM in which a standalone Transaction service will be run, one named EJB that will be used to deploy an EJB container:

```
jonas.name               TM
jonas.services           jtm
jonas.service.jtm.remote  false
```

and

```
jonas.name                 EJB
jonas.services             jmx,security,jtm,dbm,ejb
jonas.service.jtm.remote   true
jonas.service.ejb.descriptors   foo.jar
```

Another possible configuration option is the value of the transaction time-out, in seconds, via the `jonas.service.jtm.timeout` property.

The following is the default configuration:

```
jonas.service.jtm.timeout   60
```

3.5.8. Configuring the Database Service

 Note

> The description of the new JDBC Resource Adapters as a replacement for the database service is located in Chapter 55 *Configuring Resource Adapters*.

To allow access to one or more relational databases (for example, Oracle, PostgreSQL, and so on), JOnAS will create and use DataSource objects. Such a DataSource object must be configured according to the database that will be used for the persistence of a bean. More details about DataSource objects and their configuration are provided in Chapter 4 *Configuring JDBC DataSources*.

The following subsections briefly explain how to configure a DataSource object for your database in order to be able to run the Entity Bean example.

Note that the SQL requests sent to the database can be easily traced using the JOnAS Logging system and the integrated P6Spy tool. The configuration steps are described in Section 4.4 *Tracing SQL Requests Through P6Spy*.

3.5.8.1. Configuring Oracle for the Supplied Example

You can find a template `Oracle1.properties` file in the installation directory (refer to `$JONAS_BASE/conf/Oracle1.properties`). This file is used to define a `DataSource` object that is named `Oracle1`. This template must be updated with values appropriate to your installation. The fields are the following:

datasource.name	JNDI name of the DataSource: The name used in the example is `jdbc_1`.
datasource.url	The JDBC database URL: for the Oracle JDBC "Thin" driver it is jdbc:oracle:thin:@*hostname*:*sql*net_port_number*:ORACLE_SID If using an Oracle OCI JDBC driver, the URL is `jdbc:oracle:oci7:` or `jdbc:oracle:oci8:`
datasource.classname	Name of the class implementing the Oracle JDBC driver: `oracle.jdbc.driver.OracleDriver`
datasource.mapper	Adapter (JORM), mandatory for CMP2.0 only (more details in Section 4.2 *CMP2.0/JORM*): `rdb.oracle8` for Oracle 8 and prior versions
datasource.username	Database user name
datasource.password	Database user password

For the EJB platform to create the corresponding DataSource object, the `Oracle1` name must be in the `jonas.properties` file on the `jonas.service.dbm.datasources` line:

```
jonas.service.dbm.datasources      Oracle1
```

There may be several DataSource objects defined for an EJB server, in which case there will be several `dataSourceName.properties` files and a list of the DataSource names on the `jonas.service.dbm.datasources` line of the `jonas.properties` file:

```
jonas.service.dbm.datasources      Oracle1, Oracle2
```

To create the table used in the example with the SQL script that is provided in the

`examples/src/eb/Account.sql` file, the Oracle server must be running with a JDBC driver installed. (Oracle JDBC drivers can be downloaded from Oracle's web site: http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html)

For example:

```
sqlplus user/passwd
    SQL> @Account.sql
    SQL> quit
```

The JDBC driver classes must be accessible from the classpath. To do this, update the `config_env` file `$JONAS_ROOT/bin/unix/config_env`.

3.5.8.2. Configuring Other Databases

The same type of process can be used for other databases. A template of datasource for PostgreSQL and for InterBase is supplied with JOnAS. Although many other databases are currently used by the JOnAS users (for example, Informix, Sybase, SQL Server), not all JDBC drivers have been tested against JOnAS.

3.5.9. Configuring the Security Service

The *Security service* is used by the *Container service* to provide security for EJB components. The `Container service` provides security in two forms: declarative security and programmatic security. The Security service uses *security roles* and *method permissions* located in the EJB deployment descriptor.

Note that:

- JOnAS relies on Tomcat (http://jakarta.apache.org/tomcat) for the identification of the web clients. The Java clients use the JAAS login modules for the identification. JOnAS performs the user authentication.

  In the `$JONAS_ROOT/conf/jonas-realm.xml` file you can define three types of Realm for JOnAS:

  - Memory realm: users, groups, and roles are written in the file in the section `<jonas-memoryrealm>` of the `$JONAS_ROOT/conf/jonas-realm.xml` file.
  - Datasource realm: users, groups, and roles information is stored in a database; the configuration for accessing a specific datasource is described in the section `<jonas-dsrealm>` of the `$JONAS_ROOT/conf/jonas-realm.xml` file.

    The configuration requires the name of the datasource, the tables used, and the names of the columns.
  - LDAP realm: users, groups, and roles information is stored in an LDAP directory. This is described in the section `<jonas-ldaprealm>` of the `$JONAS_ROOT/conf/jonas-realm.xml` file.

    There are some optional parameters. If they are not specified, some of the parameters are set to a default value. That is, if the `providerUrl` element is not set, the default value is `ldap://localhost:389`.

    Edit the `jonas-realm_1_0.dtd` DTD file to see the default values.

  For Tomcat, use the realm: `org.objectweb.jonas.security.realm.web.catalina55.JACC`

  These realms require as an argument the `name` of the resource to be used for the authentication. This is the name that is in the `jonas-realm.xml` file.

- There is no mapping for the security between JOnAS and the target operational environment. More specifically, the roles defined for JOnAS cannot be mapped to roles of the target operational environment (for example, groups).

There is one property in the `jonas.properties` file for configuring the security service: the `jonas.security.propagation` property should be set to `true` (which is the default value) to allow the security context to propagate across method calls. Refer to Section 3.3.2 *Security and Transaction Context Propagation*.

3.5.9.1. Using Web Container Tomcat 5.0.x Interceptors for Authentication

With Tomcat 5.0.x, go to the `$JONAS_ROOT/conf/server.xml` file, the `$JONAS_BASE/conf/server.xml` file, the `$CATALINA_HOME/conf/server.xml` file, or the `$CATALINA_BASE/conf/server.xml` file and replace the following line:

```
<Realm className="org.objectweb.jonas.security.realm.web.catalina55.JACC"
        debug="99" resourceName="memrlm_1"/>
```

with this line:

```
<Realm className="org.objectweb.jonas.security.realm.JRealmCatalina41" \
    resourceName="memrlm_1" />
```

A memory, Datasource, or LDAP resource can be used for the authentication, with the correct name of the specified resource as `resourceName` that is: `memrlm_1`, `memrlm_2`, `dsrlm_1`, `ldaprlm_1`, etc.

3.5.9.2. Configuring Mapping Principal/Roles

JOnAS relies on the `jonas-realm.xml` file for access control to the methods of EJB components (refer to `$JONAS_BASE/conf/jonas-realm.xml`).

Example of a secured bean with the role `jonas`:

```
<assembly-descriptor>
    <security-role>
        <role-name>jonas</role-name>
    </security-role>
    <method-permission>
        <role-name>jonas</role-name>
        <method>
            <ejb-name>Bean</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    ...
    ...
</assembly-descriptor>
```

The following subsections describe how to configure the different resources for performing authentication if it is necessary to add a user that has the specified role (jonas) and is authorized to call methods, etc.

3.5.9.2.1. Configuring a Memory Resource in the jonas-realm.xml File

To add the role 'jonas', place this example role in the <roles> section:

```
<roles>
    <role name="jonas" description="Role used in
      the sample security howto" />
</roles>
```

Then, add a user with the specified role. Add a user with the 'jonas' role in the <users> section:

```
<users>
    <user name="jonas_user" password="jonas_password" roles="jonas" />
</users>
```

The <groups> section permits grouping roles. Add the memory resource in the jonas-realm.xml file:

```
<jonas-memoryrealm>
  [...]
  <memoryrealm name="howto_memory_1">
    <roles>
      <role name="jonas" description="Role used in
        the sample security howto" />
    </roles>
    <users>
      <user name="jonas_user" password="jonas_password" roles="jonas" />
    </users>
  </memoryrealm>
  [...]
</jonas-memoryrealm>
```

3.5.9.2.2. Configuring a Datasource Resource in the jonas-realm.xml File

First, build the tables in which the users and roles will be stored.

Example of tables :

realm_users: Add a user jonas_user with the password jonas_password

```
...          ...
jonas_user   jonas_password
...          ...
```

Note that the table can contain more than two columns.

realm_roles: Add the role jonas to the user jonas_user

```
...          ...
jonas_user   jonas
...          ...
```

Now, declare the resource in the jonas-realm.xml file.

The dsName element describes the JNDI name of the datasource to use. Thus, a Datasource configuration with the right JNDI name for the dbm service must be set in the jonas.properties file.

The datasource resource to add in the jonas-realm.xml file is:

```
<jonas-dsrealm>
    [...]
```

```
    <dsrealm name="howto_datasource_realm1"
        dsName="jdbc_1"
        userTable="realm_users" userTableUsernameCol="user_name"
            userTablePasswordCol="user_pass"
        roleTable="realm_roles" roleTableUsernameCol="user_name"
            roleTableRolenameCol="role_name"
     />
    [...]
</jonas-dsrealm>
```

3.5.9.2.3. Configuring an LDAP Resource in the jonas-realm.xml File

The user is added in the LDAP server. In this case, all the users are on the `ou=people,dc=jonas,dc=objectweb,dc=org` DN.

For example, for the user `jonas_user` the unique name will be: DN `uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org`

The role `jonas` will be added on the `ou=groups,dc=jonas,dc=objectweb,dc=org` DN. In this case: DN `cn=jaas,ou=groups,dc=jonas,dc=objectweb,dc=org`

The user is added to the role by adding a field `uniquemember` to the role: `uniquemember = uid=jonas,ou=people,dc=jonas,dc=objectweb,dc=org`

LDIF format for the user:

```
# jonas_user, people, jonas, objectweb, org
dn: uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org
objectClass: inetOrgPerson
uid: jonas_user
sn: jonas_user
cn: JOnAS user
userPassword:: jonas_password
```

LDIF format for the role:

```
# jonas, groups, jonas, objectweb, org
dn: cn=jonas,ou=groups,dc=jonas,dc=objectweb,dc=org
objectClass: groupOfUniqueNames
uniqueMember: uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org
cn: jonas
```

Now the `jonas-realm.xml` file can be customized by adding a LDAP resource.

There are two authentication methods:

- The bind method (default): In order to check the access rights, the resource attempts to login to the LDAP server with the given username and password.
- The compare method: The resource retrieves the password of the user from the LDAP server and compares this password to the password given by the user.

Note

The compare method requires the admin roles in the configuration in order to read the user passwords.

All the elements of the configuration for the LDAP resource can be found in the `jonas-realm_1_0.dtd` DTD file (refer to http://jonas.objectweb.org/current/xml/jonas-realm_1_0.dtd).

For this sample, it is assumed that the LDAP server is on the same computer and is on the default port (389). It takes all the default values of the DTD.

The datasource resource to add in the `jonas-realm.xml` file is:

```
<jonas-ldaprealm>
  [...]
  <ldaprealm name="howto_ldap_realm1"
             baseDN="dc=jonas,dc=objectweb,dc=org" />
  [...]
</jonas-ldaprealm>
```

3.5.9.3. Configuring Client Authentication Based on the Client Certificate in the Web Container

3.5.9.3.1. Introduction

In order to set up the client authentication based on client certificate in a Web container, do the following:

1. Configure the Realm the Web container will have to use.
2. Configure an SSL listener on the Web container.
3. Configure the Web application to make it ask a client certificate.
4. Configure the JAAS LoginModules.
5. Populate the Realm access list.

It is mandatory to possess a X.509 certificate for your Web container on each external interface (IP address) that accepts secure connections. This one can be digitally signed by a Certification Authority or can be autosigned.

3.5.9.3.2. Step 1: Configure the Realm the Web Container Uses

With Tomcat 5.0.x, in the `$JONAS_ROOT/conf/server.xml` file, the `$JONAS_BASE/conf/server.xml` file, the `$CATALINA_HOME/conf/server.xml` file, or the `$CATALINA_BASE/conf/server.xml` file, replace the current Realm by the following:

```
<Realm
  className="org.objectweb.jonas.security.realm.web.catalina55.JAAS" />
```

The class specified uses the JAAS model to authenticate the users. Thus, to choose the resource in which to look for authentication data, configure JAAS.

3.5.9.3.3. Step 2: Configure an SSL Listener on the Web Container

Uncomment the following section in the `server.xml` file:

```
<Connector className="org.apache.catalina.connector.http.HttpConnector"
    port="9043" minProcessors="5" maxProcessors="75" enableLookups="true"
```

```
      acceptCount="10" debug="0" scheme="https" secure="true">
   <Factory className="org.apache.catalina.net.SSLServerSocketFactory"
     clientAuth="false" protocol="TLS"/>
</Connector>
```

☆**Important**

> Set the `clientAuth` parameter to false, otherwise all Web applications will request a client certificate if they need SSL. The client authentication will be configured later in the `web.xml` file in the specific WAR files.

For more information, refer to http://jakarta.apache.org/tomcat/tomcat-5.0-doc/ssl-howto.html.

3.5.9.3.4. Step 3: Configure your Web Application to Request a Client Certificate

Add the following lines to the `web.xml` file of the WAR file of the Web application:

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
    <realm-name>Example Authentication Area</realm-name>
</login-config>
```

☆**Important**

> - Ensure that the restricted Web application area is configured in the `web.xml` file in the WAR file with a `security-constraint` declaration.
> - Ensure that the Web application is always accessed with SSL, unless the Web container will not get a certificate and will not be able to authenticate the user.
> - When authentication is enabled on client certificate, the user's Web browser receives the list of the Certification Authorities trusted by your Web application. A connection will be established with the client only if it has a certificate issued by a trusted Certification Authority, otherwise its certificate will be rejected.
>
>   The certificates of all the trusted Certification Authorities must be imported in the `$JAVA_HOME/jre/lib/security/cacerts` keystore file (customize the SSL listener to modify this location).

3.5.9.3.5. Step 4: Configure the JAAS LoginModules

This authentication process is built on the JAAS technology. This means that authentication is performed in a pluggable way and the desired authentication technology is specified at runtime. Therefore, you must choose the LoginModules to use in your Web container to authenticate the users.

3.5.9.3.5.1. Choose the LoginModules

`org.objectweb.jonas.security.auth.spi.JResourceLoginModule` is the main *LoginModule*. It is highly recommended that this be used in every authentication, as it verifies the user authentication information in the specified resource.

`org.objectweb.jonas.security.auth.spi.JResourceLoginModule` accepts the following parameters:

- `resourceName`: the name of the entry in the `jonas-realm.xml` file being used; this entry represents how and where the authentication information is stored. This is the only required parameter.
- `certCallback`: Specify this optional parameter if you want this login module to ask for a certificate callback. By default it is set to `false`. When using JAAS realms with certificates, set it to `true`.

`org.objectweb.jonas.security.auth.spi.CRLLoginModule` is the LoginModule that contains authentication based on certificates. However, when enabled, it will also permit non-certificate based accesses. It verifies that the certificate presented by the user has not been revoked by the Certification Authority that signed it. To use it, the directory in which to store the revocation lists (CRLs) files or a LDAP repository must exist.

`org.objectweb.jonas.security.auth.spi.CRLLoginModule` can take the following parameters:

- `CRLsResourceName`: this parameter specifies how the CRLs are stored:
  - `Directory`: if the CRL files are stored in a directory on the machine, you must specify another parameter pointing to that directory:
    - `CRLsDirectoryName`: the directory containing the CRL files (the extension for these files must be `.crl`).

  - `LDAP`: *This functionality is experimental* if your CRL files are stored in a LDAP repository, two additional parameters must be specified:
    - `address`: the address of the server that hosts the LDAP repository
    - `port`: the port used by the LDAP repository; CRLs are retrieved from an LDAP directory using the LDAP schema defined in RFC 2587 (refer to http://www.ietf.org/rfc/rfc2587.txt).

3.5.9.3.5.2. Specify the Configuration Parameters

The JAAS configuration sits on a file in which the login module to use for authentication is described. This file is located in `$JONAS_BASE/conf` and named `jaas.config`. To change its location and name, edit the `$JOANS_BASE/bin/jonas.sh` script and modify the following line:

```
-Djava.security.auth.login.config=$JONAS_BASE/conf/jaas.config
```

The contents of the JAAS configuration file follow this structure:

```
Application_1 {
  LoginModuleClassA Flag Options;
  LoginModuleClassB Flag Options;
  LoginModuleClassC Flag Options;
};
Application_2  {
  LoginModuleClassB Flag Options;
  LoginModuleClassC Flag Options;
};
Other {
  LoginModuleClassC Flag Options;
```

```
  LoginModuleClassA Flag Options;
};
```

Sample of a configuration file with a CRL directory:

```
tomcat {
  org.objectweb.jonas.security.auth.spi.CRLLoginModule required
  CRLsResourceName="Directory"
  CRLsDirectoryName="path_to/CRLs";
  org.objectweb.jonas.security.auth.spi.JResourceLoginModule
  required
  resourceName="memrlm_1";
};
```

There can be multiple entries in this file, specifying different configurations that JOnAS can use. The entry dedicated to Tomcat must be named `tomcat`. Note that everything in this file is case-sensitive.

There is a flag associated with all the LoginModules to configure their behavior in case of success or failure:

- `required`: The LoginModule is required to succeed. If it succeeds or fails, authentication still proceeds through the LoginModule list.

- `requisite`: The LoginModule is required to succeed. If it succeeds, authentication continues through the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed through the LoginModule list).

- `sufficient`: The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed through the LoginModule list). If it fails, authentication continues through the LoginModule list.

- `optimal`: The LoginModule is not required to succeed. If it succeeds or fails, authentication still proceeds through the LoginModule list.

3.5.9.3.6. Step 5: Populate the Realm Access List

Now, users will not have to enter a login/password. They will just present their certificates and authentication is performed transparently by the browser (after the user has imported the certificate into it). Therefore, the identity presented to the server is not a login, but a Distinguished Name: that is the name identifying the person to whom the certificate belongs.

This name has the following structure:

```
CN=Someone Unknown, OU=ObjectWeb, O=JOnAS, C=ORG

E  : Email Address
CN : Common Name
OU : Organizational Unit
O  : Organization
L  : Locality
ST : State or Province Name
C  : Country Name
```

The `Subject` in a certificate contains the main attributes and may include additional ones, such as Title, Street Address, Postal Code, Phone Number.

Previously in the `jonas-realm.xml` file a memory realm might contain:

```
<user name="jps_admin" password="admin" roles="administrator"/>
```

To enter a certificate-based user access, you must now enter the user's DN preceded by the String: `##DN##`, as shown in the following example:

```
<user name="##DN##CN=whale, OU=ObjectWeb, O=JOnAS, L=JOnAS, ST=JOnAS,
   C=ORG" password="" roles="jadmin" />
```

3.5.10. Configuring the JMS Service

Until JOnAS release 4.1, the only way to provide the JMS facilities was by setting a JMS service when configuring JOnAS. Now in the default setting, the JMS service is disabled in the `jonas.properties` configuration file. Note that this JMS service does not allow deployment of 2.1 MDBs *and will become deprecated*.

The new way to integrate a JMS platform is by deploying a J2EE 1.4 compliant resource adapter. How you do this is described in Section 3.8 *Configuring JMS Resource Adapters*.

JOnAS integrates a third-party JMS implementation, JORAM (http://www.objectweb.org/joram/), which is the default JMS service. Other JMS providers, such as SwiftMQ (http://www.swiftmq.com/ and WebSphere MQ (http://www-3.ibm.com/software/integration/mqfamily/), can easily be integrated as JMS services.

The JMS service is used to contact (or launch) the corresponding MOM (*Message Oriented Middleware*) or `JMS server`. You should create the JMS-administered objects used by the EJB components, such as the connection factories and the destinations, prior to the EJB execution, using the proprietary JMS implementation administration facilities. JOnAS provides "wrappers" on such JMS administration APIs, which enable the EJB server itself to perform simple administration operations automatically.

The JMS service is an optional service that must be started before the EJB container service.

The following are the properties that can be set in `jonas.properties` file for the JMS service:

- `jonas.service.jms.collocated` for setting the `JMS server` launching mode. If set to `true`, it is launched in the same JVM as the JOnAS Server (this is the default value). If set to `false`, it is launched in a separate JVM, in which case the `jonas.service.jms.url` must be set with the connection URL to the JMS server.

- `jonas.service.ejb.minworkthreads` is used for setting the minimum number of threads in the pool.

- `jonas.service.ejb.maxworkthreads` is used for setting the maximum number of threads in the pool.

- `jonas.service.jms.queues` and `jonas.service.jms.topics` are used for setting lists of administered objects `queues` or `topics` at launching time.

- `jonas.service.jms.mom` is used to indicate which class must be used to perform administrative operations. This class is the wrapper to the actual JMS provider implementation. The default class is `org.objectweb.jonas_jms.JmsAdminForJoram`, which is required for JORAM (refer to http://joram.objectweb.org//). For the SwiftMQ product, obtain a `com.swiftmq.appserver.jonas.JmsAdminForSwiftMQ` class from the SwiftMQ site (http://www.swiftmq.com/).

Some additional information about JMS configuration (in particular, several JORAM advanced configuration aspects) is provided in Section 26.4 *JMS Administration* and Section 26.5 *Running an EJB Performing JMS Operations*.

3.5.11. Configuring the Resource Service

The `Resource service` is an optional service that must be started as soon as EJB components require access to an external Enterprise Information Systems. The standard way to do this is to use a third-party software component called a **Resource Adapter**.

The role of the `Resource service` is to deploy the **Resource Adapters** in the JOnAS server, (that is, configure it in the operational environment and, in JNDI name space, register a connection factory instance that can be looked up by the EJB components).

The `Resource service` can be configured in one of the following ways:

- The corresponding RAR file name is listed in the `jonas.service.resource.resources` property in `jonas.properties` file. If the file name does not contain an absolute path name, then it should be located in the `$JONAS_BASE/rars/` directory.

  Example:
  ```
  jonas.service.resource.resources  MyEIS
  ```

  This file will be searched for in the `$JONAS_BASE/rars/` directory. This property is a comma-separated list of `resource adapter` file names (the `.rar` suffix is optional).
- Another way to automatically deploy `resource adapter` files at the server start-up is to place the RAR files in an `autoload` directory. The name of this directory is specified using the `jonas.service.resource.autoloaddir` property in `jonas.properties` file. This directory is relative to the `$JONAS_BASE/rars/` directory.

A *jonas-specific, resource-adapter configuration* xml file must be included in each resource adapter. This file replicates the values of all configuration properties declared in the deployment descriptor for the resource adapter. Refer to Section 25.2 *Defining the JOnAS Connector Deployment Descriptor* for additional information.

3.5.12. Configuring the JMX (Java Management Extension) Service

The `JMX service` is mandatory and will be started even if it is not present in the list of services. It is currently based on MX4J JMX implementations, and does not require any specific configuration.

3.5.13. Configuring the Mail Service

The `Mail service` is an optional service that can be used to send email. It is based on JavaMail and on the JavaBeans Activation Framework (JAF) API (refer to http://java.sun.com/products/javamail/ and http://java.sun.com/products/beans/glasgow/jaf.html respectively).

A mail factory is required in order to send or receive mail. JOnAS provides two types of mail factories: `javax.mail.Session` and `javax.mail.internet.MimePartDataSource`.

`MimePartDataSource` factories allow mail to be sent with a subject and the recipients already set.

Mail factory objects must be configured accordingly to their type. The subsections that follow briefly describe how to configure `Session` and `MimePartDataSource` mail factory objects, in order to run the `SessionMailer` SessionBean and the `MimePartDSMailer` SessionBean delivered with the platform.

3.5.13.1. Configuring a Session Mail Factory

The template `MailSession1.properties` file supplied in the installation directory defines a mail factory of type `Session`. The JNDI name of the mail factory object is `mailSession_1`. This template must be updated with values appropriate to your installation.

Refer to `$JONAS_BASE/conf/MailSession1.properties` for a sample of the file and Section 3.5.13.4 *Configuring a Mail Factory* for the list of available properties.

3.5.13.2. Configuring MimePartDataSource Mail Factory

The template MailMimePartDS1.properties file supplied in the installation directory defines a mail factory of `MimePartDSMailer` type. The JNDI name of the mail factory object is `mailMimePartDS_1`. This template must be updated with values appropriate to your installation.

Refer to `$JONAS_BASE/conf/MailMimePartDS1.properties` for a sample of the file and Section 3.5.13.4 *Configuring a Mail Factory* for a list of the available properties.

3.5.13.3. Configuring JOnAS for a Mail Factory

Mail factory objects created by JOnAS must be given a name. In the `mailsb` example, two factories called `MailSession1` and `MailMimePartDS1` are defined.

Each factory must have a configuration file whose name is the name of the factory with the `.properties` extension (`MailSession1.properties` for the MailSession1 factory).

Additionally, the `jonas.properties` file must define the `jonas.service.mail.factories` property. For this example, it is:

```
jonas.service.mail.factories MailSession1,MailMimePartDS1
```

3.5.13.4. Configuring a Mail Factory

A mail factory has the following required properties:

mail.factory.name	JNDI name of the mail factory
mail.factory.type	The type of the factory. This property can be `javax.mail.Session` or `javax.mail.internet.MimePartDataSource`.

Table 3-1. Required properties

A mail factory has the following optional authentication properties:

mail.authentication.username	Set the username for the authentication.
mail.authentication.password	Set the password for the authentication.

Table 3-2. Optional Authentication properties

The `javax.mail.Session.properties` file has the properties:

mail.authentication.username	Set the username for the authentication.
mail.authentication.password	Set the password for the authentication.
mail.debug	The initial debug mode. Default is `false`.

mail.from	The return email address of the current user, used by the `InternetAddress` method `getLocalAddress`.
mail.mime.address.strict	The MimeMessage class uses the `InternetAddress` method `parseHeader` to parse headers in messages. This property controls the strict flag passed to the `parseHeader` method. The default is `true`.
mail.host	The default host name of the mail server for both Stores and Transports. Used if the `mail.protocol.host` property is not set.
mail.store.protocol	Specifies the default message access protocol. The `Session` method `getStore()` returns a Store object that implements this protocol. By default the first Store provider in the configuration files is returned.
mail.transport.protocol	Specifies the default message access protocol. The `Session` method `getTransport()` returns a Transport object that implements this protocol. By default, the first Transport provider in the configuration files is returned.
mail.user	The default user name to use when connecting to the mail server. Used if the `mail.protocol.user` property is not set.
mail.<protocol>.class	Specifies the fully- qualified class name of the provider for the specified protocol. Used in cases where more than one provider for a given protocol exists; this property can be used to specify which provider to use by default. The provider must still be listed in a configuration file.
mail.<protocol>.host	The host name of the mail server for the specified protocol. Overrides the `mail.host` property.
mail.<protocol>.port	The port number of the mail server for the specified protocol. If it is not specified, the protocol's default port number is used.
mail.<protocol>.user	The user name to use when connecting to mail servers using the specified protocol. Overrides the `mail.user` property.

Table 3-3. `javax.mail.Session.properties`

(Refer to JavaMail documentation at http://java.sun.com/products/javamail/1.3/docs/javadocs/overview-summary.html for more information.)

mail.to	Set the list of primary recipients ("to") of the message.
mail.cc	Set the list of Carbon Copy recipients ("cc") of the message.
mail.bcc	Set the list of Blind Carbon Copy recipients ("bcc") of the message.
mail.subject	Set the subject of the message.

Table 3-4. MimePartDataSource properties (used only if mail.factory.type is javax.mail.internet.MimePartDataSource)

3.6. Configuring the DB Service (hsql)

The *DB service* is an optional service that can be used to start a Java database server in the same JVM as JOnAS.

The listening port and the database name can be set as follows:

```
jonas.service.db.port    9001
jonas.service.db.dbname  db_jonas
```

You can use the `$JONAS_ROOT/conf/HSQL1.properties` file with these default values.

The users are declared as follows:

```
jonas.service.db.user<1..n>   login:password
```

For example, to give access to this database to the user jonas with the password jonas, use:

```
jonas.service.db.user1    jonas:jonas
```

This login and this password (jonas/jonas) are used in the `HSQL1.properties` file.

3.7. Configuring the Discovery Service

The discovery service was added to JOnAS services in order to support Chapter 41 *Domain Management in JOnAS*.

In order to enable domain management, all the servers in the domain must start the discovery service by adding `discovery` in the services list (`joanas.services` property).

By default, a server running the discovery service is a *slave*, but the discovery service can be configured to make a server play the role of *master*. All the *slave* servers in the domain can be administered by a management application deployed on a *master* server. However, a management application can be deployed on a *slave* in order to administer it individually.

The discovery service is based on a communication infrastructure (currently IP Multicast) that enables message broadcasting in order to *publish* the connector-server addresses and state changes of the servers in the domain. The *master* servers listen to these *discovery* messages. Moreover, they are able to broadcast messages asking the servers in the domain to identify themselves. See Chapter 35 *Howto: JSR 160 Support in JOnAS* for more details about JMX Remote connectors.

3.7.1. Configuration Settings for All Servers

The default configuration of the discovery service provides the IP Multicast configuration. The following two properties must have identical values for all the servers in the domain:

```
jonas.service.discovery.multicast.address = 224.224.224.224
jonas.service.discovery.multicast.port = 9080
```

3.7.2. Additional Configuration Settings for Master Servers

To start a server as a *master*, you must uncomment the following line:

```
jonas.service.discovery.master = true
```

The configuration of a *master* server must also define an additional port number by setting the following property:

```
jonas.service.discovery.source = 9888
```

3.7.3. Master or Slave?

In a management domain, at least one server must be started as a *master*. But, any *slave* server can be dynamically promoted to be a *master*. This is allowed by a management operation exposed by the MBean associated with the discovery service (`startDiscoveryMaster` operation).

3.8. Configuring JMS Resource Adapters

Instead of using the JOnAS "JMS Service" for configuring a JMS platform, it is possible to use the JOnAS "Resource Service" and JMS adapters that are compliant with the J2EE Connector Architecture specification. The provided functionalities are the same, with the extra benefit of allowing the deployment of 2.1 MDBs.

JMS connections are obtained from a JMS Resource Adapter (RA), which is configured to identify a JMS server and access it. Multiple JMS RAs can be deployed, either via the `jonas.properties` file, or via the `JonasAdmin` tool, or included in the autoload directory of the resource service. For complete information about RAs in JOnAS, refer to Chapter 25 *JOnAS and the Connector Architecture*.

3.8.1. JORAM Resource Adapter

This section describes how JMS Resource Adapters should be configured to provide messaging functionalities to JOnAS components and applications.

The JORAM resource adapter archive (`joram_for_jonas_ra.rar`) is provided with the JOnAS distribution. It can be found in the `$JONAS_ROOT/rarsautoload` directory and thus is deployed automatically.

Note that `jms` *must* not appear in the list of services of the `jonas.properties` file because the JORAM's RAR and the JMS service are exclusive.

```
jonas.services registry,jmx,jtm,dbm,security,resource,ejb,web,ear
```

The JORAM RA may be seen as the central authority to go through for connecting and using a JORAM platform. The RA is provided with a default deployment configuration that:

- Starts a `collocated` JORAM server in `non-persistent` mode, with id `0` and name `s0`, on host `localhost` and using port `16010`; for doing so it relies on both an Section 3.8.1.1.1 *JORAM server configuration: a3server.xml* file located in the `$JONAS_ROOT/conf` directory and the Section 3.8.1.1.2 *Resource Adapter Configuration: ra.xml, jonas-ra.xml* file located within the RA.
- Creates managed JMS `ConnectionFactory` instances and binds them with the names `CF`, `QCF`, and `TCF`.
- Creates administered objects for this server (JMS *destinations* and non-managed *factories*) as described by the Section 3.8.1.2 *Administration Tasks: joramAdmin.xml* file, located in the `$JONAS_BASE/conf` directory; those objects are bound with the names `sampleQueue`, `sampleTopic`, `JCF`, `JQCF`, and `JTCF`

This default behavior is strictly equivalent to the default JMS service's behavior.

Of course, you can modify the default configuration.

3.8.1.1. Configuration Files

JORAM in JOnAS is composed of three different parts: the server, the RA, and the administration. Each part contains its own configuration files:

- `a3server.xml` is the JORAM platform configuration file (that is, the server part). The file is located in the `$JONAS_BASE/conf` directory. (See Section 3.8.1.1.1 *JORAM server configuration: a3server.xml*.)

- `ra.xml` and `jonas-ra.xml` are the resource adapter configuration files. They are embedded in the resource adapter (META-INF directory). (See Section 3.8.1.1.2 *Resource Adapter Configuration: ra.xml, jonas-ra.xml*.)

- `joramAdmin.xml` contains the administration tasks to perform by the JORAM server, such as the JMS objects to create. It is located in the `$JONAS_BASE/conf` directory. (See Section 3.8.1.2 *Administration Tasks: joramAdmin.xml*.)

3.8.1.1.1. JORAM server configuration: a3server.xml

The `a3server.xml` (`$JONAS_BASE/conf/a3server.xml`) file describes the JORAM platform (that is, the network domain, the used transport protocol, and the reachable JORAM servers). It is used by a starting JORAM server. By default, only one collocated JORAM server is defined (s0) based on the TCP/IP protocol. A distributed configuration example is provided in Chapter 48 *Distributed Message Beans in JOnAS 4.1*; others examples are available in the *JORAM's User Guide* (see http://joram.objectweb.org/current/doc/Joram-4.3-en.pdf).

The default file provided with JOnAS is the following:

```
<?xml version="1.0"?>
<config>
    <server id="0" name="S0" hostname="localhost">
      <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
                args="root root"/>
      <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
                args="16010"/>
    </server>
</config>
```

The above configuration describes a JORAM platform made up of one unique JORAM server (id 0, name s0), running on localhost, listening on port 16010. Those values are taken into account by the JORAM server when starting. However, *they should match the values set in the deployment descriptor of the RA*, otherwise the adapter either will not connect to the JORAM server, or it will build improper connection factories. The Section 3.8.1.2.1 *joram_raconfig command* enables you to modify these parameters in all the configuration files.

If used in non-collocated mode, the `$JONAS_BASE/conf/a3server.xml` file is not used.

3.8.1.1.2. Resource Adapter Configuration: ra.xml, jonas-ra.xml

The `ra.xml` file is the standard deployment descriptor for the JORAM adapter and the `jonas-ra.xml` file is the JOnAS-specific deployment descriptor for the JORAM adapter. These files set the central configuration of the adapter, define and set managed connection factories for outbound communication, and define a listener for inbound communication. `jonas-ra.xml` contains specific parameters such as pool parameters or JNDI names, but also may redefine some of the `ra.xml` file's parameters and override their values. Globally, a good way to proceed is to keep the original `ra.xml` file with the default values and to customize the configuration only in the `jonas-ra.xml` file.

Changing the configuration of the RA requires extracting and editing the deployment descriptor and updating the archive file. There is several possible ways to do this:

- With the **RAConfig** utility (refer to Section 6.7 *RAConfig* for a complete description).

  To extract `jonas-ra.xml`, run:
  `RAConfig -path . joram_for_jonas_ra.rar`

  Then, to update the archive:
  `RAConfig -u jonas-ra.xml joram_for_jonas_ra.rar`

- Through the **jonasAdmin** console:

  In the jonasAdmin's tree, the `Resource Adapter Module` node (under the `deployment` node) contains a `configure` tab that enables you to edit both the `ra.xml` and the `jonas-ra.xml` files of the undeployed RA.

- Through the **joram_raconfig** utility (refer to Section 3.8.1.2.1 *joram_raconfig command* for a complete description).

  This tool enables you to easily modify the network parameters of the JORAM server in all the configuration files.

The following properties are related to the central configuration of the adapter:

Property Name	Description	Possible Values
CollocatedServer	Running mode of the JORAM server to which the adapter gives access.	True: when deploying, the adapter starts a collocated JORAM server. False: when deploying, the adapter connects to a remote JORAM server. Nothing (default True value is then set).
PlatformConfigDir	Directory where the `a3servers.xml` and `joramAdmin.xml` files are located.	Any String describing an absolute path (for example: /myHome/myJonasRoot/conf). Empty String, files will be searched in `$JONAS_BASE/conf`. Nothing (default empty string is then set).
PersistentPlatform	Persistence mode of the collocated JORAM server (not taken into account if the JORAM server is set as non-collocated).	True: starts a persistent JORAM server. False: starts a non-persistent JORAM server. Nothing (default False value is then set).
ServerId	Identifier of the JORAM server to start (not taken into account if the JORAM server is set as non-collocated).	Identifier corresponding to the server to start described in the `a3servers.xml` file (ex: 1). Nothing (default 0 value is then set).

Property Name	Description	Possible Values
ServerName	Name of the JORAM server to start (not taken into account if the JORAM server is set as non-collocated).	Name corresponding to the server to start described in the `a3servers.xml` file (ex: s1). Nothing (default s0 name is then set).
AdminFileXML	Name of the file describing the administration tasks to be performed by the JORAM server; that is, JMS destinations to create, users to create, and so on. If the file does not exist, or is not found, no administration task is performed.	Name of the file (ex: myAdminFile.xml). Nothing (default `joramAdmin.xml` name is then set).
HostName	Name of the host where the JORAM server runs, used for accessing a remote JORAM server (non collocated mode), and for building appropriate connection factories.	Any host name (ex: myHost). Nothing (default localhost name is then set).
ServerPort	Port the JORAM server is listening on, used for accessing a remote JORAM server (non-collocated mode), and for building appropriate connection factories.	Any port value (ex: 16030). Nothing (default 16010 value is then set).

Table 3-5. Adapter Configuration Properties

The `jonas-connection-definition` tags wrap properties related to the managed connection factories:

Property Name	Description	Possible Values
jndi-name	Name used for binding the constructed connection factory.	Any name (such as myQueueConnectionFactory).
UserName	Default user name that will be used for opening JMS connections.	Any name (such as myName). Nothing (default anonymous name will be set).

Property Name	Description	Possible Values
UserPassword	Default user password that will be used for opening JMS connections.	Any name (such as myPass). Nothing (default anonymous password will be set).
Collocated	Specifies if the connections that will be created from the factory should be TCP or local-optimized connections (the collocated mode can only be set if the JORAM server is collocated; such factories will only be usable from within JOnAS).	True (for building local-optimized connections). False (for building TCP connections). Nothing (default TCP mode will be set).

Table 3-6. jonas-connection-definition Tags

The `jonas-activationspec` tag wraps a property related to inbound messaging:

Property Name	Description	Possible Values
jndi-name	Binding name of a JORAM object to be used by 2.1 MDBs.	Any name (such as joramActivationSpec).

Table 3-7. jonas-activationspec Tag

The common parameters are described in Section 3.9 *Common Parameters*.

3.8.1.2. Administration Tasks: joramAdmin.xml

The `joramAdmin.xml` file (formerly the `joram-admin.cfg` file) describes the configuration relating to the application. It describes the administration objects in the JORAM server such as the JMS objects, the users, or the non-managed factories. In others words, it defines the JORAM objects to be (optionally) created when deploying the adapter.

The new file format enables new possibilities compared to the former version (`joram-admin.cfg`), such as the possibility to specify a different name for the JMS destination and its JNDI name, or the ability to define dead message queues, cluster queues, scheduler queues, and so on.

If the two files are present in the configuration (`joram-admin.cfg`, `joramAdmin.xml`), the two are loaded and you have to take care of the object conflicts; otherwise, a failure will occur. Note that `joram-admin.cfg` is deprecated now.

A dynamic reloading can be performed through the class org.objectweb.joram.client.jms.admin.AdminModule.executeXMLAdmin() method.

The default file provided with JOnAS creates a queue bound with the name `sampleQueue` and a topic bound with name `sampleTopic`, sets the anonymous `user`, and creates and binds non-managed connection factories named `JCF`, `JQCF`, and `JTCF`.

◈ **Note:**

- All administration tasks are performed locally (that is, on the JORAM server to which the adapter is connected).
- If a queue, a topic, or a user already exists on the JORAM server (for example because the server is in persistent mode and has re-started after a crash, or because the adapter has been deployed, undeployed, and is re-deployed giving access to a remote JORAM server), it will be retrieved instead of being re-created.

The new format is XML; a simple example is given below:

```
<?xml version="1.0"?>
<JoramAdmin>
  <AdminModule>
    <collocatedConnect name="root" password="root"/>
  </AdminModule>
  <ConnectionFactory
    className="org.objectweb.joram.client.jms.tcp.TcpConnectionFactory">
    <tcp host="localhost"
         port="16010"/>
    <jndi name="JCF"/>
  </ConnectionFactory>
  <ConnectionFactory className="org.objectweb.joram.client.jms.tcp.QueueTcpConnecti
    <tcp host="localhost"
         port="16010"/>
    <jndi name="JQCF"/>
  </ConnectionFactory>
  <ConnectionFactory className="org.objectweb.joram.client.jms.tcp.TopicTcpConnecti
    <tcp host="localhost"
         port="16010"/>
    <jndi name="JTCF"/>
  </ConnectionFactory>
  <User name="anonymous"
        password="anonymous"
        serverId="0"/>
  <Queue name="sampleQueue">
    <freeReader/>
    <freeWriter/>
    <jndi name="sampleQueue"/>
  </Queue>
  <Topic name="sampleTopic">
    <freeReader/>
    <freeWriter/>
    <jndi name="sampleTopic"/>
  </Topic>
</JoramAdmin>
```

To request the creation of a new object, simply add the element in the file. For example, to add a queue 'MyQueue', add the following XML element:

```
<Queue name="myQueue">
  <freeReader/>
  <freeWriter/>
  <jndi name="myQueue"/>
</Queue>
```

When the JORAM is not collocated, the AdminModule must be defined as follows:

```
<AdminModule>
```

```
      <connect hostName="localhost"
              port="16020"
              name="root"
              password="root"/>
</AdminModule>
```

Possible parameters for a queue definition:

```
<Queue name=""
          serverId=""
          className=""
          dmq=""
          nbMaxMsg=""
          threshold="">
  <property name="" value=""/>
  <property name="" value=""/>
  <reader user=""/>
  <writer user=""/>
  <freeReader/>
  <freeWriter/>
  <jndi name=""/>
  </Queue>
```

Possible parameters for a topic definition:

```
<Topic name=""
          parent=""
          serverId=""
          className=""
          dmq="">
  <property name="" value=""/>
  <property name="" value=""/>
  <reader user=""/>
  <writer user=""/>
  <freeReader/>
  <freeWriter/>
  <jndi name=""/>
</Topic>
```

Example of a dead message queue definition:

```
<DMQueue name="DMQ"
             serverId="0">
  <reader user="anonymous"/>
  <writer user="anonymous"/>
  <freeReader/>
  <freeWriter/>
  <jndi name="DMQ"/>
</DMQueue>
```

Example of a scheduler queue definition:

```
<Destination type="queue"
      serverId="0"
      name="schedulerQueue"
      className="com.scalagent.joram.mom.dest.scheduler.SchedulerQueue">
  <freeReader/>
  <freeWriter/>
  <jndi name="schedulerQueue"/>
</Destination>
```

Example of a clustered queues destination:

```
<Cluster>
  <Queue name="queue0"
            serverId="0"
            className="org.objectweb.joram.mom.dest.ClusterQueue">
    <freeReader/>
    <freeWriter/>
    <property name="period" value="10000"/>
    <property name="producThreshold" value="50"/>
    <property name="consumThreshold" value="2"/>
    <property name="autoEvalThreshold" value="false"/>
    <property name="waitAfterClusterReq" value="1000"/>
    <jndi name="queue0"/>
  </Queue>
  <Queue name="queue1"
            serverId="1"
            className="org.objectweb.joram.mom.dest.ClusterQueue">
    <freeReader/>
    <freeWriter/>
    <property name="period" value="10000"/>
    <property name="producThreshold" value="50"/>
    <property name="consumThreshold" value="2"/>
    <property name="autoEvalThreshold" value="false"/>
    <property name="waitAfterClusterReq" value="1000"/>
    <jndi name="queue1"/>
  </Queue>
  <Queue name="queue2"
            serverId="2"
            className="org.objectweb.joram.mom.dest.ClusterQueue">
    <freeReader/>
    <freeWriter/>
    <property name="period" value="10000"/>
    <property name="producThreshold" value="50"/>
    <property name="consumThreshold" value="2"/>
    <property name="autoEvalThreshold" value="false"/>
    <property name="waitAfterClusterReq" value="1000"/>
    <jndi name="queue2"/>
  </Queue>
  <freeReader/>
  <freeWriter/>
  <reader user="user0"/>
  <writer user="user0"/>
  <reader user="user1"/>
  <writer user="user1"/>
  <reader user="user2"/>
  <writer user="user2"/>
</Cluster>
```

Other samples are provided with the JORAM source code.

3.8.1.2.1. joram_raconfig command

The **joram_raconfig** tool aims to facilitate changes of the parameters (host, port, server id) in JORAM's configuration. JORAM relies on several configuration files: `a3server.xml`, `joramAdmin.xml`, and `jonas-ra.xml`. With **joram_raconfig**, all of these configuration files are updated, and thus the consistency is ensured.

Usage:

```
joram_raconfig [-p <port>] [-h <host>] [-s <serverid>]
       with
       port     : listening port of the JORAM server
       host     : IP address of the JORAM server
       serverid : server id of the JORAM server
```

3.8.1.3. JORAM's Resource Adapter Tuning

3.8.1.3.1. ManagedConnection Pool

A pool of ManagedConnection is defined for each factory (connection definition) specified in the `ra.xml` file. See the pool parameters in Section 3.9.1 *ConnectionManager Pool Configuration*.

3.8.1.3.2. Session/Thread Pool in the JORAM RA

The JORAM RA manages a pool of session/thread for each connection; by default, the maximum number of parallel session is set to 10.

When linked with an MDB, this maximum number of entries in the pool corresponds to the maximum number of messages that can be processed in parallel per MDB. A session is released to the pool just after the message processing (onMessage()). When the maximum is reached, the inquiries for a session creation are blocked until a session in the pool becomes available.

You can set the maxNumberOfWorks property in the MDB JOnAS's specific deployment descriptor. For example, you can add the code below to limit the number of parallel session to 100 (default value is 10):

```
<activation-config-property>
<activation-config-property-name>maxNumberOfWorks</activation-config-property-name
<activation-config-property-value>10</activation-config-property-value>
</activation-config-property>
```

3.8.1.4. Undeploying and Redeploying a JORAM Adapter

Undeploying a JORAM adapter either stops the collocated JORAM server, or disconnects from a remote JORAM server. It is then possible to deploy the same adapter again:

- If set for running a collocated server, it will re-start it.
- If the running mode is persistent, then server will be retrieved in its pre-undeployed state (with the existing destinations, users, and possibly messages).
- If set for connecting to a remote server, the adapter will reconnect and access the destinations it previously created.

In the collocated persistent case, if you want to start a new JORAM server, remove its persistence directory. This directory is located in JOnAS' running directory, and has the same name as the JORAM server (for example `s0/` for server "s0").

3.9. Common Parameters

3.9.1. ConnectionManager Pool Configuration

The JOnAS resource service creates a connection manager for each connection definition described in the deployed RAR file. Each connection manager instance owns its managed connection pool.

Only one connection definition can be defined in a JCA 1.0-compliant RAR, while the JCA 1.5 enables you to define several connection definitions for a single RAR file.

The pool parameters may be defined in the `jonas-ra.xml` file (embedded in the resource adapter archive) either at the root level (jonas-connector element) or at the connection definition level (see the XML schema). The values defined at the connection-definition level override the values defined at the root level.

property name	description	default value
pool-init	Initial number of connections	0
pool-min	Minimum number of connections	0
pool-max	Maximum number of connections	-1 (unlimited)
pool-max-age	Number of seconds to keep the connection in the pool	0 (unlimited)
pstmt-max	Maximum number of PreparedStatements cached per connection. -1 to disable	10
pool-max-opentime	Maximum open time in minutes for a connection in the pool. The timed out active connection is automatically closed.	0 (infinite)
pool-max-waiters	Maximum number of waiters (threads) allowed to wait for a Connection. When the threshold is reached, an exception 'no more connections' is thrown.	0

property name	description	default value
pool-max-waittime	Maximum number of seconds to wait for a connection when the pool is full. When a timeout occurs, an exception 'no more connections' is thrown. If 0 is set, the default value is used.	10 (seconds)
pool-sampling-period	Pool statistics sampling period in seconds.	30 (seconds)

Table 3-8. pool-params elements

Examples:

- The `jonas-ra.xml` file with a pool definition at the jonas-connector level:

```
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
   ...
   <pool-params>
      <pool-init>0</pool-init>
      <pool-min>0</pool-min>
      <pool-max>100</pool-max>
      <pool-max-age>0</pool-max-age>
      <pstmt-max>0</pstmt-max>
      <pool-max-opentime>0</pool-max-opentime>
      <pool-max-waiters>0</pool-max-waiters>
      <pool-max-waittime>0</pool-max-waittime>
      <pool-sampling-period>30</pool-sampling-period>
   </pool-params>
   ...
</jonas-connector>
```

- The `jonas-ra.xml` file with a pool definition at the connection-definition level:

```
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
   ...
   <jonas-connection-definition>
      <description>...</description>
      <jndi-name>...</jndi-name>
      <jonas-config-property>
      ...
      </jonas-config-property>
      ...
      <pool-params>
         <pool-init>0</pool-init>
         <pool-min>0</pool-min>
         <pool-max>100</pool-max>
         <pool-max-age>0</pool-max-age>
         <pstmt-max>0</pstmt-max>
         <pool-max-opentime>0</pool-max-opentime>
         <pool-max-waiters>0</pool-max-waiters>
         <pool-max-waittime>0</pool-max-waittime>
         <pool-sampling-period>30</pool-sampling-period>
      </pool-params>
         ...
   </jonas-connection-definition>
</jonas-connector>
```

3.9.2. RAR Link

The **jonas-ra.xml** may contain the optional `rarlink` element. It allows the reuse of a base resource-adapter definition overriding only a subset of the specific parameters. Typically, it enables the definition of a JDBC resource adapter relying on the generic resource adapter provided with JOnAS. In this case, the new RA package may only contain the `META-INF/jonas-ra.xml` file.

Property Name	Description	Default Value
rarlink	JNDI name of the base RA	none

Table 3-9. rarlink Element

Example:

This is the `jonas-ra.xml` file content with a RAR link to the generic JDBC RAR 'JOnASJDBC_DM':

```
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
    ...
    <rarlink>JOnASJDBC_DM</rarlink>
    ...
</jonas-connector>
```

3.9.3. RAR Logging

The `jonas-ra.xml` may contain the optional parameters `log-enabled` and `log-topic` in order to define the logging properties for the resource adapter. `log-enabled` specifies whether the logging has to be turned on for the resource adapter software. If yes, by default the used topic is 'org.objectweb.jonas.jca', but this value may be overriden through the second parameter, `log-topic`.

Property Name	Description	Default Value
log-enabled	Turn on/off the logging for the resource adapter	false
log-topic	Logging topic used for the resource adapter logging	org.objectweb.jonas.jca

Table 3-10. Logging Elements

Example:

`jonas-ra.xml` file content with logging parameters:

```
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
    ...
    <log-enabled>true</log-enabled>
    <log-topic>mypackage</log-topic>
    ...
</jonas-connector>
```

Chapter 4.

Configuring JDBC DataSources

This chapter shows the bean or application deployer how to configure the DataSources to connect the application to databases.

4.1. Configuring DataSources

For both container-managed and bean-managed persistence, JOnAS makes use of relational storage systems through the JDBC interface. JDBC connections are obtained from an object, the `DataSource`, provided at the application server level. The DataSource interface is defined in the JDBC 2.0 standard extensions (see http://java.sun.com/products/jdbc/). A DataSource object identifies a database and a means to access it via a JDBC driver. An application server may request access to several databases and thus provide the corresponding DataSource objects. Available DataSource objects can be added on the platform; they must be defined in the `jonas.properties` file. This section explains how DataSource objects can be defined and configured in the JOnAS server.

To support distributed transactions, JOnAS requires the use of a JDBC2-XA-compliant driver. Such drivers that implement the `XADataSource` interface are not always available for all relational databases. JOnAS provides a generic *driver-wrapper* that emulates the XADataSource interface on a regular JDBC driver.

> ⭐ **Important**
>
> It is important to note that this driver-wrapper does not ensure a real two-phase commit for distributed database transactions.

JOnAS's generic `driver-wrapper` provides an implementation of the DataSource interface that allows DataSource objects to be defined using a JDBC1-compliant driver for some relational databases, such as Oracle and PostgreSQL.

Neither the EJB specification nor the J2EE specification describe how to define DataSource objects so that they are available to a J2EE platform. Therefore, this document, which describes how to define and configure DataSource objects, is *specific to JOnAS*. However, the way to use these DataSource objects in the Application Component methods is standard; that is, by using the resource manager connection factory references (refer to the example in Section 8.6 *Writing Database Access Operations (Bean-Managed Persistence)*).

A DataSource object should be defined in a file called `DataSourcename.properties` (for example, `Oracle1.properties` for an Oracle DataSource), as delivered with the platform.

In the `jonas.properties` file, to define a DataSource "Oracle1", add the name `Oracle1` (the same name as in the properties file) to the line `jonas.service.dbm.datasources`, as follows:

```
jonas.service.dbm.datasources Oracle1,PostgreSQL
```

The property file defining a DataSource should contain the following information:

datasource.name	JNDI name of the DataSource
datasource.url	The JDBC database URL: jdbc:*database_vendor_subprotocol*:...
datasource.classname	Name of the class implementing the JDBC driver
datasource.username	Database user name
datasource.password	Database user password

A DataSource object for Oracle (for example, Oracle1), named `jdbc_1` in JNDI, and using the Oracle *thin* JDBC driver, would be described in a file called `Oracle1.properties`, as in the following example:

```
datasource.name       jdbc_1
datasource.url        jdbc:oracle:thin:@malte:1521:ORA1
datasource.classname  oracle.jdbc.driver.OracleDriver
datasource.username   scott
datasource.password   tiger
```

In this example, `malte` is the hostname of the server running the Oracle DBMS, `1521` is the SQL*Net V2 port number on this server, and `ORA1` is the `ORACLE_SID`.

This example makes use of the Oracle "Thin" JDBC driver. If your application server is running on the same host as the Oracle DBMS, you can use the Oracle OCI JDBC driver; depending on the Oracle release. The URL to use for this would be jdbc:oracle:oci7, or jdbc:oracle:oci8. Oracle JDBC drivers can be downloaded at the Oracle web site: http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html

To create a PostgreSQL DataSource object named `jdbc_3` in JNDI, describe it as follows (in a file `PostGreSQL.properties`):

```
datasource.name       jdbc_3
datasource.url        jdbc:postgresql://your_host/your_db
datasource.classname  org.postgresql.Driver
datasource.username   useless
datasource.password   useless
```

Properties having the `useless` value are not used for this type of persistence storage.

If the database user and password are placed in the DataSource description (the *DataSource-name*.properties file), then Application Components should use the `getConnection()` method. If the database user and password are omitted from the DataSource description file, then Application Components should use the `getConnection(String username, String password)` method. The resource reference of the associated datasource in the standard deployment descriptor, the <res-auth> element, should have the corresponding value: `Container` if the user and password are given in the DataSource description, `Application` if the user and password are provided by the Application.

4.2. CMP2.0/JORM

To implement EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework (see http://jorm.objectweb.org/index.html). JORM must adapt its object-relational mapping to the underlying database and makes use of adapters called "mappers" for this purpose. Thus, for each type of database (and more precisely for each JDBC driver), the corresponding mapper must be specified in the DataSource properties file. This is the purpose of the `datasource.mapper` property.

The JORM database mapper named `datasource.mapper` has the following possible values:

- rdb: generic mapper (JDBC standard driver ...)
- rdb.postgres: mapper for PostgreSQL
- rdb.oracle8: mapper for Oracle 8 and lesser versions
- rdb.oracle: mapper for Oracle 9
- rdb.mysql: mapper for MySQL
- `rdb.cloudscape`: Cloudscape
- `rdb.db2`: DB2
- `rdb.firebird`: Firebird
- `rdb.hsql`: HSQL
- `rdb.sapdb`: SAP DB
- `rdb.sybase`: Sybase

4.3. JDBC Connection Pool Configuration

Each Datasource is implemented as a connection manager and manages a pool of JDBC connections. The pool can be configured via some additional properties described in the following table. Refer to the `Oracle1.properties` file to see an example of settings. All these settings have default values and are not required to be specified. All these attributes can be reconfigured when jonas is running from the jonas console.

Property Name	Description	Default Value
jdbc.connchecklevel	JDBC connection checking level: 0 (no check), 1 (check if the connection is still open), 2 (call the test statement before reusing a connection from the pool)	1
jdbc.connmaxage	number of minutes a connection can be kept in the pool. If the minconpool limit has not been reached after this time, the connection will be closed.	1440 minutes (= 1 day)
jdbc.connteststmt	test statement in case jdbc.connchecklevel = 2	select 1
jdbc.maxopentime	Maximum time (in minutes) that a connection can be left busy. If the caller has not issued a `close()` during this time, the connection will be closed automatically.	1440 mn (= 1 day)
jdbc.minconpool	Minimum number of connections in the pool. Setting a positive value here insures that the pool size will not go below this limit during the datasource's lifetime.	0
jdbc.maxconpool	Maximum number of connections in the pool. Limiting the maximum pool size avoids errors from the database.	no limit

Property Name	Description	Default Value
jdbc.samplingperiod	Sampling period for JDBC monitoring (the number of seconds between two measures)	60 sec
jdbc.maxwaittime	Maximum time (in seconds) to wait for a connection in case of shortage. This is valid only if `maxconpool` has been set.	10 sec
jdbc.maxwaiters	Maximum number of concurrent waiters for a JDBC Connection. This is valid only if `maxconpool` has been set.	1000

When a user requests a JDBC connection, the dbm connection manager first tries to see if a connection is already opened for its transaction. If not, JOnAS first tries to get a free connection from the free list. If there are no more connections available, the dbm connection manager creates a new JDBC connection (if `jdbc.maxconpool` is not reached). If it cannot create new connections, the user must wait (if `jdbc.maxwaiters` is not reached) until a connection is released. After a limited time (`jdbc.maxwaittime`), the getConnection returns an exception. When the user calls `close()` on the connection, the connection is put back in the free list. Many statistics are computed (every `jdbc.samplingperiod` seconds) and can be viewed by using the jonas console. This is useful for tuning these parameters and to see the server load at any time.

When a connection has existed for too long (`jdbc.connmaxage`), the pool tries to release it from the freelist. However, the dbm connection manager always tries to always keep at least `jdbc.minconpool` opened connections.

When the user has forgotten to close a JDBC connection, the system can automatically close it after `jdbc.maxopentime` minutes. Note that if the program tries to use this connection later expecting that it is still open, it returns an exception (socket closed).

When a connection is reused from the freelist, it is possible to verify if it is still valid. This is configured in `jdbc.conncklevel`. The ultimate test is to try a dummy statement on the connection before returning it to the caller. This statement is configured by `jdbc.connteststmt`.

4.4. Tracing SQL Requests Through P6Spy

The P6Spy tool is integrated within JOnAS to provide a means for easily tracing the SQL requests that are sent to the database (see http://www.p6spy.com/).

To enable this tracing feature, perform the following configuration steps:

1. Set the `datasource.classname` property of your DataSource properties file to `com.p6spy.engine.spy.P6SpyDriver`
2. Set the `realdriver` property in the `spy.properties` file (located within `$JONAS_BASE/conf`) to the JDBC driver of your actual database
3. Verify that `logger.org.objectweb.jonas.jdbc.sql.level` is set to `DEBUG` in the `$JONAS_BASE/conf/trace.properties` file.

Example:

DataSource properties file contents:

```
datasource.name       jdbc_3
datasource.url        jdbc:postgresql://your_host:port/your_db
datasource.classname  com.p6spy.engine.spy.P6SpyDriver
datasource.username   jonas
datasource.password   jonas
```

```
datasource.mapper      rdb.postgres
```

Within the `$JONAS_BASE/conf/spy.properties` file:

```
realdriver=org.postgresql.Driver
```

Within the `$JONAS_BASE/conf/trace.properties` file:

```
logger.org.objectweb.jonas.jdbc.sql.level   DEBUG
```

JOnAS Class Loader Hierarchy

This chapter is for the EAR application provider; that is, the person in charge of developing the J2EE application components on the server side. It describes a new and important key feature of the J2EE integration: the class loader hierarchy in JOnAS.

5.1. Understanding the Class Loader Hierarchy

An application is deployed by its own class loader. This means, for example, that if a WAR and an EJB JAR are deployed separately, the classes contained in the two archives are loaded with two separate classloaders with no hierarchy between them. Thus, the EJBs from within the JAR will not be visible to the Web components in the WAR.

This is not acceptable in cases where the Web components of an application need to reference and use some of the EJBs.

For this reason, prior to EAR files, when a Web application had to be deployed using EJBs, the EJB JAR had to be located in the `WEB-INF/lib` directory of the Web application.

Currently, with the J2EE integration and the use of the EAR packaging, class visibility problems no longer exist and the EJB JAR is no longer required to be in the `WEB-INF/lib` directory.

The following sections describe the JOnAS class loader hierarchy and explain the mechanism used to locate the referenced classes.

5.2. Commons Class Loader

The JOnAS-specific `commons` class loader will load all classes and libraries required to start the JOnAS server (that is, libraries for mail, Tomcat, etc.). This class loader has the system class loader as parent class loader. The commons class loader adds all the common libraries required to start the JOnAS server (J2EE applications, commons logging, ObjectWeb components, etc.); it also loads the classes located in `XTRA_CLASSPATH`.

The JARs loaded by the `commons` class loader are located under the `JONAS_ROOT/lib/commons` and `JONAS_BASE/lib/commons` directories. You can extend this class loader by adding your own JARs inside these directories. If you are using a Jetty packaging, `JONAS_ROOT/lib/jetty` will be loaded too.

> **Note**
>
> Note that the `lib/ext` extension mechanism is now deprecated. You should place additional JARs directly in the classloader directory (`commons`, `apps`, `tools`) under `JONAS_ROOT` and/or `JONAS_BASE`. You should now use the `JONAS_BASE/lib/commons` if you want to extend the `commons` ClassLoader, `JONAS_BASE/lib/apps` for `apps` ClassLoader, and `JONAS_BASE/lib/tools` for `tools` ClassLoader. Notice that the `JONAS_BASE` extension directories are always loaded after `JONAS_ROOT` directories, so if the same class is in `JONAS_ROOT/lib/commons` and in `JONAS_BASE/lib/commons`, the one located in `JONAS_ROOT/lib/commons` will be used.

To have a library available for each component running inside JOnAS, add the required JAR files in the `JONAS_ROOT/lib/commons` directory or in `JONAS_BASE/lib/commons`. All JARs in subordinate directories will also be loaded.

If a specific JAR is needed only for a web application (that is, you need to use a version of a JAR file that is different than a version loaded by JOnAS), change the compliance of the web application classloader to the Java 2 delegation model. See Section 5.6.3 *WEB Class Loader*.

5.3. Application Class Loader

The `application` class loader is a JOnAS-specific class loader that loads all application classes required by the user applications. This implies that this loader will load all single RAR files, so all applications have the visibility of the resource adapter's classes. This class loader has the `commons` class loader as its parent class loader.

The JARs loaded by the `application` class loader are located under the `JONAS_ROOT/lib/apps` directory and under the `JONAS_ROOT/lib/catalina/common/lib` directory. You can extend this class loader by adding your own JARs inside these directories.

5.4. Tools Class Loader

The `tools` class loader is a JOnAS-specific class loader that loads all classes for which applications do not require visibility (that is, user applications will not be able to load the classes packaged in the `tools` class loader). For example, it includes the jakarta velocity and digester components. This class loader has the `commons` class loader as its parent class loader.

The JARs loaded by the `tools` class loader are located under the `JONAS_ROOT/lib/tools` directory and under the `JONAS_BASE/lib/tools` directory. You can extend this class loader by adding your own JARs inside these directories.

5.5. Tomcat Class Loader

The `tomcat` class loader loads all classes of the Tomcat server (the `CATALINA_HOME/server/lib` directory). The classes of the common directory of Tomcat (`CATALINA_HOME/common/lib` directory) are loaded by the `application` class loader and not by this `tomcat` class loader. Applications have the visibility of the `common` classes and not the `server` classes. To have the visibility of the `server` class, the context must have the privileged attribute set to `true`. This class loader has the `application` class loader as its parent class loader.

The JARs loaded by the `tomcat` class loader are located under the `JONAS_ROOT/catalina/server/lib` directory (if using Tomcat packaging, unless these libs are located under `CATALINA_HOME/server/lib`). You can extend this class loader by adding your own JARs inside this directory.

5.6. JOnAS Class Loaders

The JOnAS class loader hierarchy that allows the deployment of EAR applications without placing the EJB JAR in the `WEB-INF/lib` directory consists of the following:

5.6.1. EAR Class Loader

The EAR class loader is responsible for loading the EAR application. There is only one EAR class loader per EAR application. This class loader is the child of the application class loader, thus making it visible to the JOnAS classes.

5.6.2. EJB Class Loader

The EJB class loader is responsible for loading all the EJB JARs of the EAR application, thus all the EJBs of the same EAR application are loaded with the same EJB classloader. This class loader is the child of the EAR class loader.

5.6.3. WEB Class Loader

The WEB class loader is responsible for loading the Web components. There is one WEB class loader per WAR file, and this class loader is the child of the EJB class loader. Using this class loader hierarchy (the EJB class loader is the parent of the WEB class loader) eliminates the problem of visibility between classes when a WEB component tries to reference EJBs; the classes loaded with the WEB class loader are definitely visible to the classes loaded by its parent class loader (EJB class loader).

The compliance of the class loader of the web application to the Java 2 delegation model can be changed by using the `jonas-web.xml` file. This is described in Chapter 17 *Defining the Web Deployment Descriptor*.

If the `java2-delegation-model` element is set to `false`, the class loader of the web application looks for the class in its own repository before asking its parent class loader.

5.7. JOnAS Class Loader Hierarchy

The resulting JOnAS class loader hierarchy is as follows:
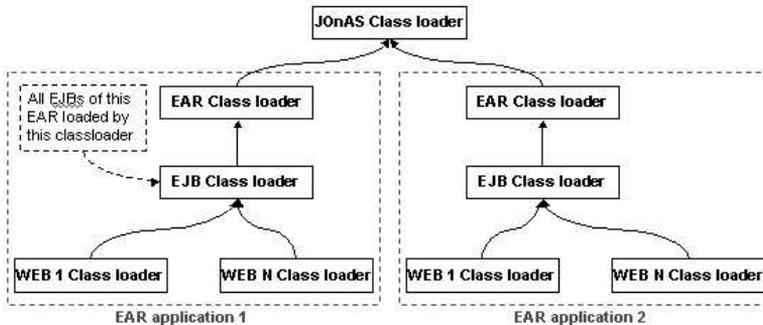
Figure 5-1. JOnAS Class Loader Hierarchy

Chapter 6.

JOnAS Command Reference

Commands provided with JOnAS are described in this chapter.

6.1. jonas

6.1.1. Synopsis

```
jonas start [-fg | -bg | -win] [-n name]
```
    start a JOnAS server

```
jonas stop [-n name]
```
    Stop a JOnAS server.

```
jonas admin [-n name] [admin_options]
```
    Administer a JOnAS server.

```
jonas check
```
    Check the environment before running a JOnAS server.

```
jonas version
```
    Print the current version of JOnAS.

6.1.2. Description

This command provides the capability to start, stop, or administer JOnAS servers.

The outcome of this program may depend on the directory from which the command is run (that is depending on the existence of a `jonas.properties` file in the current directory). It is possible to set system properties to the program by using the JAVA_OPTS environment variable, if required. Note that setting system properties with a -D option will always take precedence over the properties described in the other `jonas.properties` files.

The jonas script /usr/share/jonas/bin/unix/jonas can be reviewed and possibly modified for assistance with problems or for obtaining additional information.

There are five different sub-commands that depend on the first mandatory argument:

jonas start

> Start a new JOnAS server. The process can be run in the foreground, in the background, or in a new window. If the background option is chosen (default option), control is given back to the caller only when the server is ready. The default name is jonas. A different name can be given with the -n option.

jonas stop

> Stop a running JOnAS server. Use the -n option if the server was given a name other than the default name.

jonas admin

> Administer a JOnAS server. Use the -n option if the server was given a name other than the default name. Used without any other option, this command will prompt the user for an administrative command (interactive mode). Each administrative command exists in a non-interactive mode, for use in applications such as shell scripts, for example. Refer to the option list for a description of each.

jonas check

> Check the environment settings before running a JOnAS server. (This is a very basic troubleshooting test.)

jonas version

> Print the current version of JOnAS.

6.1.3. Options

Each option may be pertinent only for a subset of the five different sub-commands. For example, jonas check and jonas version do not accept any options.

-n name

> Give a name to the JOnAS server. The default is jonas. Used for start, stop, or admin.

-fg

> Used for start only. The server is launched in the foreground: control is given back to the user only when the JOnAS server is stopped.

-bg

> Used for start only. The server is launched in the background. Control is given back to the user only when the JOnAS server is ready. This is the default mode.

-win

> Used for start only. The server is launched in a new window.

-?

> Used for admin only. Prints a help with all possible options.

`-a filename`

> Used for `admin` only. Deploys a new application described by *`filename`* inside the JOnAS Server. The application can be one of the following:
>
> - A standard EJB-JAR file. This leads to the creation of a new EJB Container in the JOnAS Server. If the file name has a relative path, this path is relative to where the EJB server has been launched or relative to the `$JONAS_ROOT/ejbjars` directory for an EJB-JAR file.
>
> - A standard WAR file containing a WEB Component. If the file name has a relative path, this path is relative to where the EJB server has been launched or relative to the `$JONAS_ROOT/webapps` directory for a WAR file.
>
> - A standard EAR file containing a complete J2EE application. If the file name has a relative path, this path is relative to where the EJB server has been launched or relative to the `$JONAS_ROOT/apps` directory for an EAR file.

`-r filename`

> Used for `admin` only. Dynamically removes a previous `-a filename` command.

`-gc`

> Used for `admin` only. Runs the garbage collector in the specified JOnAS server.

`-passivate`

> Used for `admin` only. Passivates all Entity Bean instances. This affects only instances outside transaction.

`-e`

> Used for `admin` only. Lists the properties of the specified JOnAS server.

`-j`

> Used for `admin` only. Lists the registered JNDI names, as seen by the specified JOnAS server.

`-l`

> Used for `admin` only. Lists the beans currently loaded by the specified JOnAS server.

`-sync`

> Used for `admin` only. Synchronizes the Entity Bean instances on the current JOnAS server. Note that this affects only the instances that are not involved in a transaction.

`-debug topic`

> Used for `admin` only. Sets the topic level to DEBUG.

`-tt timeout`

> Used for `admin` only. Changes the default timeout for transactions. *`timeout`* is in seconds.

Each `jonas admin` option has its equivalent in the interactive mode. To enter interactive mode and access the following list of subcommands, type `jonas admin [-n name]` without any other argument. To exit from interactive mode, use the `quit` command.

Interactive Command	Online Matching Command
addbeans	`-a fileName`

Interactive Command	Online Matching Command
env	-e
gc	-gc
help	-?
jndinames	-j
listbeans	-l
removebeans	-r fileName
sync	-sync
trace	-debug topic
ttimeout	-tt timeout
quit	exit interactive mode

6.1.4. Examples

```
jonas check

jonas start -n jonas1

jonas admin -n jonas1 -a bean1.jar

jonas stop -n jonas1
```

6.2. jclient

6.2.1. Synopsis

jclient [options] java-class [args]

Start a Java client.

6.2.2. Description

The jclient command allows the user to easily start a "heavy" java client that will be able to reach beans in remote JOnAS servers and start distributed transactions.

6.2.3. Options

```
-cp classpath
```

Add an additional classpath before running the Java program.

```
-security
```

>Set a security manager using the policy file in $JONAS_BASE/conf/java.policy. (This is used to download stubs automatically; see Section 3.5.3 *Automatic Downloading of Stubs from Clients*.)

6.2.4. Example

```
jclient package.javaclassname args
```

6.3. newbean

6.3.1. Synopsis

newbean

>Generates skeletons for all the necessary files for making a bean.

6.3.2. Description

The `newbean` tool helps the bean writer start developing a bean by generating skeletons for all the necessary files for making a bean. Note that this tool creates only templates of the files. These templates must then be customized and the business logic written. However, the files should be compilable.

To create these templates, type `newbean` and enter a set of parameters in interactive mode.

`newbean` generates a `build.xml` file.

The `Bean Name` must start with a capital letter. Avoid the reserved names: Home, EJB, Session, Entity. This name will be used as a prefix for all filenames relative to the bean.

The `Bean Type` must be one of the following:

- `S` Session Bean
- `E` Entity Bean
- `MD` Message-Driven Bean

The `Session Type` must be one of the following:

- `L` Stateless Session Bean
- `F` Stateful Session Bean

The `Persistence manager` must be one of the following:

- `B` Bean-Managed Persistence (BMP)
- `C` Container-Managed Persistence (CMP 1.x)
- `C2` Container-Managed Persistence (CMP 2.x)

The `Bean Location` must be one of the following:

- `R` Remote Interfaces
- `L` Local Interfaces

The `Package name` is a dot-separated string representing the package to which the bean belongs. Usually this is the same as the current directory.

The `Jar name` argument is the name that will be used to build the `.jar` file. Do not provide the `.jar` extension with this argument. Typically, the last part of the package name is used.

The `Primary Key class` is the class that represents the primary key. It is required only for Entity Beans. Its possible values are:

- `S` java.lang.String
- `I` java.lang.Integer
- `O` Object (Will be chosen later)

6.3.3. Example

```
newbean
Bean Name
> MyFirstBean


Bean type
  S   Session bean
  E   Entity bean
  MD  Message-Driven bean
> E

Persistence manager
  B   Bean
  C   Container (CMP 1.x)
  C2  Container (CMP 2.x)
> C

Bean location
  R   Remote
  L   Local
> R

Package name
> truc.machin

Jar name
> machin

Primary Key class
  S   String
  I   Integer
  O   Object
> S

Creating bean MyFirstBean (type ECR) in package truc.machin
Your bean files have been created. You can now customize them.
```

6.3.4. Example 2

```
-bash-2.05b$ /usr/share/jonas/bin/unix/newbean
Bean Name
> MySecondBean


Bean type
  S   Session bean
  E   Entity bean
  MD  Message-Driven bean
> S

Session type
  L  Stateless
  F  Stateful
> F

Bean location
  R  Remote
  L  Local
> L

Package name
> com.redhat.rhdb.cc.jonas

Jar name
> rhaps-cc-jonas

Creating bean MySecondBean (type SFL) in package com.redhat.rhdb.cc.jonas
Your bean files have been created. You can now customize them.
```

6.4. registry

6.4.1. Synopsis

registry [*port*]

   Creates and starts a remote object registry.

6.4.2. Description

The registry tool creates and starts a remote object RMI registry on the specified port of the current host. If the port is omitted, the registry is started on port 1099.

Note that, by default, the registry is collocated in the same JVM as the JOnAS Server. In this case, it is not necessary to use this tool; the registry is automatically launched.

6.4.3. Options

port

   Port number.

6.4.4. Example

The `registry` command can normally be run in the background:

```
registry &
```

6.5. GenIC

6.5.1. Synopsis

GenIC [Options] *InputFilename*

   Starts the GenIC utility.

6.5.2. Description

The GenIC utility generates the container classes for JOnAS from the given Enterprise Java Beans.

The *InputFileName* is either the file name of an EJB-JAR file or the file name of an XML deployment descriptor of beans.

The GenIC utility does the following in the order listed:

   1. Generates the sources of the container classes for all the beans defined in the deployment descriptor.
   2. Compiles these classes via the Java compiler.
   3. Generates stubs and skeletons for those remote objects via the RMI compiler.
   4. If the *InputFile* is an EJB-JAR file, adds the generated classes in this EJB-JAR file.

6.5.3. Options

-d *directory*

   Specifies the root directory of the class hierarchy.

   This option can be used to specify a destination directory for the generated files.

   If the -d option is not used, the package hierarchy of the target class is ignored and the generated files are placed in the current directory.

   If the InputFile is an EJB-JAR file, the generated classes are added to the EJB-JAR file, unless the -noaddinjar option is set.

-invokecmd

Invoke, directly in some cases, the method of the Java class corresponding to the command.

-javac *options*

Specifies the `java` compiler name to use (`javac` by default).

-javacopts *options*

Specifies the options to pass to the `java` compiler.

-keepgenerated

Do not immediately delete generated files.

-noaddinjar

If the `InputFile` is an EJB-JAR file, do not add the generated classes to the EJB-JAR file.

-nocompil

Do not compile the generated source files via the Java and RMI compilers.

-novalidation

Remove xml validation during parsing.

-protocols

Comma-separated list of protocols (chosen from jrmp, iiop, cmi) for which stubs should be generated. Default is `jrmp`.

-rmiopts *options*

Specifies the options to pass to the `rmi` compiler.

-verbose

Displays additional information about command execution.

-nofastrmic

Disable the use of `fastrmic` for the generation of stubs/ties.

6.5.4. Example

```
GenIC -d ../../classes sb.xml
```

Generates container classes of all the Enterprise JavaBeans defined in the `sb.xml` file. Classes are generated in the `../../classes` directory adhering to the classes hierarchy.

```
GenIC sb.jar
```

Generates container classes for all the Enterprise JavaBeans defined in the `sb.jar` file and adds the generated classes to this EJB-JAR file.

6.5.5. Environment

If `InputFile` is an XML deployment descriptor, the classpath must include the paths of the directories in which the Enterprise Bean's classes can be found, as well as the path of the directory specified by the `-d` option.

If `InputFile` is an EJB-JAR file, the classpath must include the path of the directory specified by the `-d` option.

6.6. JmsServer

6.6.1. Synopsis

JmsServer

   Launches the JORAM Server.

6.6.2. Description

Launches the JORAM Server (that is, the MOM) with its default options.

6.6.3. Options

None.

6.6.4. Example

The `JmsServer` command is typically run in the background:

```
JmsServer &
```

6.7. RAConfig

6.7.1. Synopsis

RAConfig [Options] *InputFilename* [*OutputFilename*]

   Generates a JOnAS-specific resource-adapter configuration file.

6.7.2. Description

The RAConfig utility generates a JOnAS-specific resource-adapter configuration file (`jonas-ra.xml`) from an `ra.xml` file (Resource adapter deployment descriptor).

The *InputFileName* is the file name of a resource adapter.

The *OutputFileName* is the file name of an output resource adapter used with the `-p` (required) or `-u` (optional).

6.7.3. Options

Only capital letters or the full option name can be specified.

`-? options`

>   Gives a summary of the options.

`-DM,-DS,-PC,-XA DriverManager, DataSource, PooledConnection, XAConnection`

>   Specifies the rarlink value to configure; used with the `-p` option.

`-ENcrypt`

>   Used with `-SecurityFile` to encrypt the specified passwords.

`-J jndiname`

>   It is a mandatory option. It specifies the JNDI name of the *connection factory*. This name corresponds to the name of the <jndi-name> element of the <jonas-resource> element in the JOnAS-specific deployment descriptor. This name is used by the *resource service* for registering in JNDI the connection factory corresponding to this resource adapter.

`-NoValidation`

>   Turns off the XML DTD/schema validation.

`-PATH output_directory_for_the_generated_jonas-ra.xml`

>   Specifies the directory name to place the generated `jonas-ra.xml` file. When not specified, the default value is the System attribute of `java.io.tmpdir`.

`-Property database_properties_file`

>   Specifies the name of the database.properties file to process. The result of this processing will be a `jonas-ra.xml` file that will update the `/META-INF/jonas-ra.xml` file in the output RAR.

`-Rarlink rarlink`

>   Specifies the JNDI name of the RAR file with which to link. This option can be used when this RAR file will inherit all attributes associated with the specified JNDI name. If this option is specified in the `jonas-ra.xml` file, it is the only file needed in the RAR, and the `ra.xml` file will be processed from the rarlink file.

`-SecurityFile security_property_file_to_process`

>   Specifies the security property file to process and adds security information to `jonas-ra.xml`. This maps the specified principal name to the user on the EIS system. The specified file must be in the form: `principal = user::password`. When used in conjunction with the `-ENcrypt` option, the resulting information is encrypted in `jonas-ra.xml`.

`-Update inputname`

>   Specifies the name of the XML file to process. This file will update the `/META-INF/jonas-ra.xml` file in the RAR. If this argument is used, it is the only argument executed.

`-Verbose`

>   Verbose mode. Displays the deployment descriptor of the resource adapter on standard `System.out`.

6.7.4. Example

```
RAConfig -j adapt_1 MyRA.rar
```

> Generates the `jonas-ra.xml` file from the `ra.xml` file.

> After `jonas-ra.xml` has been configured for the `MyRA.rar` file,
> `RAConfig -u jonas-ra.xml MyRA.rar`

> Updates/inserts the `jonas-ra.xml` file into the RAR file.

```
RAConfig -dm -p MySQL1 $JONAS_ROOT/rars/autoload/JOnAS_jdbcDM MySQL_dm
```

> Generates the `jonas-ra.xml` file from the `ra.xml` file of the `JOnAS_jdbcDM.rar` and inserts
> the corresponding values from the `MySQL1.properties` file. The `jonas-ra.xml` file is then
> added/updated to the `MySQL_dm.rar` file. This RAR file can then be deployed and will replace
> the configured MySQL1 datasource.

6.8. newjb

6.8.1. Synopsis

```
newjb;
```

6.8.2. Description

The newjb utility builds a new `JONAS_BASE` directory that enables you to launch the conformance
tests. At the start, you must choose:

- The protocol, either `jrmp`, `jeremie`, `iiop`, `cmi`
- The database
- The web container, either `tomcat`, `jetty`

Once this is done, the tool generates the configuration automatically.

You must set the `$JONAS_BASE` variable before the launch of the tool; it specifies the path to the new
directory to be built.

The `$HOME/jb.config/lib` directory must be created before launching the tool. It can contain some
specific user configuration (see below).

The tool relies on JOnAS's ant tasks and thus builds a configuration compatible with the JOnAS ver-
sion. Firstly, a `JONAS_BASE` with default values is built, and then the configuration files are modified
with the values defined in the centralized configuration file of `newjb` (see below).

A default configuration file is provided in `$JONAS_ROOT/build-jc.properties`. It contains the
variable parameters used by the tool, such as the port number and database properties.

A user configuration can be set in the `$HOME/jb.config/conf/jonas-newjb.properties` file.
If this file is present, the parameters it contains will override the default parameters.

By default, only the HSQL database can be configured with this tool. For other databases, the specific
drivers must be stored in the `$HOME/jb.config/lib` directory before the run and the properties
must be set in the `$HOME/jb.config/conf/jonas-newjb.properties` file.

Also by default, the load-balancing at the web level is implemented through the Apache
`mod_jk` plugin. You can also use the Enhydra-director plugin by modifying the configuration

file ($HOME/jc.config/conf/jonas-newjb.properties) and by storing the tomcat5.5-director.jar (http://director.objectweb.org/) file in the $HOME/jc.config/lib directory before the run.

6.9. newjc

6.9.1. Synopsis

```
cd $JONAS_ROOT/examples/sampleCluster2/newjc
newjc.sh;
```

6.9.2. Description

The newjc utility builds a new cluster configuration. Here a cluster configuration is a set of new JONAS_BASE directories and the mod_jk/director configuration files. At the start, you must choose:

- The output directory. The default value is the $JONAS_BASE value. This is the directory where all the JONAS_BASE directories of the cluster configuration and the mod_jk/director configuration will be created.
- The prefix for the JONAS_BASE directories. By default jb is suggested and in this case, the JONAS_BASE will be jb1, jb2, and so on.
- The protocol, from jrmp, jeremie, iiop, cmi
- The database
- The web container
- The cluster architecture between bothWebEjb and diffWebEjb. The first describes a cluster where all the nodes contain the same services, whereas the second one describes an architecture where the web level and the EJB level are separated.
- The number of web instances to create.
- The number of EJB instances to create. If there is not a separation between the web and ejb levels, the total number of instances is the sum of the web instances and the ejb instances.

Once this is done, the tool generates the configuration automatically.

You must create the $HOME/jc.config/lib directory before launching the tool. It can contain some specific user configuration (see below).

The tool relies on JOnAS's Ant tasks and thus builds a configuration compatible with the JOnAS version. Firstly, the JONAS_BASE directories with default values are built, and then the configuration files are modified with the values defined in the centralized configuration file of newjc (see below).

A default configuration file is provided in $JONAS_ROOT/examples/sampleCluster2/newjc/build-jc.properties. It contains the variable parameters used by the tool, such as port number range and database properties.

A user configuration can be set in the $HOME/jc.config/conf/jonas-newjc.properties file. If this file is present, the parameters it contains override the default parameters.

By default, only the HSQL database can be configured with this tool. For other databases, the specific drivers must be stored in the $HOME/jc.config/lib directory before the run and the properties must be set in the $HOME/jc.config/conf/jonas-newjb.properties file.

II. Enterprise Beans Programmer's Guide

This part contains information for the Enterprise Beans programmer; that is, the person in charge of developing the software components on the server side and, more specifically, the Session Beans.

The individual in charge of developing Enterprise Beans should consult chapters in this part for instructions on how to perform the following tasks:

- Write the source code for the beans.
- Specify the deployment descriptor.
- Bundle the compiled classes and the deployment descriptor into an EJB JAR file.

For information on developing the three types of enterprise beans, refer to:

- Chapter 7 *Developing Session Beans*
- Chapter 8 *Developing Entity Beans*
- Chapter 9 *Developing Message-Driven Beans*.

The deployment descriptor specification is presented in Chapter 10 *Defining the Deployment Descriptor*.

More specific issues related to transaction behavior, the Enterprise Bean environment, and security service are presented in the corresponding chapters: Chapter 11 *Transactional Behavior of EJB Applications*, Chapter 12 *Enterprise Bean Environment*, and Chapter 13 *Security Management*.

Principles and tools for providing EJB JAR files are presented in Chapter 14 *EJB Packaging* and Chapter 15 *Application Deployment and Installation Guide*.

Table of Contents

Developing Session Beans

This chapter is for the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side and, more specifically, the Session Beans.

Note

In this documentation, the term "Bean" always means "Enterprise Bean."

7.1. Introduction to Session Beans

A Session Bean is composed of the following parts, which are developed by the Enterprise Bean provider:

- The *Component Interface* is the client view of the bean. It contains all the "business methods" of the bean.
- The *Home Interface* contains all the methods for the bean life cycle (creation, suppression) used by the client application.
- The *bean implementation class* implements the business methods and all the methods (described in the EJB specification), allowing the bean to be managed in the container.
- The *deployment descriptor* contains the bean properties that can be edited at assembly or deployment time.

Note that, according to the EJB 2.0 specification, the couple "Component Interface and Home Interface" may be either local or remote. *Local Interfaces* (Home and Component) are to be used by a client running in the same JVM as the EJB component. Create and finder methods of a local or remote home interface return local or remote component interfaces respectively. An EJB component can have both remote and local interfaces, even if typically only one type of interface is provided.

The description of these elements is provided in the sections that follow.

A Session Bean object is a short-lived object that executes on behalf of a single client. There are *stateless* and *stateful Session Beans*. Stateless Beans do not maintain state across method calls. Any instance of stateless beans can be used by any client at any time. Stateful Session Beans maintain state within and between transactions. Each stateful session bean object is associated with a specific client. A stateful Session Bean with container-managed transaction demarcation can optionally implement the *SessionSynchronization* interface. In this case, the bean objects will be informed of transaction boundaries. A rollback could result in a Session Bean object's state being inconsistent; in this case, implementing the SessionSynchronization interface may enable the bean object to update its state according to the transaction completion status.

7.2. The Home Interface

A Session Bean's home interface defines one or more `create(...)` methods. Each `create` method must be named `create` and must match one of the `ejbCreate` methods defined in the enterprise Bean class. The return type of a create method must be the enterprise Bean's remote interface type.

The home interface of a stateless Session Bean must have one `create` method that takes no arguments.

All the exceptions defined in the `throws` clause of an `ejbCreate` method must be defined in the `throws` clause of the matching `create` method of the home interface.

A *remote home interface* extends the `javax.ejb.EJBHome` interface, while a *local home interface* extends the `javax.ejb.EJBLocalHome` interface.

7.2.1. Session Bean Example:

The following examples use a Session Bean named `Op`.

```
public interface OpHome extends EJBHome {
    Op create(String user) throws CreateException, RemoteException;
}
```

A local home interface could be defined as follows (`LocalOp` being the local component interface of the bean):

```
public interface LocalOpHome extends EJBLocalHome {
    LocalOp create(String user) throws CreateException;
}
```

7.3. The Component Interface

The Component Interface is the client's view of an instance of the Session Bean. This interface contains the business methods of the Enterprise Bean. If it is remote, the interface must extend the `javax.ejb.EJBObject` interface; if it is local, the interface must extend the `javax.ejb.EJBLocalObject` interface. The methods defined in a remote component interface must follow the rules for Java RMI (this means that their arguments and return value must be valid types for Java RMI, and their throws clause must include the `java.rmi.RemoteException`). For each method defined in the component interface, there must be a matching method in the enterprise Bean's class (same name, same arguments number and types, same return type, and same exception list, except for `RemoteException`).

7.3.1. Example:

```
public interface Op extends EJBObject {
    public void buy (int Shares)   throws RemoteException;
    public int  read ()            throws RemoteException;
}
```

The same type of component interface could be defined as a local interface (even if it is not considered good design to define the same interface as both local and remote):

```
public interface LocalOp extends EJBLocalObject {
    public void buy (int Shares);
    public int  read ();
}
```

7.4. The Enterprise Bean Class

This class implements the Bean's business methods of the component interface and the methods of the `SessionBean` interface, which are those dedicated to the EJB environment. The class must be defined as public and may not be abstract. The `Session Bean` interface methods that the EJB provider must develop are the following:

- `public void` *setSessionContext*(`SessionContext ic`);

  This method is used by the container to pass a reference to the SessionContext to the bean instance. The container invokes this method on an instance after the instance has been created. Generally, this method stores this reference in an instance variable.

- `public void` *ejbRemove*();

  This method is invoked by the container when the instance is in the process of being removed by the container. Since most session Beans do not have any resource state to clean up, the implementation of this method is typically left empty.

- `public void` *ejbPassivate*();

  This method is invoked by the container when it wants to passivate the instance. After this method completes, the instance must be in a state that allows the container to use the Java Serialization protocol to externalize and store the instance's state.

- `public void` *ejbActivate*();

  This method is invoked by the container when the instance has just been reactivated. The instance should acquire any resource that it has released earlier in the `ejbPassivate()` method.

A stateful session Bean with container-managed transaction demarcation can optionally implement the `javax.ejb.SessionSynchronization` interface. This interface can provide the Bean with transaction-synchronization notifications. The `Session Synchronization` interface methods that the EJB provider must develop are the following:

- `public void` *afterBegin*();

  This method notifies a session Bean instance that a new transaction has started. At this point the instance is already in the transaction and can do any work it requires within the scope of the transaction.

- `public void` *afterCompletion*(`boolean committed`);

  This method notifies a session Bean instance that a transaction commit protocol has completed and tells the instance whether the transaction has been committed or rolled back.

- `public void` *beforeCompletion*();

  This method notifies a session Bean instance that a transaction is about to be committed.

7.4.1. Enterprise Bean Class Example:

```
package sb;

import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.EJBObject;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.SessionSynchronization;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```java
// This is an example of Session Bean, stateful, and synchronized.

public class OpBean implements SessionBean, SessionSynchronization {

    protected int total = 0;          // actual state of the bean
    protected int newtotal = 0;       // value inside Tx, not yet committed.
    protected String clientUser = null;
    protected SessionContext sessionContext = null;

    public void  ejbCreate(String user) {
        total = 0;
        newtotal = total;
        clientUser = user;
    }

    public void ejbActivate() {
        // Nothing to do for this simple example
    }

    public void ejbPassivate() {
        // Nothing to do for this simple example
    }

    public void ejbRemove() {
        // Nothing to do for this simple example
    }

    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    public void afterBegin() {
        newtotal = total;
    }

    public void beforeCompletion() {

        // We can access the bean environment everywhere in the bean,
        // for example here!
        try {
            InitialContext ictx = new InitialContext();
            String value = (String) ictx.lookup("java:comp/env/prop1");
            // value should be the one defined in ejb-jar.xml
        } catch (NamingException e) {
            throw new EJBException(e);
        }
    }

    public void afterCompletion(boolean committed) {
        if (committed) {
            total = newtotal;
        } else {
            newtotal = total;
        }
    }

    public void buy(int s) {
        newtotal = newtotal + s;
        return;
    }

    public int read() {
```

```
        return newtotal;
    }
}
```

7.5. Tuning the Stateless-Session Bean Pool

JOnAS handles a pool for each stateless Session Bean. The pool can be configured in the JOnAS-specific deployment descriptor with the following tags:

min-pool-size

> This optional integer value represents the minimum instances that will be created in the pool when the bean is loaded. This will improve bean instance creation time, at least for the first beans. The default value is `0`.

max-cache-size

> This optional integer value represents the maximum of instances in memory. This value keeps JOnAS scalable.

> The policy is that, at bean creation time, an instance is taken from the pool of free instances. If the pool is empty, a new instance is always created. When the instance must be released (at the end of a business method), it is pushed into the pool, except if the current number of instances created exceeds the `max-cache-size`, in which case this instance is dropped. The default value is `no limit`.

```
<jonas-ejb-jar>
  <jonas-session>
    <ejb-name>SessSLR</ejb-name>
    <jndi-name>EJB/SessHome</jndi-name>
    <max-cache-size>20</max-cache-size>
    <min-pool-size>10</min-pool-size>
  </jonas-session>
</jonas-ejb-jar>
```
Example 7-1. Stateless-Session Bean Pool Example

Chapter 8.
Developing Entity Beans

This chapter is for is the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side, and more specifically the Entity Beans.

> **Note**
>
> In this documentation, the term "Bean" always means "Enterprise Bean."

8.1. Introduction to Entity Beans

An Entity Bean is comprised of the following elements, which are developed by the Enterprise Bean Provider:

- The *Component Interface* is the client view of the bean. It contains all the "business methods" of the bean.
- The *Home Interface* contains all the methods for the bean life cycle (creation, suppression) and for instance retrieval (finding one or several bean objects) used by the client application. It can also contain methods called "home methods," supplied by the bean provider, for business logic that is not specific to a bean instance.
- The *Primary Key class* (for Entity Beans only) contains a subset of the bean's fields that identifies a particular instance of an Entity Bean. This class is optional since the bean programmer can alternatively choose a standard class (for example, `java.lang.String`)
- The *bean implementation class* implements the business methods and all the methods (described in the EJB specification) allowing the bean to be managed in the container.
- The *deployment descriptor*, containing the bean properties that can be edited at assembly or deployment time.

> **Note**
>
> According to the EJB 2.0 specification, the "Component Interface and Home Interface" can be either local or remote. *Local Interfaces* (Home and Component) are to be used by a client running in the same JVM as the EJB component. Create and finder methods of a local (or remote) home interface return local (or remote) component interfaces. An EJB component may have both remote and local interfaces, even if normally only one type of interface is provided. If an Entity Bean is the target of a container-managed relationship (refer to EJB 2.0 persistence), then it must have local interfaces.
>
> These elements are described in the following sections.

An Entity Bean represents persistent data. It is an object view of an entity stored in a relational database. The persistence of an Entity Bean can be handled in two ways:

- *Container-Managed Persistence*: the persistence is implicitly managed by the container; no code for data access is supplied by the bean provider. The bean's state will be stored in a relational

database according to a mapping description delivered within the deployment descriptor (CMP 1.1) or according to an implicit mapping (CMP 2.0).

- *Bean-Managed Persistence*: the bean provider writes the database access operations (JDBC code) in the methods of the enterprise bean that are specified for data creation, load, store, retrieval, and remove operations (`ejbCreate`, `ejbLoad`, `ejbStore`, `ejbFind...`, `ejbRemove`).

Currently, the platform handles persistence in relational storage systems through the JDBC interface. For both container-managed and bean-managed persistence, JDBC connections are obtained from an object provided at the EJB server level, the *DataSource*. The DataSource interface is defined in the JDBC 2.0 standard extensions (see http://java.sun.com/products/jdbc/). A DataSource object identifies a database and a means to access it via JDBC (a JDBC driver). An EJB server may propose access to several databases and thus provides the corresponding DataSource objects. DataSources are described in more detail in Chapter 4 *Configuring JDBC DataSources*.

8.2. The Home Interface

In addition to "home business methods," the Home interface is used by any client application to create, remove, and retrieve instances of the Entity Bean. The bean provider needs to provide only the desired interface; the container will automatically provide the implementation. If it is remote, the interface must extend the `javax.ejb.EJBHome` interface; if it is local, the interface must extend the `javax.ejb.EJBLocalHome` interface. The methods of a remote home interface must follow the rules for Java RMI. The signatures of the `create` and `find...` methods should match the signatures of the `ejbCreate` and `ejbFind...` methods that will be provided later in the Enterprise Bean implementation class (the same number and types of arguments, but different return types).

8.2.1. create Methods

- The return type is the Enterprise Bean's component interface.
- The exceptions defined in the `throws` clause must include the exceptions defined for the ejbCreate and ejbPostCreate methods, and must include `javax.ejb.CreateException` and `java.rmi.RemoteException` (the latter is only for a remote interface).

8.2.2. remove Methods

- The interfaces for these methods must not be defined—they are inherited from `EJBHome` or `EJBLocalHome`.
- The method is `void remove`, taking as an argument the primary key object or the handle (for a remote interface).
- The exceptions defined in the throws clause should be `javax.ejb.RemoveException` and `java.rmi.RemoteException` for a remote interface.
- The exceptions defined in the `throws` clause should be `javax.ejb.RemoveException` and `java.ejb.EJBException` for a local interface.

8.2.3. finder Methods

Finder methods are used to search for an EJB object or a collection of EJB objects. The arguments of the method are used by the Entity Bean implementation to locate the requested entity objects. For bean-managed persistence, the bean provider is responsible for developing the corresponding ejbFinder methods in the bean implementation. For container-managed persistence, the bean provider

does not write these methods; they are generated at deployment time by the platform tools; the description of the method is provided in the deployment descriptor, as defined in Section 8.7 *Configuring Database Access for Container-Managed Persistence*. In the Home interface, the finder methods must adhere to the following rules:

- They must be named `find<method>` (for example, `findLargeAccounts`).
- The return type must be the Enterprise Bean's component interface, or a collection thereof.
- The exceptions defined in the `throws` clause must include the exceptions defined for the matching `ejbFind` method, and must include `javax.ejb.FinderException` and `java.rmi.RemoteException` (the latter, only for a remote interface).

At least one of these methods is mandatory: `findByPrimaryKey`, which takes as argument a primary key value and returns the corresponding EJB object.

8.2.4. home Methods

Home methods are methods that the bean provider supplies for business logic that is not specific to an Entity Bean instance.

- The `throws` clause of every home method on the remote home interface includes the `java.rmi.RemoteException`.
- Home methods implementation is provided by the bean developer in the bean implementation class as public static methods named `ejbHome<METHOD_NAME>(...)`, where `<METHOD_NAME>` is the name of the method in the home interface.

8.2.5. Home Interface Example

The Account Bean example, provided with the platform examples, is used to illustrate these concepts. The state of an Entity Bean instance is stored in a relational database, where the following table should exist, if CMP 1.1 is used:

```
create table ACCOUNT (ACCNO integer primary key,
      CUSTOMER varchar(30),
      BALANCE number(15,4));

public interface AccountHome extends EJBHome {
    public Account create(int accno, String customer, double balance)
throws RemoteException, CreateException;

    public Account findByPrimaryKey(Integer pk)
throws RemoteException, FinderException;

    public Account findByNumber(int accno)
throws RemoteException, FinderException;

    public Enumeration findLargeAccounts(double val)
throws RemoteException, FinderException;

}
```

8.3. The Component Interface

8.3.1. Business Methods

The Component Interface is the client's view of an instance of the Entity Bean. It is what is returned to the client by the Home interface after creating or finding an Entity Bean instance. This interface contains the business methods of the Enterprise Bean. The interface must extend the `javax.ejb.EJBObject` interface if it is remote, or the `javax.ejb.EJBLocalObject` if it is local. The methods of a remote component interface must follow the rules for Java RMI. For each method defined in this component interface, there must be a matching method of the bean implementation class (same arguments number and types, same return type, same exceptions except for RemoteException).

8.3.1.1. Component Interface Example

```
public interface Account extends EJBObject {
    public double getBalance() throws RemoteException;
    public void setBalance(double d) throws RemoteException;
    public String getCustomer() throws RemoteException;
    public void setCustomer(String c) throws RemoteException;
    public int getNumber() throws RemoteException;
}
```

8.4. The Primary Key Class

The Primary Key class is necessary for entity beans only. It encapsulates the fields representing the primary key of an Entity Bean in a single object. If the primary key in the database table is composed of a single column with a basic data type, the simplest way to define the primary key in the bean is to use a standard Java class (for example, `java.lang.Integer` or `java.lang.String`). This must have the same type as a field in the bean class. It is not possible to define it as a primitive field (for example, int, float or boolean). Then, it is only necessary to specify the type of the primary key in the deployment descriptor:

`<prim-key-class>java.lang.Integer</prim-key-class>`

And, for container-managed persistence, the field which represents the primary key:

`<primkey-field>accno</primkey-field>`

The alternative way is to define its own Primary Key class, described as follows:

The class must be serializable and must provide suitable implementation of the `hashcode()` and `equals(Object)` methods.

For `container-managed persistence`, the following rules apply:

• The fields of the primary key class must be declared as public.

• The primary key class must have a public default constructor.

• The names of the fields in the primary key class must be a subset of the names of the container-managed fields of the Enterprise Bean.

8.4.1. Primary Key Class Example

```
public class AccountBeanPK implements java.io.Serializable {
    public int accno;
    public AccountBeanPK(int accno) { this.accno = accno; }
    public AccountBeanPK() { }

    public int hashcode() { return accno; }
    public boolean equals(Object other) {
 ...
    }
}
```

8.4.1.1. Special Case: Automatic Generation of Primary Keys Field

There are two ways to manage automatic primary keys with JOnAS. The first method is closer to the EJB specification (that is, an automatic PK is a hidden field, the type of which is not even known by the application). In the second method, the idea is to declare a typical PK CMP field of the type `java.lang.Integer` as `automatic`. These two cases are described below.

8.4.1.1.1. Method 1: Standard Automatic Primary Keys (from JOnAS 4.0.0)

In this case, an automatic PK is a hidden field, the type of which is not known by the application. All that is necessary is to stipulate in the standard deployment descriptor that this EJB has an automatic PK; you do this by specifying `java.lang.Object` as primkey-class. The primary key will be completely hidden from the application (no CMP field, no getter/setter method). This is valid for both CMP 2.x and CMP1 entity beans. The container will create an internal CMP field and generate its value when the Entity Bean is created.

8.4.1.1.1.1. Method 1 Example:

Standard deployment descriptor:

```
<entity>
  ...
  <ejb-name>AddressEJB</ejb-name>
  <local-home>com.titan.address.AddressHomeLocal</local-home>
  <local>com.titan.address.AddressLocal</local>
  <ejb-class>com.titan.address.AddressBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Object</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Cmp2_Address</abstract-schema-name>
  <cmp-field><field-name>street</field-name></cmp-field>
  <cmp-field><field-name>city</field-name></cmp-field>
  <cmp-field><field-name>state</field-name></cmp-field>
  <cmp-field><field-name>zip</field-name></cmp-field>
```

Address Bean Class extract:

```
// Primary key is not explicitly initialized during ejbCreate method
// No cmp field corresponds to the primary key
public Integer ejbCreateAddress(String street, String city,
    String state, String zip ) throws javax.ejb.CreateException {
  setStreet(street);
  setCity(city);
  setState(state);
  setZip(zip);
```

```
  return null;
}
```

If nothing else is specified and the JOnAS default CMP 2 database mapping is used, the JOnAS container generates a database column with the name `JPK_` to handle this PK. However, it is possible to specify in the JOnAS-specific Deployment Descriptor the name of the column that will be used to store the PK value in the table. You can do this as follows using the specific <automatic-pk-field-name> element (this technique is necessary for CMP2 legacy and for CMP1):

JOnAS-specific deployment descriptor:

```
<jonas-ejb-jar xmlns="http://www.objectweb.org/jonas/ns"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.objectweb.org/jonas/ns
  http://www.objectweb.org/jonas/ns/jonas-ejb-jar_4_0.xsd" >
<jonas-entity>
   <ejb-name>AddressEJB</ejb-name>
   <jdbc-mapping>
      <jndi-name>jdbc_1</jndi-name>
      <automatic-pk-field-name>FieldPkAuto</automatic-pk-field-name>
   </jdbc-mapping>
</jonas-entity>
```

8.4.1.1.2. Method 2: CMP Field as Automatic Primary Key (from JOnAS 3.3.x)

The idea here is to declare a typical PK CMP field of type `java.lang.Integer` as `automatic`. By doing this, the field will no longer appear in `create` methods and its value will be automatically generated by the container at the EJB instance-creation time. However, it is still a cmp field, with getter/setter methods, and is accessible from the application.

8.4.1.1.2.1. Method 2 Example:

In the standard deployment descriptor, there is a typical primary key definition:

```
<entity>
   ...
   <prim-key-class>java.lang.Integer</prim-key-class>
   <cmp-field><field-name>id</field-name></cmp-field>
   <primkey-field>id</primkey-field>
```

In the JOnAS-specific deployment descriptor, specify that this PK is automatic:

```
<jonas-entity>
   ...
   <jdbc-mapping>
   <automatic-pk>true</automatic-pk>
```

> **Note**
>
> The automatic primary key is given a unique ID by an algorithm that is based on the system time; therefore, IDs may not be sequential.
>
> Important restriction: This algorithm will not work when used inside a cluster with a same entity bean managed in several JOnAS servers.

8.5. The Enterprise Bean Class

The EJB implementation class implements the bean's business methods of the component interface and the methods dedicated to the EJB environment, the interface of which is explicitly defined in the EJB specification. The class must implement the `javax.ejb.EntityBean` interface, must be defined as public, cannot be `abstract` for CMP 1.1, and must be abstract for CMP 2.0 (in this case, the abstract methods are the get and set accessor methods of the bean's cmp and cmr fields). Following is a list of the EJB-environment dedicated methods that the EJB provider must develop.

The first set of methods are those corresponding to the create and find methods of the Home interface:

- `public PrimaryKeyClass` *ejbCreate*`(...);`

  This method is invoked by the container when a client invokes the corresponding create operation on the enterprise Bean's home interface. The method should initialize instance's variables from the input arguments. The returned object should be the primary key of the created instance. For bean-managed persistence, the bean provider should develop here the JDBC code to create the corresponding data in the database. For container-managed persistence, the container will perform the database insert *after* the ejbCreate method completes and the return value should be `null`.

- `public void` *ejbPostCreate*`(...);`

  There is a matching ejbPostCreate method (same input parameters) for each ejbCreate method. The container invokes this method after the execution of the matching ejbCreate(...) method. During the ejbPostCreate method, the object identity is available.

- `public <PrimaryKeyClass or Collection>` *ejbFind<method>* `(...); // for bean managed persistence only`

  The container invokes this method on a bean instance that is not associated with any particular object identity (some kind of class method ...) when the client invokes the corresponding method on the Home interface. The implementation uses the arguments to locate the requested object(s) in the database and returns a primary key (or a collection thereof). Currently, collections will be represented as `java.util.Enumeration` objects or `java.util.Collection`. The mandatory `FindByPrimaryKey` method takes as argument a primary key type value and returns a primary key object (it verifies that the corresponding entity exists in the database). For *container-managed persistence*, the bean provider does not have to write these finder methods; they are generated at deployment time by the EJB platform tools. The information needed by the EJB platform for automatically generating these finder methods should be provided by the bean programmer. The EJB 1.1 specification does not specify the format of this finder method description; for `JOnAS`, the CMP 1.1 finder methods description should be provided in the JOnAS-specific deployment descriptor of the Entity Bean (as an SQL query). Refer to Section 8.7 *Configuring Database Access for Container-Managed Persistence*. The EJB 2.0 specification defines a standard way to describe these finder methods, that is, in the standard deployment descriptor, as an EJB-QL query. Also refer to Section 8.7 *Configuring Database Access for Container-Managed Persistence*. Then, the methods of the `javax.ejb.EntityBean` interface must be implemented:

  - `public void` *setEntityContext*`(EntityContext ic);`

    Used by the container to pass a reference to the EntityContext to the bean instance. The container invokes this method on an instance after the instance has been created. Generally, this method is used to store this reference in an instance variable.

  - `public void` *unSetEntityContext*`();`

    Unset the associated entity context. The container calls this method before removing the instance. This is the last method the container invokes on the instance.

  - `public void` *ejbActivate*`();`

    The container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB object. This method transitions the instance to the ready state.

- `public void` *ejbPassivate*`();`

  The container invokes this method on an instance before the instance becomes dissociated with a specific EJB object. After this method completes, the container will place the instance into the pool of available instances.

- `public void` *ejbRemove*`();`

  This method is invoked by the container when a client invokes a remove operation on the Enterprise Bean. For entity beans with `bean-managed persistence`, this method should contain the JDBC code to remove the corresponding data in the database. For `container-managed persistence`, this method is called *before* the container removes the entity representation in the database.

- `public void` *ejbLoad*`();`

  The container invokes this method to instruct the instance to synchronize its state by loading it from the underlying database. For `bean-managed persistence`, the EJB provider should code at this location the JDBC statements for reading the data in the database. For `container-managed persistence`, loading the data from the database will be done automatically by the container just *before* ejbLoad is called, and the ejbLoad method should only contain some "after loading calculation statements."

- `public void` *ejbStore*`();`

  The container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database. For `bean-managed persistence`, the EJB provider should code at this location the JDBC statements for writing the data in the database. For entity beans with `container-managed persistence`, this method should only contain some "pre-store statements," since the container will extract the container-managed fields and write them to the database just *after* the ejbStore method call.

8.5.1. Enterprise Bean Class Example

The following examples are for container-managed persistence with EJB 1.1 and EJB 2.0. For bean-managed persistence, refer to the examples delivered with your specific platform.

8.5.1.1. CMP 1.1

```
package eb;

import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.EJBException;

public class AccountImplBean implements EntityBean {

    // Keep the reference on the EntityContext
    protected EntityContext entityContext;

    // Object state
    public Integer accno;
    public String customer;
    public double balance;

    public Integer ejbCreate(int val_accno, String val_customer,
        double val_balance) {
```

```
    // Init object state
    accno = new Integer(val_accno);
    customer = val_customer;
    balance = val_balance;
    return null;
}

public void ejbPostCreate(int val_accno, String val_customer,
    double val_balance) {
    // Nothing to be done for this simple example.
}

public void ejbActivate() {
    // Nothing to be done for this simple example.
}

public void ejbLoad() {
    // Nothing to be done for this simple example,
    // in implicit persistence.
}

public void ejbPassivate() {
    // Nothing to be done for this simple example.
}

public void ejbRemove() {
    // Nothing to be done for this simple example,
    // in implicit persistence.
}

public void ejbStore() {
    // Nothing to be done for this simple example,
    // in implicit persistence.
}

public void setEntityContext(EntityContext ctx) {
    // Keep the entity context in object
    entityContext = ctx;
}

public void unsetEntityContext() {
    entityContext = null;
}

public double getBalance() {
    return balance;
}

public void setBalance(double d) {
    balance = balance + d;
}

public String getCustomer() {
    return customer;
}

public void setCustomer(String c) {
    customer = c;
}

public int getNumber()  {
```

```
        return accno.intValue();
    }
}
```

8.5.1.2. CMP 2.0

```java
import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
import javax.ejb.EJBException;

public abstract class AccountImpl2Bean implements EntityBean {

    // Keep the reference on the EntityContext
    protected EntityContext entityContext;


    /*==== Abstract set and get accessors for cmp fields ====*/

    public abstract String getCustomer();
    public abstract void setCustomer(String customer);

    public abstract double getBalance();
    public abstract void setBalance(double balance);

    public abstract int getAccno();
    public abstract void setAccno(int accno);

    /*================= ejbCreate methods ===================*/


    public Integer ejbCreate(int val_accno,
        String val_customer, double val_balance)
        throws CreateException {

        // Init object state
        setAccno(val_accno);
        setCustomer(val_customer);
        setBalance(val_balance);
        return null;
    }

    public void ejbPostCreate(int val_accno, String val_customer,
        double val_balance) {
        // Nothing to be done for this simple example.
    }


    /*========= javax.ejb.EntityBean implementation =========*/

    public void ejbActivate() {
        // Nothing to be done for this simple example.
    }

    public void ejbLoad() {
        // Nothing to be done for this simple example,
        // in implicit persistence.
```

```
    }

    public void ejbPassivate() {
        // Nothing to be done for this simple example.
    }

    public void ejbRemove() throws RemoveException {
        // Nothing to be done for this simple example,
        // in implicit persistence.
    }

    public void ejbStore() {
        // Nothing to be done for this simple example,
        // in implicit persistence.
    }

    public void setEntityContext(EntityContext ctx) {

        // Keep the entity context in object
        entityContext = ctx;
    }

    public void unsetEntityContext()  {
        entityContext = null;
    }

    /**
     * Business method to get the Account number
     */
    public int getNumber()  {
        return getAccno();
    }

}
```

8.6. Writing Database Access Operations (Bean-Managed Persistence)

For `bean-managed persistence`, data access operations are developed by the bean provider using the JDBC interface. However, getting database connections must be obtained through the `javax.sql.DataSource` interface on a datasource object provided by the EJB platform. This is mandatory since the EJB platform is responsible for managing the connection pool and for transaction management. Thus, to get a JDBC connection, in each method performing database operations, the bean provider must:

- Call the `getConnection(...)` method of the DataSource object to obtain a connection to perform the JDBC operations in the current transactional context (if there are JDBC operations)
- Call the `close()` method on this connection after the database access operations so that the connection can be returned to the connection pool (and be dissociated from the potential current transaction).

A method that performs database access must always contain the getConnection and close statements, as follows:

```
public void doSomethingInDB (...) {
    conn = dataSource.getConnection();
    ... // Database access operations
```

```
    conn.close();
}
```

A DataSource object associates a JDBC driver with a database (as an ODBC datasource). It is created and registered in JNDI by the EJB server at launch time (refer also to Chapter 4 *Configuring JDBC DataSources*).

A DataSource object is a resource manager connection factory for `java.sql.Connection` objects, which implements connections to a database management system. The Enterprise Bean code refers to resource factories using logical names called *Resource manager connection factory references*. The resource manager connection factory references are special entries in the Enterprise Bean environment. The bean provider must use resource manager connection factory references to obtain the datasource object as follow:

• Declare the resource reference in the standard deployment descriptor using a `resource-ref` element.

• Look up the datasource in the Enterprise Bean environment using the JNDI interface (refer to Chapter 12 *Enterprise Bean Environment*).

The deployer binds the resource manager connection factory references to the actual resource factories that are configured in the server. This binding is done in the JOnAS-specific deployment descriptor using the jonas-resource element.

8.6.1. Database Access Operation Example

The declaration of the resource reference in the standard deployment descriptor looks like the following:

```
<resource-ref>
<res-ref-name>jdbc/AccountExplDs</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
```

The <res-auth> element indicates which of the two resource manager authentication approaches is used:

• `Container`: the deployer sets up the sign-on information.

• `Bean`: the bean programmer should use the getConnection method with user and password parameters.

The JOnAS-specific deployment descriptor must map the environment JNDI name of the resource to the actual JNDI name of the resource object managed by the EJB server. This is done in the <jonas-resource> element.

```
<jonas-entity>
  <ejb-name>AccountExpl</ejb-name>
  <jndi-name>AccountExplHome</jndi-name>
  <jonas-resource>
    <res-ref-name>jdbc/AccountExplDs</res-ref-name>
    <jndi-name>jdbc_1</jndi-name>
  </jonas-resource>
</jonas-entity>
```

The ejbStore method of the same Account example with bean-managed persistence is shown in the following example. It performs JDBC operations to update the database record representing the state of the Entity Bean instance. The JDBC connection is obtained from the datasource associated with the

bean. This datasource has been instantiated by the EJB server and is available for the bean through its resource reference name, which is defined in the standard deployment descriptor.

In the bean, a reference to a datasource object of the EJB server is initialized:

```
it = new InitialContext();

ds = (DataSource)it.lookup("java:comp/env/jdbc/AccountExplDs");
```

Then, this datasource object is used in the implementation of the methods performing JDBC operations, such as ejbStore, as illustrated in the following:

```
public void ejbStore
    Connection conn = null;
    PreparedStatement stmt = null;
    try { // get a connection
        conn = ds.getConnection();
        // store Object state in DB
        stmt = conn.prepareStatement("update account
          set customer=?,balance=? where accno=?");
        stmt.setString(1, customer);
        stmt.setDouble(2, balance);
        Integer pk = (Integer)entityContext.getPrimaryKey();
        stmt.setInt(3, pk.accno);
        stmt.executeUpdate();
    } catch (SQLException e) {
        throw new javax.ejb.EJBException("Failed to store bean
          to database", e);
    } finally {
        try {
            if (stmt != null) stmt.close();    // close statement
            if (conn != null) conn.close();    // release connection
        } catch (Exception ignore) {}
    }
}
```

Note that the close statement instruction may be important if the server is intensively accessed by many clients performing Entity Bean access. If the statement is not closed in the finally block, since stmt is in the scope of the method, it will be deleted at the end of the method (and the close will be implicitly done). However, it may be some time before the Java garbage collector deletes the statement object. Therefore, if the number of clients performing Entity Bean access is important, the DBMS may raise a "too many opened cursors" exception (a JDBC statement corresponds to a DBMS cursor). Since connection pooling is performed by the platform, closing the connection will not result in a physical connection close, therefore opened cursors will not be closed. Thus, it is preferable to explicitly close the statement in the method.

It is a good programming practice to put the JDBC connection and JDBC statement close operations in a finally block of the try statement.

8.7. Configuring Database Access for Container-Managed Persistence

The standard way to indicate to an EJB platform that an Entity Bean has container-managed persistence is to fill the <persistence-type> tag of the deployment descriptor with the value "container," and to fill the <cmp-field> tag of the deployment descriptor with the list of container-managed fields (the fields that the container will have in charge to make persistent). The CMP version (1.x or 2.x) should also be specified in the <cmp-version> tag. In the textual format of the deployment descriptor, this is represented by the following lines:

```
<persistence-type>container</persistence-type>
<cmp-version>1.x</cmp-version>
<cmp-field>
  <field-name>fieldOne</field-name>
</cmp-field>
<cmp-field>
  <field-name>fieldTwo</field-name>
</cmp-field>
```

With `container-managed persistence` the programmer need not develop the code for accessing the data in the relational database; this code is included in the container itself (generated by the platform tools). However, for the EJB platform to know how to access the database and which data to read and write in the database, two types of information must be provided with the bean:

- First, the container must know which database to access and how to access it. To do this, the only required information is the *name of the DataSource* that will be used to get the JDBC connection. For container-managed persistence, only one DataSource per bean should be used.

- Then, it is necessary to know *the mapping of the bean fields to the underlying database* (which table, which column). For CMP 1.1 or CMP 2.0, this mapping is specified by the deployer in the JOnAS-specific deployment descriptor. Note that for CMP 2.0, this mapping may be entirely generated by JOnAS.

The EJB specification does not specify how this information should be provided to the EJB platform by the bean deployer. Therefore, what is described in the remainder of this section is *specific to JOnAS*.

For CMP 1.1, the bean deployer is responsible for defining the mapping of the bean fields to the database table columns. The name of the DataSource can be set at deployment time, since it depends on the EJB platform configuration. This database configuration information is defined in the JOnAS-specific deployment descriptor via the `jdbc-mapping` element. The following example defines the mapping for a CMP 1.1 Entity Bean:

```
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
  <jdbc-table-name>accountsample</jdbc-table-name>
  <cmp-field-jdbc-mapping>
    <field-name>mAccno</field-name>
    <jdbc-field-name>accno</jdbc-field-name>
  </cmp-field-jdbc-mapping>
  <cmp-field-jdbc-mapping>
    <field-name>mCustomer</field-name>
    <jdbc-field-name>customer</jdbc-field-name>
  </cmp-field-jdbc-mapping>
  <cmp-field-jdbc-mapping>
    <field-name>mBalance</field-name>
    <jdbc-field-name>balance</jdbc-field-name>
  </cmp-field-jdbc-mapping>
</jdbc-mapping>
```

`jdbc_1` is the JNDI name of the DataSource object identifying the database. `accountsample` is the name of the table used to store the bean instances in the database. `mAccno`, `mCustomer`, and `mBalance` are the names of the container-managed fields of the bean to be stored in the `accno`, `customer`, and `balance` columns of the `accountsample` table. This example applies to `container-managed persistence`. For `bean-managed persistence`, the database mapping does not exist.

For a CMP 2.0 Entity Bean, only the `jndi-name` element of the `jdbc-mapping` is mandatory, since the mapping may be generated automatically:

```
<jdbc-mapping>
```

```
  <jndi-name>jdbc_1</jndi-name>
  <cleanup>create</cleanup>
</jdbc-mapping>
```

For an explicit mapping definition, refer to Section 8.12 *JOnAS Database Mapping (Specific Deployment Descriptor)*.

For a CMP 2.0 Entity Bean, the JOnAS-specific deployment descriptor contains an additional element, `cleanup`, at the same level as the `jdbc-mapping` element, which can have one of the following values:

`removedata`

At bean loading time, the content of the tables storing the bean data is deleted.

`removeall`

At bean loading time, the tables storing the bean data are dropped (if they exist) and created.

`none`

Do nothing.

`create`

Default value (if the element is not specified), at bean loading time, the tables for storing the bean data are created if they do not exist.

For CMP 1.1, the `jdbc-mapping` element can also contain information defining the behavior of the implementation of a `find<method>` method (that is, the `ejbFind<method>` method, that will be generated by the platform tools). This information is represented by the `finder-method-jdbc-mapping` element.

For each finder method, this element provides a way to define an SQL `WHERE` clause that will be used in the generated finder method implementation to query the relational table storing the bean entities. Note that the table column names should be used, not the bean field names. Example:

```
<finder-method-jdbc-mapping>
  <jonas-method>
    <method-name>findLargeAccounts</method-name>
  </jonas-method>
  <jdbc-where-clause>where balance >
    ?</jdbc-where-clause>
</finder-method-jdbc-mapping>
```

The previous finder method description will cause the platform tools to generate an implementation of ejbFindLargeAccount(double arg) that returns the primary keys of the Entity Bean objects corresponding to the tuples returned by the `select ... from Account where balance > ?`, where '?' will be replaced by the value of the first argument of the findLargeAccount method. If several '?' characters appear in the provided `WHERE` clause, this means that the finder method has several arguments and the '?' characters will correspond to these arguments, adhering to the order of the method signature.

In the `WHERE` clause, the parameters can be followed by a number, which specifies the method parameter number that will be used by the query in this position.

Example: The WHERE clause of the following finder method can be:

```
Enumeration findByTextAndDateCondition(String text, java.sql.Date date)

WHERE (description like ?1 OR summary like ?1) AND (?2 > date)
```

Note that a `<finder-method-jdbc-mapping>` element for the `findByPrimaryKey` method is not necessary, since the meaning of this method is known.

Additionally, note that for CMP 2.0, the information defining the behavior of the implementation of a `find<method>` method is located in *the standard deployment descriptor*, as an EJB-QL query (that is, this is not JOnAS-specific information). The same finder method example in CMP 2.0:

```
<query>
  <query-method>
    <method-name>findLargeAccounts</method-name>
    <method-params>
        <method-param>double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(o) FROM accountsample o WHERE o.balance
    > ?1</ejb-ql>
</query>
```

8.8. Using CMP2.0 Persistence

The following sections highlight the main differences between CMP as defined in EJB 2.0 specification (called CMP2.0) and CMP as defined in EJB 1.1 specification (called CMP1.1). Major new features in the standard development and deployment of CMP2.0 Entity Beans are listed (comparing them to CMP1.1), along with JOnAS-specific information. Mapping CMP2.0 Entity Beans to the database is described in detail. Note that the database mapping can be created entirely by JOnAS, in which case the JOnAS-specific deployment descriptor for an Entity Bean should contain only the datasource and the element indicating how the database should be initialized.

8.9. Standard CMP2.0 Aspects

This section briefly describes the new features available in CMP2.0 as compared to CMP 1.1, and how these features change the development of Entity Beans.

8.9.1. Entity Bean Implementation Class

The EJB implementation class:

- Implements the bean's business methods of the component interface
- Implements the methods dedicated to the EJB environment (the interface of which is explicitly defined in the EJB specification)
- Defines the abstract methods representing both the persistent fields (cmp-fields) and the relationship fields (cmr-fields).

The class must implement the `javax.ejb.EntityBean` interface, be defined as `public`, and be `abstract` (which is not the case for CMP1.1, where it must *not* be `abstract`). The abstract methods are the `get` and `set` accessor methods of the bean `cmp` and `cmr` fields. Refer to the examples and details in Chapter 8 *Developing Entity Beans*.

8.9.2. Standard Deployment Descriptor

The standard way to indicate to an EJB platform that an Entity Bean has container-managed persistence is to fill the `<persistence-type>` tag of the deployment descriptor with the value `container`, and to fill the `<cmp-field>` tags of the deployment descriptor with the list of container-managed fields (the fields that the container will have in charge to make persistent) and the `<cmr-field>` tags identifying the relationships. The CMP version (1.x or 2.x) should also be specified in the `<cmp-version>` tag. This is represented by the following lines in the deployment descriptor:

```
<persistence-type>container</persistence-type>
<cmp-version>1.x</cmp-version>
<cmp-field>
  <field-name>fieldOne</field-name>
</cmp-field>
<cmp-field>
  <field-name>fieldTwo</field-name>
</cmp-field>
```

⚠️ **Warning**

> To run CMP1.1-defined Entity Beans on an EJB2.0 platform, such as JOnAS 3.x, you must introduce the `<cmp-version>` element in your deployment descriptors, because the default cmp-version value (if not specified) is 2.x.

Note that for CMP 2.0, the information defining the behavior of the implementation of a `find<method>` method is located in *the standard deployment descriptor* as an EJB-QL query (this is *not* JOnAS-specific information). For CMP 1.1, this information is located in the JOnAS-specific deployment descriptor as an SQL `WHERE` clause specified in a `<finder-method-jdbc-mapping>` element.

The following example shows a finder method in CMP 2.0 for a `findLargeAccounts(double val)` method defined on the Account Entity Bean of the JOnAS `eb` example.

```
<query>
  <query-method>
    <method-name>findLargeAccounts</method-name>
    <method-params>
        <method-param>double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(o) FROM accountsample o
    WHERE o.balance > ?1</ejb-ql>
</query>
```

8.10. JOnAS EJBQL extension

8.10.1. LIMIT clause

The `LIMIT` feature has been added to the standard EJBQL query language. This feature enables you to retrieve just a portion of the results generated by the rest of a query.

The syntax of the `LIMIT` clause is:

```
limit_clause ::= LIMIT limit_expression (, limit_expression )?
limit_expression ::= integer_literal | input_parameter
```

The first `limit_expression` corresponds to the `start_at` range and the second one to the `size` range.

The `limit_clause` is the last clause of the query:

```
ejbql ::= select_clause from_clause [where_clause] [orderby_clause]
    [limit_clause]
```

Example:

```
SELECT OBJECT(c) FROM jt2_Customer AS c ORDER BY c.id LIMIT ?1, 20
```

Note

This feature is currently not implemented on all the database types supported by JORM/MEDOR.

8.11. JOnAS Database Mappers

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework (see http://www.objectweb.org/jorm/index.html). JORM itself relies on JOnAS DataSources (specified in DataSource properties files) for connecting to the actual database. JORM must adapt its object-relational mapping to the underlying database, for which it makes use of adapters called *mappers*. Thus, for each type of database (and more precisely for each JDBC driver), the corresponding mapper must be specified in the DataSource. This is the purpose of the `datasource.mapper` property of the DataSource properties file. Note that all JOnAS-provided DataSource properties files (in `$JONAS_ROOT/conf`) already contain this property with the correct mapper.

For the JORM database mapper `datasource.mapper`, the possible values are:

- rdb: generic mapper (JDBC standard driver ...)
- rdb.postgres: mapper for PostgreSQL
- rdb.oracle8: mapper for Oracle 8 and lesser versions
- rdb.oracle: mapper for Oracle 9
- rdb.mysql: mapper for MySQL
- `rdb.cloudscape`: Cloudscape
- `rdb.db2`: DB2
- `rdb.firebird`: Firebird
- `rdb.hsql`: HSQL
- `rdb.sapdb`: SAP DB
- `rdb.sybase`: Sybase

Contact the JOnAS team (jonas-team@objectweb.org) to obtain a mapper for other databases.

The container code generated at deployment is now independent of the JORM mappers. Until JOnAS 4.1.4, the container code generated at deployment (GenIC or ejbjar ant task) was dependent on this mapper. It was possible to deploy (generate container code) a bean for several mappers in order to change the database (that is, the DataSource file) without redeploying the bean. These mappers were specified as the `mappernames` argument of the GenIC command or as the `mappernames` attribute of the JOnAS ANT ejbjar task. The value was a comma-separated list of mapper names for which the container classes were generated. These mappernames options are now deprecated.

8.12. JOnAS Database Mapping (Specific Deployment Descriptor)

You can specify the mapping to the database of Entity Beans and their relationships in the JOnAS-specific deployment descriptor, in `jonas-entity` elements, and in `jonas-ejb-relation` elements. Since JOnAS is able to generate the database mapping, all the elements of the JOnAS-specific deployment descriptor defined in this section (which are sub-elements of `jonas-entity` or `jonas-ejb-relation`) are optional, except those for specifying the datasource and the initialization mode (that is, the `jndi-name` of `jdbc-mapping` and `cleanup`). The default values of these mapping elements, provided in this section, define the JOnAS-generated database mapping.

8.12.1. Specifying and Initializing the Database

To specify the database within which a CMP 2.0 Entity Bean is stored, use the `jndi-name` element of the `jdbc-mapping`. This is the JNDI name of the DataSource representing the database storing the Entity Bean.

```
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
</jdbc-mapping>
```

For a CMP 2.0 Entity Bean, the JOnAS-specific deployment descriptor contains an additional element, `cleanup`, to be specified before the `jdbc-mapping` element. The `cleanup` element can have one of the following values:

`removedata`

At bean loading time, delete the content of the tables storing the bean data

`removeall`

At bean loading time, drop the tables storing the bean data (if they exist) and re-create them

`none`

Do nothing

`create`

Default value (if the element is not specified). At bean loading time, create the tables for storing the bean data (if they do not exist).

It may be useful for testing purposes to delete the database data each time a bean is loaded. To do this, the part of the JOnAS-specific deployment descriptor related to the Entity Bean may look like the following:

```
<cleanup>removedata</cleanup>
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
</jdbc-mapping>
```

8.12.2. CMP fields mapping

Mapping CMP fields in CMP2.0 is similar to that of CMP1.1, but in CMP2.0 it is also possible to specify the SQL type of a column. Usually this SQL type is used if JOnAS creates the table (the `create` value of the `cleanup` element) and if the JORM SQL type is not appropriate.

8.12.2.1. Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>idA</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>f</field-name>
  </cmp-field>
.....
</entity>
.....
```

8.12.2.2. Database Mapping

t_A	
c_idA	c_f
...	...

8.12.2.3. JOnAS Deployment Descriptor

```
.....
<jonas-entity>
  <ejb-name>A</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_A</jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idA</field-name>
      <jdbc-field-name>c_idA</jdbc-field-name>
    </cmp-field-jdbc-mapping>
    <cmp-field-jdbc-mapping>
      <field-name>f</field-name>
      <jdbc-field-name>c_f</jdbc-field-name>
      <sql-type>varchar(40)</sql-type>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
</jonas-entity>
.....
```

jndi-name	Mandatory
jdbc-table-name	Optional. Default value is the *upper-case* CMP2 abstract-schema-name, or the CMP1 EJB-name, suffixed by "_".
cmp-field-jdbc-mapping	Optional.
jdbc-field-name	Optional. Default value is the field-name suffixed by "_". "idA_" and "f_" in the example.
sql-type	Optional. Default value defined by JORM.

Table 8-1. CMP fields mapping: Default values

8.12.3. CMR fields mapping to primary-key-fields (simple pk)

8.12.3.1. 1-1 unidirectional relationships

8.12.3.1.1. Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>idA</field-name>
  </cmp-field>
  <primkey-field>idA</primkey-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <cmp-field>
    <field-name>idB</field-name>
  </cmp-field>
  <primkey-field>idB</primkey-field>
  .....
</entity>
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
        <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
```

```
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.1.2. Database Mapping

t_A	
c_idA	cfk_idB
...	...

t_B
c_idB
...

There is a foreign key in the table of the bean that owns the CMR field.

8.12.3.1.3. JOnAS Deployment Descriptor

```
.....
<jonas-entity>
  <ejb-name>A</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_A</jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idA</field-name>
      <jdbc-field-name>c_idA</jdbc-field-name>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
</jonas-entity>
.....
<jonas-entity>
  <ejb-name>B</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_B</jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idB</field-name>
      <jdbc-field-name>c_idB</jdbc-field-name>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
```

```
   .....
</jonas-entity>
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

`foreign-key-jdbc-name` is the column name of the foreign key in the table of the source bean of the relationship-role.

In this example, where the destination bean has a primary-key-field, it is possible to deduce that this foreign-key-jdbc-name column is to be associated with the column of this primary-key-field in the table of the destination bean.

`jonas-ejb-relation`	Optional
`foreign-key-jdbc-name`	Optional. Default value is the abstract-schema-name of the destination bean, suffixed by "_" and by its primary-key-field. `B_idb` in the example.

Table 8-2. 1-1 unidirectional relationships: Default values

8.12.3.2. 1-1 bidirectional relationships

In contrast to 1-1 unidirectional relationships, there is a CMR field in both of the beans, thus making two types of mapping possible.

8.12.3.2.1. Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
```

```
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.2.2. Database Mapping

Two mappings are possible. One of the tables may hold a foreign key.

Case 1:

t_A	
c_idA	cfk_idB
...	...

t_B
c_idB
...

Case 2:

t_A	
c_idA	
...	

t_B	
c_idB	cfk_idA
...	...

8.12.3.2.3. JOnAS Deployment Descriptor

Case 1:

```
    .....
    <jonas-ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <jonas-ejb-relationship-role>
        <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <foreign-key-jdbc-mapping>
          <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
        </foreign-key-jdbc-mapping>
      </jonas-ejb-relationship-role>
    </jonas-ejb-relation>
```

```
    .....
```

Case 2:

```
    .....
    <jonas-ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <jonas-ejb-relationship-role>
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <foreign-key-jdbc-mapping>
          <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
        </foreign-key-jdbc-mapping>
      </jonas-ejb-relationship-role>
    </jonas-ejb-relation>
    .....
```

For the default mapping, the foreign key is in the table of the source bean of the first ejb-relationship-role of the ejb-relation. In the example, the default mapping corresponds to case 1, since the ejb-relationship-role `a2b` is the first defined in the ejb-relation `a-b`. Then, the default values are similar to those of the 1-1 unidirectional relationship.

8.12.3.3. 1-N unidirectional relationships

8.12.3.3.1. Standard Deployment Descriptor

```
    .....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
          <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.3.2. Database Mapping

t_A	
c_idA	
...	

t_B	
c_idB	cfk_idA
...	...

In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (that is, in the table of the source bean of the relationship role with multiplicity many), `t_B`.

8.12.3.3.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

`jonas-ejb-relation`	Optional
`foreign-key-jdbc-name`	Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (that is, the source bean of the relationship role with multiplicity one) suffixed by "_" and by its primary-key-field. `A_ida` in the example.

Table 8-3. 1-N unidirectional relationships: Default values

8.12.3.4. 1-N bidirectional relationships

Similar to 1-N unidirectional relationships, but with a CMR field in each bean.

8.12.3.4.1. Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
```

```
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.4.2. Database Mapping

t_A	
c_idA	
...	

t_B	
c_idB	cfk_idA
...	...

In this case, the foreign key must be in the table of the bean that is on the "many" side of the relationship (that is, in the table of the source bean of the relationship role with multiplicity many), `t_B`.

8.12.3.4.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
```

```
</jonas-ejb-relation>
.....
```

jonas-ejb-relation	Optional
foreign-key-jdbc-name	Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (that is, the source bean of the relationship role with multiplicity one), suffixed by "_" and its primary-key-field. `A_ida` in the example.

Table 8-4. 1-N bidirectional relationships: Default values

8.12.3.5. N-1 Unidirectional Relationships

Similar to 1-N unidirectional relationships, but the CMR field is defined on the "many" side of the relationship, that is, on the (source bean of the) relationship role with multiplicity "many."

8.12.3.5.1. Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
          <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.5.2. Database Mapping

t_A	
c_idA	cfk_idB
...	...

t_B
c_idB
...

In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (that is, in table of the source bean of the relationship role with multiplicity many), t_A.

8.12.3.5.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

`jonas-ejb-relation`	Optional
`foreign-key-jdbc-name`	Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (that is, the source bean of the relationship role with multiplicity one) suffixed by "_" and by its primary-key-field. `B_idb` in the example.

Table 8-5. N-1 unidirectional relationships: Default values

8.12.3.6. N-M Unidirectional Relationships

8.12.3.6.1. Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
          <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
```

```
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.6.2. Database Mapping

t_A	
c_idA	
...	

t_B
c_idB
...

tJoin_AB	
cfk_idA	cfk_idB
...	...

In this case, there is a join table composed of the foreign keys of each Entity Bean table.

8.12.3.6.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

`jonas-ejb-relation`	Optional
`jdbc-table-name`	Optional. Default value is built from the abstract-schema-names of the beans, separated by `_`. `A_B` in the example.
`foreign-key-jdbc-name`	Optional. Default value is the abstract-schema-name of the destination bean, suffixed by "_" and by its primary-key-field. `B_idb` and `A_ida` in the example.

Table 8-6. N-M unidirectional relationships: Default values

8.12.3.7. N-M Bidirectional Relationships

Similar to N-M unidirectional relationships, but a CMR field is defined for each bean.

8.12.3.7.1. Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.3.7.2. Database Mapping

t_A
c_idA
...

t_B
c_idB
...

tJoin_AB	
cfk_idA	cfk_idB
...	...

In this case, there is a join table composed of the foreign keys of each Entity Bean table.

8.12.3.7.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

jonas-ejb-relation	Optional
jdbc-table-name	Optional. Default value is built from the abstract-schema-names of the beans, separated by _. A_B in the example.

foreign-key-jdbc-name	Optional. Default value is the abstract-schema-name of the destination bean, suffixed by "_" and by its primary-key-field. `B_idb` and `A_ida` in the example.

Table 8-7. CMR fields mapping to primary-key-fields: Default values

8.12.4. CMR fields Mapping to Composite Primary-Keys

In the case of composite primary keys, the database mapping should provide the capability to specify which column of a foreign key corresponds to which column of the primary key. This is the only difference between relationships based on simple primary keys. For this reason, not all types of relationship are illustrated below.

8.12.4.1. 1-1 Bidirectional Relationships

8.12.4.1.1. Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <prim-key-class>p.PkA</prim-key-class>
  .....
  <cmp-field>
    <field-name>id1A</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2A</field-name>
  </cmp-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <prim-key-class>p.PkB</prim-key-class>
  .....
  <cmp-field>
    <field-name>id1B</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2B</field-name>
  </cmp-field>
  .....
</entity>
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
```

```
      <ejb-name>A</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>b</cmr-field-name>
    </cmr-field>
  </ejb-relationship-role>
  <ejb-relationship-role>
    <!-- B => A -->
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
      <ejb-name>B</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>a</cmr-field-name>
    </cmr-field>
  </ejb-relationship-role>
</ejb-relation>
</relationships>
.....
```

8.12.4.1.2. Database Mapping

Two mappings are possible, one or another of the tables may hold the foreign key.

Case 1:

t_A			
c_id1A	c_id2A	cfk_id1B	cfk_id2B
...	...	...	...

t_B	
c_id1B	c_id2B
...	...

Case 2:

t_A	
c_id1A	c_id2A
...	...

t_B			
c_id1B	c_id2B	cfk_id1A	cfk_id2A
...	...	...	...

8.12.4.1.3. JOnAS Deployment Descriptor

Case 1:

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Case 2:

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

For the default mapping (values), the foreign key is in the table of the source bean of the first ejb-relationship-role of the ejb-relation. In the example, the default mapping corresponds to case 1, since the ejb-relationship-role a2b is the first defined in the ejb-relation a-b.

8.12.4.2. N-M Unidirectional Relationships

8.12.4.2.1. Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>id1A</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2A</field-name>
  </cmp-field>
```

```
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <cmp-field>
    <field-name>id1B</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2B</field-name>
  </cmp-field>
  .....
</entity>
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
           <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

8.12.4.2.2. Database Mapping

t_A	
c_id1A	c_id2A
...	...

t_B	
c_id1B	c_id2B
...	...

tJoin_AB			
cfk_id1A	cfk_id2A	cfk_id1B	cfk_id2B
...	...	...	...

In this case, there is a join table composed of the foreign keys of each Entity Bean table.

8.12.4.2.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

8.13. Tuning a Container for Entity Bean Optimizations

JOnAS must make a compromise between scalability and performance. Towards this end, we have introduced some tags in the JOnAS-specific deployment descriptor. For most applications, there is no need to change the default values for all these tags. For a complete description of the JOnAS-specific deployment descriptor, see `$JONAS_ROOT/xml/jonas-ejb-jar_4_0.xsd` (http://jonas.objectweb.org/current/xml/jonas-ejb-jar_4_0.xsd).

8.13.1. Lock-Policy

The JOnAS ejb container can manage the following lock-policies:

container-serialized

> (Default.) The container ensures the transaction serialization. This policy is suitable for entity beans that have non-transacted methods that can modify the bean state. It works only if the bean is accessed from one JOnAS server (shared = false).

container-serialized-transacted

> The container ensures the transaction serialization. This policy is suitable for most entity beans.

container-read-committed

> This policy is also container-serialized-transacted, except that accesses of outside transactions do not interfere with transactional accesses. This can avoid deadlocks when accessing a bean concurrently with and without a transactional context. The only drawback of this policy is that it consumes more memory (two instances instead of one).

container-read-uncommitted

> All methods share the same instance (as with container-serialized), but there is no synchronization. This policy is interesting for read-only entity beans or if the bean instances are very rarely modified. It will fail if two or more threads try to modify the same instance concurrently.

> This policy is deprecated because it can be replaced by `container-read-write`

container-read-write

> All methods share the same instance (like container-serialized). A synchronization is done only when the instance is modified. No lock is taken while the instance is read only. This policy does not work if the bean is `shared` or for CMP 1.x beans.

> Important: If you deploy CMP1 beans, you should use only these policies:

> - container-serialized
> - container-serialized-transacted
> - read-only
> - container-read-committed
> - database, but only with shared=true.

database

> Let the database deal with transaction isolation. With this policy, you can choose the transaction isolation in your database. This may be interesting for applications that heavily use transactional read-only operations, or when the shared flag is needed. It does not work with all databases, and is expensive in terms of memory.

read-only

> The bean state is never written to the database. If the bean is `shared`, the bean state is read from the database regularly.

Note

If you deploy CMP1 beans, you should use the default policy only (container-serialized), unless your beans are "read-only." In this latter case, you could use container-read-uncommitted.

8.13.2. shared

This flag will be defined as `true` if the bean persistent state can be accessed outside the JOnAS Server. When this flag is `false`, the JOnAS Server can do some optimization, such as not re-reading the bean state before starting a new transaction. The default value is `false` if the lock-policy is `container-serialized`, and `true` in the other cases.

8.13.3. prefetch

This is a CMP2-specific option. The default is false. To optimize further accesses inside the same transaction, set the value to true to cache the data that is buffered after finder methods execution.

Notes

- You cannot set the prefetch option when the lock policy is container-read-uncommitted.
- The prefetch will be used only for methods that have transactional context.

8.13.4. min-pool-size

This optional integer value represents the minimum instances that will be created in the pool when the bean is loaded. This will improve bean instance create time, at least for the first instances. The default value is 0.

8.13.5. max-cache-size

This optional integer value represents the maximum number of instances in memory. The purpose of this value is to keep JOnAS scalable. The default value is "no limit." If you know that instances will not be reused, you should set a very low value to save memory.

8.13.6. is-modified-method-name

To improve performance of CMP 1.1 entity beans, JOnAS implements the isModified extension. Before performing an update, the container calls a method of the bean whose name is identified in the `is-modified-method-name` element of the JOnAS-specific deployment descriptor. This method is responsible for determining if the state of the bean has been changed. By doing this, the container determines if it must store data in the database or not.

Note

This is not required with CMP2 entity beans because the container does this automatically.

8.13.6.1. Example

The bean implementation manages a boolean `isDirty` and implements a method that returns the value of the boolean `isModified`.

```
private transient boolean isDirty;
public boolean isModified() {
  return isDirty;
}
```

The JOnAS-specific deployment descriptor directs the bean to implement an `isModified` method:

```
<jonas-entity>
  <ejb-name>Item</ejb-name>
  <is-modified-method-name>isModified</is-modified-method-name>
  .....
</jonas-entity>
```

Methods that modify the value of the bean must set the flag `isDirty` to `true`. Methods that restore the value of the bean from the database must reset the flag `isDirty` to `false`. Therefore, the flag must be set to `false` in the `ejbLoad()` and `ejbStore()` methods.

8.13.7. passivation-timeout

Entity Bean instances are passivated at the end of the transaction and reactivated at the beginning of the next transaction, when the `shared` flag has been set to true. If the `shared` has been set to false, a passivation will occur only if `max-cache-size` has been reached. In the event that these instances are accessed outside of any transaction, their state is kept in memory to improve performance. However, a passivation will occur in three situations:

• When the bean is unloaded from the server, at a minimum when the server is stopped.

• When a transaction is started on this instance.

• After a configurable timeout. If the bean is always accessed with no transaction, it may be prudent to periodically store the bean state on disk.

This passivation timeout can be configured in the JOnAS-specific deployment descriptor, with the optional tag <passivation-timeout>.

```
<jonas-entity>
  <ejb-name>Item</ejb-name>
  <passivation-timeout>5</passivation-timeout>
  .....
</jonas-entity>
```
Example 8-1. Passivation timeout

This Entity Bean will be passivated every 5 seconds, if not accessed within transactions.

Note

This flag is also used to set the timeout to passivate beans when `max-cache-size` has been reached.

8.13.8. inactivity-timeout

Bean passivation sends the state of the bean to persistent storage and removes only the bean-instance objects that are holding this state from memory. All container objects handling bean access (remote object, interceptors, and so on) are kept in memory so that future access will work, requiring only a reload operation (getting the state).

It may be advantageous to conserve more memory and completely remove the bean instance from memory; you can do this through the <inactivity-timeout> element. This element is used to save memory when a bean has not been used for a long time. If the bean has not been used after the specified time (in seconds), all its container objects are removed from the server. If a client has kept a reference on a remote object and tries to use the removed element, it will receive an exception.

Developing Message-Driven Beans

The EJB 2.1 specification defines a new kind of EJB component for receiving asynchronous messages. This implements some type of "asynchronous EJB component method invocation" mechanism. The Message-Driven Bean (also referred to as MDB in the following) is an Enterprise JavaBean, not an Entity Bean or a Session Bean, that plays the role of a JMS MessageListener.

The EJB 2.1 specification contains detailed information about MDB. The Java Message Service Specification 1.1 contains detailed information about JMS. This chapter focuses on the use of Message-Driven Beans within the JOnAS server.

9.1. Description of a Message-Driven Bean

A Message-Driven Bean is an EJB component that can be considered as a JMS MessageListener, that is, processing JMS messages asynchronously; it implements the onMessage(javax.jms.Message) method, defined in the javax.jms.MessageListener interface. It is associated with a JMS destination, that is, a Queue for "point-to-point" messaging or a Topic for "publish/subscribe." The onMessage method is activated on receipt of messages sent by a client application to the corresponding JMS destination. It is possible to associate a JMS message selector to filter the messages that the Message-Driven Bean should receive.

JMS messages do not carry any context, thus the onMessage method will execute without pre-existing transactional context. However, a new transaction can be initiated at this moment (refer to Section 9.5 *Transactional Aspects* for more details). The onMessage method can call other methods on the MDB itself or on other beans, and can involve other resources by accessing databases or by sending messages. Such resources are accessed the same way as for other beans (entity or session), that is, through resource references declared in the deployment descriptor.

The JOnAS container maintains a pool of MDB instances, allowing large volumes of messages to be processed concurrently. An MDB is similar in some ways to a stateless Session Bean: its instances are relatively short-lived, it retains no state for a specific client, and several instances may be running at the same time.

9.2. Developing a Message-Driven Bean

The MDB class must implement the javax.jms.MessageListener and the javax.ejb.MessageDrivenBean interfaces. In addition to the onMessage method, the following must be implemented:

- A public constructor with no argument.
- public void ejbCreate(): with no arguments, called at the bean-instantiation time. It may be used to allocate some resources, such as connection factories, for example if the bean sends messages, or datasources or if the bean accesses databases.
- public void ejbRemove(): usually used to free the resources allocated in the ejbCreate method.
- public void setMessageDrivenContext(MessageDrivenContext mdc): called by the container after the instance creation, with no transaction context. The JOnAS container provides the bean with a container context that can be used for transaction management, for example, for calling setRollbackOnly(), getRollbackOnly(), getUserTransaction().

The following is an example of an MDB class:

```
public class MdbBean  implements MessageDrivenBean, MessageListener {

    private transient MessageDrivenContext mdbContext;

    public MdbBean() {}

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        mdbContext = ctx;
    }

    public void ejbRemove() {}

    public void ejbCreate() {}

    public void onMessage(Message message) {
        try {
            TextMessage mess = (TextMessage)message;
            System.out.println( "Message received: "+mess.getText());
        }catch(JMSException ex){
            System.err.println("Exception caught: "+ex);
        }
    }
}
```

The destination associated with an MDB is specified in the deployment descriptor of the bean. A destination is a JMS-administered object, accessible via JNDI (the Java Naming and Directory Interface). The description of an MDB in the EJB 2.1 deployment descriptor contains the following elements, which are specific to MDBs:

• The JMS acknowledgement mode: auto-acknowledge or dups-ok-acknowledge (refer to the JMS specification for the definition of these modes)

• An eventual JMS message selector: this is a JMS concept which allows the filtering of the messages sent to the destination

• A message-driven-destination, which contains the destination type (Queue or Topic) and the subscription durability (in the case of Topic)

The following example illustrates such a deployment descriptor:

```
<enterprise-beans>
  <message-driven>
    <description>Describe here the message driven bean Mdb</description>
    <display-name>Message Driven Bean Mdb</display-name>
    <ejb-name>Mdb</ejb-name>
    <ejb-class>samplemdb.MdbBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-selector>Weight >= 60.00 AND LName
      LIKE 'Sm_th'</message-selector>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
      <subscription-durability>NonDurable</subscription-durability>
    </message-driven-destination>
    <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
  </message-driven>
</enterprise-beans>
```

If the transaction type is "container," the transactional behavior of the MDB's methods are defined as for other enterprise beans in the deployment descriptor, as in the following example:

```
<assembly-descriptor>
```

```
    <container-transaction>
      <method>
        <ejb-name>Mdb</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

For the `onMessage` method, only the `Required` or `NotSupported` transaction attributes must be used, since there can be no pre-existing transaction context.

For the message selector specified in the previous example, the sent JMS messages are expected to have two properties, "Weight" and "LName." For example, to assign the JMS client program sending the messages:

```
message.setDoubleProperty("Weight",75.5);
message.setStringProperty("LName","Smith");
```

Such a message will be received by the Message-Driven Bean. The message selector syntax is based on a subset of the SQL92. Only messages whose headers and properties match the selector are delivered. Refer to the JMS specification for more details.

The JNDI name of a destination associated with an MDB is defined in the JOnAS-specific deployment descriptor, within a `jonas-message-driven` element, as illustrated in the following:

```
<jonas-message-driven>
  <ejb-name>Mdb</ejb-name>
  <jonas-message-driven-destination>
    <jndi-name>sampleTopic</jndi-name>
  </jonas-message-driven-destination>
</jonas-message-driven>
```

Once the destination is established, a client application can send messages to the MDB through a destination object obtained via JNDI as follows:

```
Queue q = context.lookup("sampleTopic");
```

If the client sending messages to the MDB is an EJB component itself, it is preferable that it use a resource environment reference to obtain the destination object. The use of resource environment references is described in Section 26.2 *Writing JMS Operations Within an Application Component*.

9.3. Administration Aspects

It is assumed at this point that the JOnAS server will make use of an existing JMS implementation; for example, JORAM or SwiftMQ.

The default policy is that the MDB developer and deployer are not concerned with JMS administration. This means that the developer/deployer will not create or use any JMS Connection factories and will not create a JMS destination (which is necessary for performing JMS operations within an EJB component (see Chapter 26 *JMS User's Guide*)); they will simply define the type of destination in the deployment descriptor and identify its JNDI name in the JOnAS-specific deployment descriptor, as described in the previous section. This means that JOnAS will implicitly create the necessary administered objects by using the proprietary administration APIs of the JMS implementation (since the administration APIs are not standardized). To perform such administration operations, JOnAS uses wrappers to the JMS provider administration API. For JORAM, the wrapper is `org.objectweb.jonas_jms.JmsAdminForJoram` (which is the default wrapper class defined by the `jonas.service.jms.mom` property in the `jonas.properties` file). For SwiftMQ, a

`com.swiftmq.appserver.jonas.JmsAdminForSwiftMQ` class can be obtained from the SwiftMQ site.

For the purpose of this implicit administration phase, the deployer must add the "jms" service in the list of the JOnAS services. For the example provided, the jonas.properties file should contain the following:

```
jonas.services                     registry,security,jtm,dbm,jms,ejb
                                   // The jms service must be added
jonas.service.ejb.descriptors     samplemdb.jar
jonas.service.jms.topics          sampleTopic        // not mandatory
```

The destination objects may or may not pre-exist. The EJB server will not create the corresponding JMS destination object if it already exists. (Refer also to Section 26.4 *JMS Administration*). The `sampleTopic` should be explicitly declared only if the JOnAS Server is going to create it first, even if the Message-Driven Bean is not loaded, or if it is use by another client before the Message-Driven Bean is loaded. In general, it is not necessary to declare the `sampleTopic`.

JOnAS uses a *pool of threads* for executing Message-Driven Bean instances on message reception, thus allowing large volumes of messages to be processed concurrently. As previously explained, MDB instances are stateless and several instances can execute concurrently on behalf of the same MDB. Minimum and maximum number of threads in the pool can be customized via the JOnAS properties `jonas.service.ejb.minworkthreads` and `jonas.service.ejb.maxworkthreads`, which are specified in the `jonas.properties` file as in the following example:

```
jonas.service.ejb.minworkthreads    5
jonas.service.ejb.maxworkthreads    80
```

9.4. Running a Message-Driven Bean

To deploy and run a Message-Driven Bean, perform the following steps:

- Verify that a registry is running.
- Start the Message-Oriented Middleware (the JMS provider implementation). See Section 9.4.1 *Launching the Message-Oriented Middleware* or Section 26.5.1 *Accessing the Message-Oriented Middleware as a Service*.
- Create and register in JNDI the JMS destination object that will be used by the MDB.

  This can be done automatically by the JMS service or explicitly by the proprietary administration facilities of the JMS provider (Section 26.4 *JMS Administration*). The JMS service creates the destination object if this destination is declared in the `jonas.properties` file (as specified in the previous section).

- Deploy the MDB component in JOnAS.

  Note that, if the destination object is not already created when deploying an MDB, the container asks the JMS service to create it based on the deployment descriptor content.

- Run the EJB client application.
- Stop the application.

  When using JMS, it is very important to stop JOnAS using the `jonas stop` command; it should not be stopped directly by killing it.

9.4.1. Launching the Message-Oriented Middleware

If the configuration property `jonas.services` contains the jms service, then the JOnAS JMS service will be launched and may try to launch a JMS implementation (a MOM).

For launching the MOM, three possibilities can be considered:

1. *Launching the MOM in the same JVM as JOnAS*

   This is the default situation obtained by assigning the `true` value to the configuration property `jonas.service.jms.collocated` in the `jonas.properties` file.
   ```
   jonas.services                security,jtm,dbm,jms,ejb
                                 // The jms service must be in the list
   jonas.service.jms.collocated  true
   ```
   In this case, the MOM is automatically launched by the JOnAS JMS service at the JOnAS launching time (command `jonas start`).

2. *Launching the MOM in a separate JVM*

   The JORAM MOM can be launched using the command:
   JmsServer

   For other MOMs, the proprietary command should be used.

   The configuration property `jonas.service.jms.collocated` must be set to `false` in the `jonas.properties` file. Setting this property is sufficient if the JORAM's JVM runs on the same host as JONAS, and if the MOM is launched with its default options (unchanged `a3servers.xml` configuration file under `JONAS_BASE/conf` or `JONAS_ROOT/conf` if `JONAS_BASE` is not defined).
   ```
   jonas.services                security,jtm,dbm,jms,ejb
                                 // The jms service must be in the list
   jonas.service.jms.collocated  false
   ```
   To use a specific configuration for the MOM, such as changing the default host (which is local-host) or the default connection port number (which is 16010), requires defining the additional `jonas.service.jms.url` configuration property as presented in the following case.

3. *Launching the MOM on another host*

   This requires defining the `jonas.service.jms.url` configuration property. When using JO-RAM, its value should be the JORAM URL `joram://`*host*`:`*port* where *host* is the host name, and *port* is the connection port (by default, 16010). For SwiftMQ, the value of the URL is similar to the following: `smqp://host:4001/timeout=10000`.
   ```
   jonas.services                security,jtm,dbm,jms,ejb
                                 // The jms service must be in the list
   jonas.service.jms.collocated  false
   jonas.service.jms.url         joram://host2:16010
   ```

9.4.1.1. Change the JORAM default configuration

As mentioned previously, the default host or default connection port number may need to be changed. This requires modifying the `a3servers.xml` configuration file provided by the JOnAS delivery in `JONAS_ROOT/conf` directory. For this, JOnAS must be configured with the property `jonas.service.jms.collocated` set to `false`, and the property `jonas.service.jms.url` set to `joram://host:port`. Additionally, the MOM must have been previously launched with the `JmsServer` command. This command defines a `Transaction` property set to `fr.dyade.aaa.util.NullTransaction`. If the messages need to be persistent, replace the `-DTransaction=fr.dyade.aaa.util.NullTransaction` option with the `-DTransaction=fr.dyade.aaa.util.ATransaction` option. Refer to the JORAM documentation for more details about this command. To define a more complex

configuration (for example, distribution, multi-servers), refer to the JORAM documentation on
http://joram.objectweb.org.

9.5. Transactional Aspects

Because a transactional context cannot be carried by a message (according to the EJB 2.1 speci-
fication), an MDB will never execute within an existing transaction. However, a transaction may
be started during the `onMessage` method execution, either due to a "required" transaction attribute
(container-managed transaction) or because it is explicitly started within the method (if the MDB is
bean-managed transacted). In the second case, the message receipt will not be part of the transac-
tion. In the case of the container-managed transaction, the container starts a new transaction before
de-queuing the JMS message (the receipt of which will, thus, be part of the started transaction), then
enlist the resource manager associated with the arriving message and all the resource managers ac-
cessed by the `onMessage` method. If the `onMessage` method invokes other Enterprise Beans, the
container passes the transaction context with the invocation. Therefore, the transaction started at the
`onMessage` method execution may involve several operations, such as accessing a database (via a call
to an Entity Bean, or by using a "datasource" resource), or sending messages (by using a "connection
factory" resource).

9.6. Message-Driven Beans Example

JOnAS provides examples that are located in the `examples/src/mdb` install directory.

`samplemdb` is a very simple example, the code of which is used in the previous topics for illustrating
how to use Message-Driven Beans.

`sampleappli` is a more complex example that shows how the sending of JMS messages and updates
in a database via JDBC may be involved in the same distributed transaction.

The following figure illustrates the architecture of this example application.
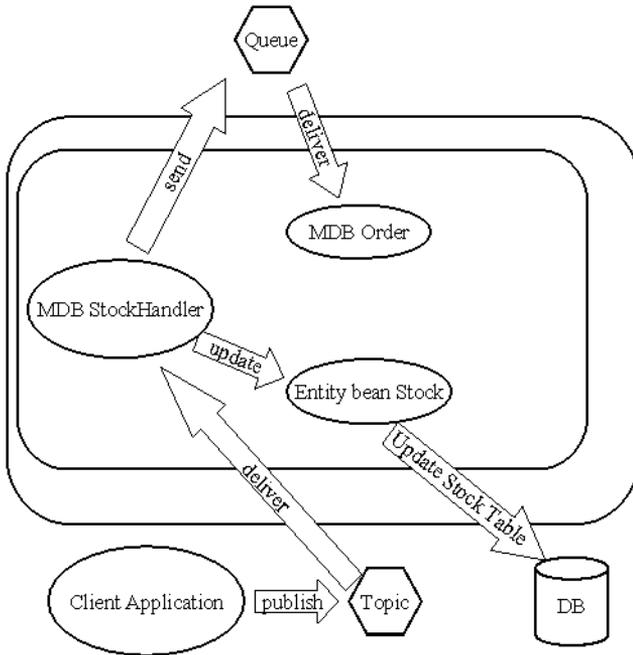
Figure 9-1. Example Architecture

There are two Message-Driven Beans in this example:

- `$JONAS_ROOT/examples/src/mdb/sampleappli/StockHandlerBean` is a Message-Driven Bean listening to a topic and receiving Map messages. The `onMessage` method runs in the scope of a transaction started by the container. It sends a Text message on a Queue (OrdersQueue) and updates a Stock element by decreasing the stock quantity. If the stock quantity becomes negative, an exception is received and the current transaction is marked for rollback.

- `$JONAS_ROOT/examples/src/mdb/sampleappli/OrderBean` is another Message-Driven Bean listening on the OrdersQueue Queue. On receipt of a Text message on this queue, it writes the corresponding String as a new line in a file (`Order.txt`).

The example also includes a CMP Entity Bean `$JONAS_ROOT/examples/src/mdb/sampleappli/Stock` that handles a stock table.

A Stock item is composed of a Stockid (String), which is the primary key, and a Quantity (int). The method `decreaseQuantity(int qty)` decreases the quantity for the corresponding stockid, but can throw a RemoteException `Negative stock`.

The JMS Client application **SampleAppliClient** sends several messages on `StockHandlerTopic` (see `$JONAS_ROOT/examples/src/mdb/sampleappli/SampleAppliClient.java`). It uses Map messages with three fields: `CustomerId`, `ProductId`, and `Quantity`. Before

sending messages, this client calls the EnvBean for creating the `StockTable` in the database with known values in order to check the results of updates at the end of the test (see `$JONAS_ROOT/examples/src/mdb/sampleappli/EnvBean.java`). Eleven messages are sent, the corresponding transactions are committed, and the last message sent causes the transaction to be rolled back.

9.6.1. Compiling This Example

To compile the sample application `examples/src/mdb/sampleappli`, use **Ant** with the `$JONAS_ROOT/examples/src/build.xml` file.

9.6.2. Running This Example

The default configuration of the JMS service in `jonas.properties` is:

```
jonas.services                 jmx,security,jtm,dbm,jms,ejb
                               // The jms service must be added
jonas.service.ejb.descriptors  sampleappli.jar
jonas.service.jms.topics       StockHandlerTopic
jonas.service.jms.queues       OrdersQueue
jonas.service.jms.collocated   true
```

This indicates that the JMS Server will be launched in the same JVM as the JOnAS Server, and the JMS-administered objects `StockHandlerTopic` (Topic) and `OrdersQueue` (Queue) will be created and registered in JNDI, if it does not already exist.

1. Run the JOnAS Server.
   service jonas start
2. Deploy the `sampleappli` container:
   jonas admin –a sampleappli.jar
3. Run the EJB client:
   jclient sampleappli.SampleAppliClient
4. Stop the server:
   service jonas stop

9.7. Tuning the Message-Driven Bean Pool

A pool is handled by JOnAS for each Message-Driven Bean. The pool can be configured in the JOnAS-specific deployment descriptor with the following tags:

min-pool-size

> This optional integer value represents the minimum instances that will be created in the pool when the bean is loaded. This will improve bean instance creation time, at least for the first beans. The default value is 0.

max-cache-size

> This optional integer value represents the maximum number of instances in memory. The purpose of this value is to keep JOnAS scalable. The policy is that, at bean-creation time, an instance is taken from the pool of free instances. If the pool is empty, a new instance is always created. When the instance must be released (at the end of the `onMessage` method), it is pushed into the

pool, except if the current number of instances created exceeds the max-cache-size, in which case this instance is dropped. The default value is no limit.

9.7.1. Message-Driven Bean Pool Example

```
<jonas-ejb-jar>
  <jonas-message-driven>
    <ejb-name>Mdb</ejb-name>
    <jndi-name>mdbTopic</jndi-name>
    <max-cache-size>20</max-cache-size>
    <min-pool-size>10</min-pool-size>
  </jonas-message-driven>
</jonas-ejb-jar>
```

Chapter 10.

Defining the Deployment Descriptor

This chapter is for the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side.

10.1. Principles

The bean programmer is responsible for providing the deployment descriptor associated with the developed Enterprise Beans. The Bean Provider's responsibilities and the Application Assembler's responsibilities are to provide an XML deployment descriptor that conforms to the deployment descriptor's XML schema as defined in the EBJ specification version 2.0. (Refer to `$JONAS_ROOT/xml/ejb-jar_2_1.xsd` or http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd).

To deploy Enterprise JavaBeans on the EJB server, information not defined in the standard XML deployment descriptor may be needed. For example, this information may include the mapping of the bean to the underlying database for an Entity Bean with container-managed persistence. This information is specified during the deployment step in another XML deployment descriptor that is specific to JOnAS. The JOnAS-specific deployment descriptor's XML schema is located in `$JONAS_ROOT/xml/jonas-ejb-jar_X_Y.xsd`. The file name of the JOnAS-specific XML deployment descriptor must be the file name of the standard XML deployment descriptor prefixed by "jonas-".

The parser gets the specified schema via the classpath (schemas are packaged in the `$JONAS_ROOT/lib/common/ow_jonas.jar` file).

The standard deployment descriptor should include the following structural information for each Enterprise Bean:

- The Enterprise Bean's name
- The Enterprise Bean's class
- The Enterprise Bean's home interface
- The Enterprise Bean's remote interface
- The Enterprise Bean's type
- A re-entrancy indication for the Entity Bean
- The Session Bean's state management type
- The Session Bean's transaction demarcation type
- The Entity Bean's persistence management
- The Entity Bean's primary key class
- Container-managed fields
- Environment entries
- The bean's EJB references
- Resource manager connection factory references
- Transaction attributes.

The JOnAS-specific deployment descriptor contains information for each Enterprise Bean including:

- The JNDI name of the Home object that implement the Home interface of the Enterprise Bean

- The JNDI name of the DataSource object corresponding to the resource manager connection factory referenced in the Enterprise Bean's class

- The JNDI name of each EJB references

- The JNDI name of JMS administered objects

• Information for the mapping of the bean to the underlying database, if it is an entity with container-managed persistence.

10.2. Example of Session Descriptors

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<ejb-jar 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/ejb-jar_2_1.xsd"
         version="2.1">
  <description>Here is the description of the test's beans</description>
  <enterprise-beans>
    <session>
      <description>... Bean example one ...</description>
      <display-name>Bean example one</display-name>
      <ejb-name>ExampleOne</ejb-name>
      <home>tests.Ex1Home</home>
      <remote>tests.Ex1</remote>
      <ejb-class>tests.Ex1Bean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>name1</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>value1</env-entry-value>
      </env-entry>
      <ejb-ref>
        <ejb-ref-name>ejb/ses1</ejb-ref-name>
        <ejb-ref-type>session</ejb-ref-type>
        <home>tests.SS1Home</home>
        <remote>tests.SS1</remote>
      </ejb-ref>
      <resource-ref>
        <res-ref-name>jdbc/mydb</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Application</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-inter>Home</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>methodOne</method-name>
```

```
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>methodTwo</method-name>
        <method-params>
          <method-param>int</method-param>
        </method-params>
      </method>
      <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>methodTwo</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>

<?xml version="1.0" encoding="ISO-8859-1"?>
<jonas-ejb-jar xmlns="http://www.objectweb.org/jonas/ns"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.objectweb.org/jonas/ns
   http://www.objectweb.org/jonas/ns/jonas-ejb-jar_4_0.xsd" >

  <jonas-session>
    <ejb-name>ExampleOne</ejb-name>
    <jndi-name>ExampleOneHome</jndi-name>
    <jonas-ejb-ref>
      <ejb-ref-name>ejb/ses1</ejb-ref-name>
      <jndi-name>SS1Home_one</jndi-name>
    </jonas-ejb-ref>
    <jonas-resource>
      <res-ref-name>jdbc/mydb</res-ref-name>
      <jndi-name>jdbc_1</jndi-name>
    </jonas-resource>
  </jonas-session>
</jonas-ejb-jar>
```

10.3. Example of Container-managed Persistence Entity Descriptors (CMP 1.1)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ejb-jar 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/ejb-jar_2_1.xsd"
     version="2.1">
  <description>Here is the description of the test's
  beans</description>
  <enterprise-beans>
    <entity>
```

```xml
      <description>... Bean example one ...</description>
      <display-name>Bean example two</display-name>
      <ejb-name>ExampleTwo</ejb-name>
      <home>tests.Ex2Home</home>
      <remote>tests.Ex2</remote>
      <local-home>tests.Ex2LocalHome</local-home>
      <local>tests.Ex2Local</local>
      <ejb-class>tests.Ex2Bean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>tests.Ex2PK</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>1.x</cmp-version>
      <cmp-field>
        <field-name>field1</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>field2</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>field3</field-name>
      </cmp-field>
      <primkey-field>field3</primkey-field>
      <env-entry>
        <env-entry-name>name1</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>value1</env-entry-value>
      </env-entry>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>ExampleTwo</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>

<?xml version="1.0" encoding="ISO-8859-1"?>
<jonas-ejb-jar xmlns="http://www.objectweb.org/jonas/ns"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.objectweb.org/jonas/ns
      http://www.objectweb.org/jonas/ns/jonas-ejb-jar_4_0.xsd" >
  <jonas-entity>
    <ejb-name>ExampleTwo</ejb-name>
    <jndi-name>ExampleTwoHome</jndi-name>
    <jndi-local-name>ExampleTwoLocalHome</jndi-local-name>
    <jdbc-mapping>
      <jndi-name>jdbc_1</jndi-name>
      <jdbc-table-name>YourTable</jdbc-table-name>
      <cmp-field-jdbc-mapping>
        <field-name>field1</field-name>
        <jdbc-field-name>dbf1</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <cmp-field-jdbc-mapping>
        <field-name>field2</field-name>
        <jdbc-field-name>dbf2</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <cmp-field-jdbc-mapping>
        <field-name>field3</field-name>
```

```
      <jdbc-field-name>dbf3</jdbc-field-name>
    </cmp-field-jdbc-mapping>
    <finder-method-jdbc-mapping>
      <jonas-method>
        <method-name>findByField1</method-name>
      </jonas-method>
      <jdbc-where-clause>where dbf1 = ?</jdbc-where-clause>
    </finder-method-jdbc-mapping>
  </jdbc-mapping>
 </jonas-entity>
</jonas-ejb-jar>
```

10.4. Tips

Although some characters, such as ">", are legal, it is good practice to replace them with XML entity references.

The following is a list of the predefined entity references for XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Chapter 11.

Transactional Behavior of EJB Applications

This chapter is for the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side.

11.1. Declarative Transaction Management

For container-managed transaction management, the transactional behavior of an Enterprise Bean is defined at configuration time and is part of the assembly-descriptor element of the standard deployment descriptor. It is possible to define a common behavior for all the methods of the bean, or to define the behavior at the method level. This is done by specifying a transactional attribute, which can be one of the following:

NotSupported

> If the method is called within a transaction, this transaction is suspended during the time of the method execution.

Required

> If the method is called within a transaction, the method is executed in the scope of this transaction; otherwise, a new transaction is started for the execution of the method and committed before the method result is sent to the caller.

RequiresNew

> The method will always be executed within the scope of a new transaction. The new transaction is started for the execution of the method, and committed before the method result is sent to the caller. If the method is called within a transaction, this transaction is suspended before the new one is started and resumed when the new transaction has completed.

Mandatory

> The method should always be called within the scope of a transaction, else the container will throw the `TransactionRequired` exception.

Supports

> The method is invoked within the caller transaction scope; if the caller does not have an associated transaction, the method is invoked without a transaction scope.

Never

> The client is required to call the bean without any transaction context; if it is not the case, a java.rmi.RemoteException is thrown by the container.

This is illustrated in the following table:

Transaction Attribute	Client transaction	Transaction associated with enterprise Bean's method
NotSupported	- T1	- -

Transaction Attribute	Client transaction	Transaction associated with enterprise Bean's method
Required	- T1	T2 T1
RequiresNew	- T1	T2 T2
Mandatory	- T1	error T1
Supports	- T1	- T1
Never	- T1	- error

In the deployment descriptor, the specification of the transactional attributes appears in the assembly-descriptor as follows:

```
<assembly-descriptor>
  <container-transaction>
    <method>
    <ejb-name>AccountImpl</ejb-name>
    <method-name>*</method-name>
    </method>
    <trans-attribute>Supports</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
    <ejb-name>AccountImpl</ejb-name>
    <method-name>getBalance</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
    <ejb-name>AccountImpl</ejb-name>
    <method-name>setBalance</method-name>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

In this example, for all methods of the AccountImpl bean which are not explicitly specified in a container-transaction element, the default transactional attribute is Supports (defined at the bean level), and the transactional attributes are Required and Mandatory (defined at the method-name level) for the methods getBalance and setBalance respectively.

11.2. Bean-managed Transactions

A bean that manages its transactions itself must set the `transaction-type` element in its standard deployment descriptor to:

```
<transaction-type>Bean</transaction-type>
```

To demarcate the transaction boundaries in a bean with bean-managed transactions, the bean programmer should use the `javax.transaction.UserTransaction` interface, which is defined on an EJB server object that may be obtained using the `EJBContext.getUserTransaction()` method (the SessionContext object or the EntityContext object depending on whether the method is defined on a session or on an Entity Bean).

The following example shows a Session Bean method `doTxJob` demarcating the transaction boundaries; the UserTransaction object is obtained from the sessionContext object, which should have been initialized in the setSessionContext method (refer to the Section 7.4 *The Enterprise Bean Class*).

```
public void doTxJob() throws  RemoteException {
    UserTransaction ut = sessionContext.getUserTransaction();
    ut.begin();
    ... // transactional operations
    ut.commit();
}
```

Another way to do this is to use JNDI and to retrieve UserTransaction with the name `java:comp/UserTransaction` in the initial context.

11.3. Distributed Transaction Management

As explained in the previous section, the transactional behavior of an application can be defined in a declarative way or coded in the bean and/or the client itself (transaction boundaries demarcation). In any case, the distribution aspects of the transactions are completely transparent to the bean provider and to the application assembler. This means that a transaction may involve beans located on several JOnAS servers and that the platform itself will handle management of the global transaction. It will perform the two-phase commit protocol between the different servers, and the bean programmer need do nothing.

Once the beans have been developed and the application has been assembled, it is possible for the deployer and for the administrator to configure the distribution of the different beans on one or several machines, and within one or several JOnAS servers. This can be done without impacting either the beans' code or their deployment descriptors. The distributed configuration is specified at launch time. In the environment properties of an EJB server, the following can be specified:

- Which Enterprise Beans the JOnAS server will handle
- If a Java Transaction Monitor will be located in the same Java Virtual Machine (JVM) or not.

To achieve this goal, two properties must be set in the `jonas.properties` file, `jonas.service.ejb.descriptors` and `jonas.service.jtm.remote`. The first one lists the beans that will be handled on this server (by specifying the name of their EJB-JAR files), and the second one sets the Java Transaction Monitor (JTM) launching mode:

- If set to `true`, the JTM is remote, that is, the JTM must be launched previously in another JVM
- If set to `false`, the JTM is local, that is, it will run in the same JVM as the EJB Server.

Example:

```
jonas.service.ejb.descriptors    Bean1.jar, Bean2.jar
```

```
jonas.service.jtm.remote          false
```

The Java Transaction Monitor can run outside any EJB server, in which case it can be launched in a stand-alone mode using the following command:

TMServer

Using these configuration facilities, it is possible to adapt the beans' distribution to the resources (CPU and data) location, for optimizing performance.

The following figure illustrates four cases of distribution configuration for three beans.
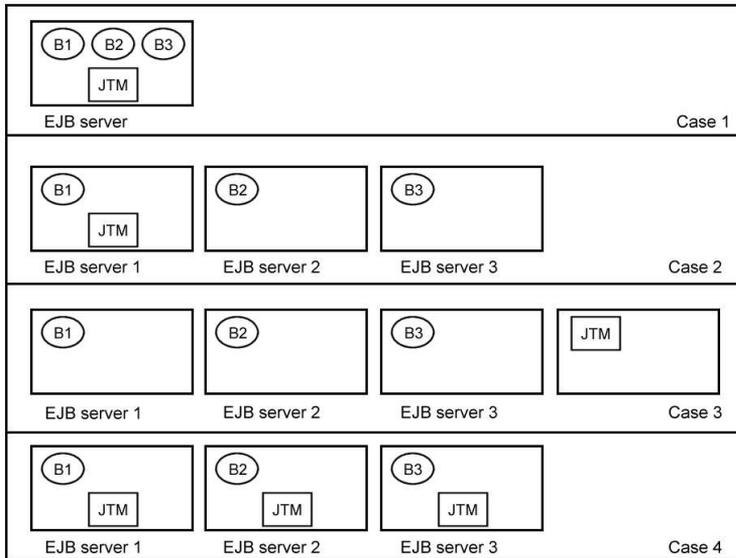
Figure 11-1. Distribution Configuration For Three Beans

1. Case 1: The three beans B1, B2, and B3 are located on the same JOnAS server, which embeds a Java Transaction Monitor.

2. Case 2: The three beans are located on different JOnAS servers, one of them running the Java Transaction Monitor, which manages the global transaction.

3. Case 3: The three beans are located on different JOnAS servers, the Java Transaction Monitor is running outside of any JOnAS server.

4. Case 4: The three beans are located on different JOnAS servers. Each EJB server is running a Java Transaction Monitor. One of the JTM acts as the master monitor, while the two others are slaves.

These different configuration cases may be obtained by launching the JOnAS servers and eventually the JTM (case 3) with the adequate properties. The rational when choosing one of these configurations is resources location and load balancing. However, consider the following points:

- If the beans should run on the same machine, with the same server configuration, case 1 is the more appropriate.
- If the beans should run on different machines, case 4 is the more appropriate, since it favors local transaction management.
- If the beans should run on the same machine, but require different server configurations, case 2 is a good approach.

Enterprise Bean Environment

This chapter is for the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side.

12.1. Introduction

The Enterprise Bean environment is a mechanism that allows customization of the Enterprise Bean's business logic during assembly or deployment. The environment is a way for a bean to refer to a value, to a resource, or to another component so that the code will be independent of the actual referred object. The actual value of such environment references (or variables) is set at deployment time, according to what is contained in the deployment descriptor. The Enterprise Bean's environment allows the Enterprise Bean to be customized without the need to access or change the Enterprise Bean's source code.

The Enterprise Bean environment is provided by the container (that is, the JOnAS server) to the bean through the JNDI interface as a JNDI context. The bean code accesses the environment using JNDI with names starting with `java:comp/env/`.

12.2. Environment Entries

The bean provider declares all the bean environment entries in the deployment descriptor via the `env-entry` element. The deployer can set or modify the values of the environment entries.

A bean accesses its environment entries with a code similar to the following:

```
InitialContext ictx = new InitialContext();
Context myenv = ictx.lookup("java:comp/env");
Integer min = (Integer) myenv.lookup("minvalue");
Integer max = (Integer) myenv.lookup("maxvalue");
```

In the standard deployment descriptor, the declaration of these variables are as follows:

```
<env-entry>
  <env-entry-name>minvalue</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>12</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>maxvalue</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>120</env-entry-value>
</env-entry>
```

12.3. Resource References

The resource references are another examples of environment entries. For such entries, using subcontexts is recommended:

- `java:comp/env/jdbc` for references to DataSources objects.
- `java:comp/env/jms` for references to JMS connection factories.

In the standard deployment descriptor, the declaration of a resource reference to a JDBC connection factory is:

```
<resource-ref>
  <res-ref-name>jdbc/AccountExplDs</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

And the bean accesses the datasource as in the following:

```
InitialContext ictx = new InitialContext();
DataSource ds = ictx.lookup("java:comp/env/jdbc/AccountExplDs");
```

Binding of the resource references to the actual resource manager connection factories that are configured in the EJB server is done in the JOnAS-specific deployment descriptor using the `jonas-resource` element.

```
<jonas-resource>
  <res-ref-name>jdbc/AccountExplDs</res-ref-name>
  <jndi-name>jdbc_1</jndi-name>
</jonas-resource>
```

12.4. Resource Environment References

The resource environment references are another example of environment entries. They allow the Bean Provider to refer to administered objects that are associated with resources (for example, JMS destinations), by using *logical* names. Resource environment references are defined in the standard deployment descriptor.

```
<resource-env-ref>
  <resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

Binding of the resource environment references to administered objects in the target operational environment is done in the JOnAS-specific deployment descriptor using the `jonas-resource-env` element.

```
<jonas-resource-env>
  <resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
  <jndi-name>myQueue<jndi-name>
</jonas-resource-env>
```

12.5. EJB References

The EJB reference is another special entry in the Enterprise Bean's environment. EJB references allow the Bean Provider to refer to the homes of other enterprise beans using *logical* names. For such entries, using the subcontext `java:comp/env/ejb` is recommended.

The declaration of an EJB reference used for accessing the bean through its *remote* home and component interfaces in the standard deployment descriptor is shown in the following example:

```
<ejb-ref>
  <ejb-ref-name>ejb/ses1</ejb-ref-name>
  <ejb-ref-type>session</ejb-ref-type>
```

```
  <home>tests.SS1Home</home>
  <remote>tests.SS1</remote>
</ejb-ref>
```

The declaration of an EJB reference used for accessing the bean through its *local* home and component interfaces in the standard deployment descriptor is shown in the following example:

```
<ejb-local-ref>
  <ejb-ref-name>ejb/locses1</ejb-ref-name>
  <ejb-ref-type>session</ejb-ref-type>
  <local-home>tests.LocalSS1Home</local-home>
  <local>tests.LocalSS1</local>
  <ejb-link>LocalBean</ejb-link>
</ejb-local-ref>
```

Local interfaces are available in the same JVM as the bean providing this interface. The use of these interfaces also implies that the classloader of the component performing a lookup (bean or servlet component) is a child of the EJB classloader providing the local interface. Local interfaces, then, are not available to outside WARs or outside EJB-JARs even if they run in the same JVM. This is due to the fact that classes of the local interfaces are not visible on the client side. Putting them under the WEB-INF/lib folder of a WAR would not change anything as the two classes would be loaded by different classloaders, which will throw a "ClassCastException".

To summarize, local interfaces are available only:

- for beans in the same EJB-JAR file
- from servlets to beans or from EJBs to EJBs that are in the same EAR file.

If the referred bean is defined in the same EJB-JAR or EAR file, the optional ejb-link element of the ejb-ref element can be used to specify the actual referred bean. The value of the ejb-link element is the name of the target Enterprise Bean, that is, the name defined in the ejb-name element of the target Enterprise Bean. If the target Enterprise Bean is in the same EAR file, but in a different EJB-JAR file, the name of the ejb-link element may be the name of the target bean, prefixed by the name of the containing EJB-JAR file followed by '#' (for example, My_EJBs.jar#bean1). Prefixing by the name of the EJB-JAR file is necessary only if some ejb-name conflicts occur; otherwise the name of the target bean is enough. In the following example, the ejb-link element has been added to the ejb-ref (in the referring bean SSA) and a part of the description of the target bean (SS1) is shown:

```
<session>
  <ejb-name>SSA</ejb-name>
  ...
  <ejb-ref>
    <ejb-ref-name>ejb/ses1</ejb-ref-name>
    <ejb-ref-type>session</ejb-ref-type>
    <home>tests.SS1Home</home>
    <remote>tests.SS1</remote>
    <ejb-link>SS1</ejb-link>
  </ejb-ref>
...
</session>
...
<session>
  <ejb-name>SS1</ejb-name>
  <home>tests.SS1Home</home>
  <local-home>tests.LocalSS1Home</local-home>
  <remote>tests.SS1</remote>
  <local>tests.LocalSS1</local>
  <ejb-class>tests.SS1Bean</ejb-class>
  ...
</session>
```

```
 ...
```

If the bean SS1 is not in the same EJB-JAR file as SSA, but in another file named `product_ejbs.jar`, the ejb-link element could be:

```
<ejb-link>product_ejbs.jar#SS1</ejb-link>
```

If the referring component and the referred bean are in separate files and not in the same EAR, the current JOnAS implementation does not allow use of the ejb-link element. To resolve the reference in this case, the `jonas-ejb-ref` element in the JOnAS-specific deployment descriptor would be used to bind the environment JNDI name of the EJB reference to the actual JNDI name of the associated Enterprise Bean home. In the following example, it is assumed that the JNDI name of the SS1 bean home is SS1Home_one.

```
<jonas-session>
  <ejb-name>SSA</ejb-name>
  <jndi-name>SSAHome</jndi-name>
  <jonas-ejb-ref>
    <ejb-ref-name>ejb/ses1</ejb-ref-name>
    <jndi-name>SS1Home_one</jndi-name>
  </jonas-ejb-ref>
</jonas-session>
...
<jonas-session>
  <ejb-name>SS1</ejb-name>
  <jndi-name>SS1Home_one</jndi-name>
  <jndi-local-name>SS1LocalHome_one</jndi-local-name>
</jonas-session>
...
```

The bean locates the home interface of the other Enterprise Bean using the EJB reference with the following code:

```
InitialContext ictx = new InitialContext();
Context myenv = ictx.lookup("java:comp/env");
SS1Home home =
  (SS1Home)javax.rmi.PortableRemoteObject.narrow(myEnv.lookup("ejb/ses1"),
  SS1Home.class);
```

Chapter 13.

Security Management

This chapter is for the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side.

13.1. Introduction

The EJB architecture encourages the Bean programmer to implement the Enterprise Bean class without hard-coding the security policies and mechanisms into the business methods.

13.2. Declarative Security Management

The application assembler can define a *security view* of the Enterprise Beans contained in the EJB-JAR file. The security view consists of a set of *security roles*. A security role is a semantic grouping of permissions for a given type of application user that allows that user to successfully use the application. The application assembler can define (declaratively in the deployment descriptor) *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods for the Enterprise Beans' home and remote interfaces. The security roles defined by the application assembler present this simplified security view of the Enterprise Beans application to the deployer; the deployer's view of security requirements for the application is the small set of security roles, rather than a large number of individual methods.

13.2.1. Security Roles

The application assembler can define one or more security roles in the deployment descriptor. The application assembler then assigns groups of methods of the Enterprise Beans' home and remote interfaces to the security roles in order to define the security view of the application.

The scope of the security roles defined in the `security-role` elements is the EJB-JAR file level, and this includes all the Enterprise Beans in the EJB-JAR file.

```
...
<assembly-descriptor>
   <security-role>
      <role-name>tomcat</role-name>
   </security-role>
   ...
</assembly-descriptor>
```

13.2.2. Method Permissions

After defining security roles for the Enterprise Beans in the EJB-JAR file, the application assembler can also specify the methods of the remote and home interfaces that each security role can invoke.

Method permissions are defined as a binary relationship in the deployment descriptor from the set of security roles to the set of methods of the home and remote interfaces of the Enterprise Beans, including all their super interfaces (including the methods of the `javax.ejb.EJBHome` and `javax.ejb.EJBObject` interfaces). The method permissions relationship includes the pair `(R, M)` only if the security role `R` is allowed to invoke the method `M`.

The application assembler defines the method permissions relationship in the deployment descriptor using the `method-permission` element as follows:

- Each `method-permission` element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the `role-name` element, and each method is identified by the `method` element.
- The method permissions relationship is defined as the union of all the method permissions defined in the individual `method-permission` elements.
- A security role or a method can appear in multiple `method-permission` elements.

It is possible that some methods are not assigned to any security roles. This means that these methods can be accessed by anyone.

The following example illustrates how security roles are assigned to methods' permissions in the deployment descriptor:

```
...
<method-permission>
   <role-name>tomcat</role-name>
   <method>
      <ejb-name>Op</ejb-name>
      <method-name>*</method-name>
   </method>
</method-permission>
...
```

13.3. Programmatic Security Management

Because not all security policies can be expressed declaratively, the EJB architecture also provides a simple programmatic interface that the Bean programmer can use to access the security context from the business methods.

The `javax.ejb.EJBContext` interface provides two methods that allow the bean programmer to access security information about the Enterprise Bean's caller.

```
public interface javax.ejb.EJBContext {
   ...
   //
   // The following two methods allow the EJB class
   // to access security information
   //
   java.security.Principal getCallerPrincipal() ;
   boolean isCallerInRole (String roleName) ;
   ...
}
```

13.3.1. Use of getCallerPrincipal()

The purpose of the `getCallerPrincipal()` method is to allow the Enterprise Bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to access information in a database.

An Enterprise Bean can invoke the `getCallerPrincipal()` method to obtain a `java.security.Principal` interface representing the current caller. The Enterprise Bean can

then obtain the distinguished name of the caller principal using the `getName()` method of the `java.security.Principal` interface.

13.3.2. Use of isCallerInRole(String roleName)

The main purpose of the `isCallerInRole(String roleName)` method is to allow the Bean programmer to code the security checks that cannot be easily defined declaratively in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The Enterprise Bean code uses the `isCallerInRole(String roleName)` method to test whether the current caller has been assigned to a given security role or not. Security roles are defined by the application assembler in the deployment descriptor and are assigned to principals by the deployer.

13.3.3. Declaration of Security Roles Referenced from the Bean's Code

The Bean programmer must declare in the `security-role-ref` elements of the deployment descriptor all the security role names used in the Enterprise Bean code. Declaring the security roles' references in the code allows the application assembler or deployer to link the names of the security roles used in the code to the actual security roles defined for an assembled application through the `security-role` elements.

```
...
<enterprise-beans>
    ...
    <session>
        <ejb-name>Op</ejb-name>
        <ejb-class>sb.OpBean</ejb-class>
        ...
        <security-role-ref>
            <role-name>role1</role-name>
        </security-role-ref>
        ...
    </session>
    ...
</enterprise-beans>
...
```

The deployment descriptor in this example indicates that the Enterprise Bean `Op` makes the security checks using `isCallerInRole("role1")` in at least one of its business methods.

13.3.4. Linking Security Role References and Security Roles

If the `security-role` elements have been defined in the deployment descriptor, all the security role references declared in the `security-role-ref` elements must be linked to the security roles defined in the `security-role` elements.

The following deployment descriptor example shows how to link the security role references named `role1` to the security role named `tomcat`.

```
...
<enterprise-beans>
    ...
    <session>
        <ejb-name>Op</ejb-name>
        <ejb-class>sb.OpBean</ejb-class>
```

```
        ...
        <security-role-ref>
            <role-name>role1</role-name>
            <role-link>tomcat</role-link>
        </security-role-ref>
        ...
    </session>
    ...
</enterprise-beans>
...
```

In summary, the role names used in the EJB code (in the isCallerInRole method) are, in fact, references to actual security roles, which makes the EJB code independent of the security configuration described in the deployment descriptor. The programmer makes these role references available to the Bean deployer or application assembler via the `security-role-ref` elements included in the `session` or `entity` elements of the deployment descriptor. Then, the Bean deployer or application assembler must map the security roles defined in the deployment descriptor to the "specific" roles of the target operational environment (for example, groups on Unix systems). However, this last mapping step is not currently available in JOnAS.

Chapter 14.
EJB Packaging

This chapter describes how the bean components should be packaged. It is for the Enterprise Bean provider; that is, the person in charge of developing the software components on the server side.

14.1. Enterprise Bean Principles

Enterprise Beans are packaged for deployment in a standard Java programming language Archive file, called an EJB-JAR file. This file must contain the following:

The beans' class files

> The class files of the remote and home interfaces, of the beans' implementations, of the beans' primary key classes (if there are any), and of all necessary classes.

The beans' deployment descriptor

> The EJB-JAR file must contain the deployment descriptors, which are made up of:

> - The standard xml deployment descriptor, in the format defined in the EJB 2.1 specification. Refer to `$JONAS_ROOT/xml/ejb-jar_2_1.xsd` or http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd. This deployment descriptor must be stored with the name `META-INF/ejb-jar.xml` in the EJB-JAR file.

> - The JOnAS-specific XML deployment descriptor in the format defined in `$JONAS_ROOT/xml/jonas-ejb-jar_X_Y.xsd`. This JOnAS deployment descriptor must be stored with the name `META-INF/jonas-ejb-jar.xml` in the EJB-JAR file.

14.1.1. Entity Bean Example

Before building the EJB-JAR file of the Account Entity Bean example, the Java source files must be compiled to obtain the class files and the two XML deployment descriptors must be written.

Then, the EJB-JAR file (`OpEB.jar`) can be built using the `jar` command:

```
cd your_bean_class_directory
mkdir META-INF
cp .../eb/*.xml META-INF
jar cvf OpEB.jar sb/*.class META-INF/*.xml
```

Chapter 15.

Application Deployment and Installation Guide

This chapter is for the application deployer.

15.1. Deployment and Installation Process Principles

15.1.1. The Deployment and Installation of Enterprise Beans

This chapter assumes that the Enterprise Bean provider followed the *Enterprise Beans Programmer's Guide* and packaged the beans' classes together with the deployment descriptors in a EJB-JAR file. To deploy un-packed Enterprise Beans, refer to Section 3.5.2 *Configuring the EJB Container Service*.

To deploy the Enterprise Beans in JOnAS, the deployer must add the interposition classes interfacing the EJB components with the services provided by the JOnAS application server.

The Chapter 6 *JOnAS Command Reference* tool supplied in the JOnAS distribution provides the capability of generating interposition classes and updating the EJB-JAR file.

The application deployer may also need to customize the deployment descriptors in order to adapt it to a specific operational environment. This must be done before using GenIC.

The deployer may choose to deploy the Enterprise Beans as stand-alone application components, in which case the EJB-JAR must be installed in the `$JONAS_ROOT/ejbjars` directory. The deployer may also choose to include them in WAR or EAR packaging, which is presented in the following sections.

15.1.2. The Deployment and Installation of Web and J2EE Applications

Once the packaging of the application components has been completed as described in the Chapter 18 *WAR Packaging* or Chapter 23 *EAR Packaging* guides, the obtained archive file must be installed in the:

- `$JONAS_ROOT/webapps` directory, for WAR files
- `$JONAS_ROOT/apps` directory, for EAR files

15.2. Example of Deploying and Installing an EJB Using an EJB-JAR File

For this example, it is assumed that you want to customize the deployment of the `AccountImpl` bean in the JOnAS example `examples/src/eb` by changing the name of the database table used for the persistence of the `AccountImpl`.

The current directory is `$JONAS_ROOT/examples/src/eb`. Do the following:

- Edit `jonas-ejb-jar.xml` and modify the value of the `<jdbc-table-name>` element included in the `<jdbc-mapping>` element corresponding to `AccountImpl` entity.
- Compile all the `.java` files present in this directory:
  ```
  javac -d ../../classes Account.java AccountImplBean.java
   AccountExplBean.java AccountHome.java ClientAccount.java
  ```

- Perform the deployment:

  - Build an EJB-JAR file named `ejb-jar.jar` with all the corresponding classes and the two deployment descriptors:
    ```
    mkdir -p ../../classes/META-INF
    cp  ejb-jar.xml ../../classes/META-INF/ejb-jar.xml
    cp jonas-ejb-jar.xml ../../classes/META-INF/jonas-ejb-jar.xml
    cd ../../classes
    jar cvf eb/ejb-jar.jar META-INF/ejb-jar.xml
       META-INF/jonas-ejb-jar.xml eb/Account.class eb/AccountExplBean.class
       eb/AccountHome.class eb/AccountImplBean.class
    ```
  - From the source directory, run the `GenIC` generation tool that will generate the final `ejb-jar.jar` file with the interposition classes:
    ```
    GenIC -d ../../classes ejb-jar.jar
    ```

- Install the EJB-JAR in the `$JONAS_ROOT/ejbjars` directory:
  ```
  cp ../../classes/eb/ejb-jar.jar $JONAS_ROOT/ejbjars/ejb-jar.jar
  ```

The JOnAS application Server can now be launched using the command:

```
service jonas start
```

The steps just described for building the new `ejb-jar.jar` file explain the deployment process. It is generally implemented by an ANT build script.

If `Apache ANT` is installed on your machine, type `ant install` in the `$JONAS_ROOT/examples/src` directory to build and install all `ejb-jar.jar` files for the examples.

To write a `build.xml` file for ANT, use the `ejbjar` task, which is one of the optional EJB tasks defined in ANT (see http://jakarta.apache.org/ant/manual/index.html). The `ejbjar` task contains a nested element called `jonas`, which implements the deployment process described above (interposition classes generation and EJB-JAR file update).

Generally, the `$JONAS_ROOT/lib/common/ow_jonas_ant.jar` file has the most up-to-date version of the EJB task containing an updated implementation of the `jonas` nested element. See Chapter 27 *Ant EJB Tasks: Using EJB-JAR* for information on the jonas nested element.

For example, `$JONAS_ROOT/examples/src/alarm/build.xml` contains this code snippet:

```
<!-- ejbjar task  -->
<taskdef name="ejbjar"
   classname="org.objectweb.jonas.ant.EjbJar"
   classpath="${jonas.root}/lib/common/ow_jonas_ant.jar" />

<!-- Deploying ejbjars via ejbjar task  -->
<target name="jonasejbjar"
     description="Build and deploy the ejb-jar file"
     depends="compile" >
  <ejbjar basejarname="alarm"
       srcdir="${classes.dir}"
       descriptordir="${src.dir}/beans/org/objectweb/alarm/beans"
       dependency="full">
    <include name="**/alarm.xml"/>
    <support dir="${classes.dir}">
       <include name="**/ViewProxy.class"/>
    </support>
    <jonas destdir="${dist.ejbjars.dir}"
       jonasroot="${jonas.root}"
       protocols="${protocols.names}" />
```

```
    </ejbjar>
</target>
```

15.3. Deploying and Installing a Web Application

Before deploying a Web application in the JOnAS application server, first package its components in a WAR file as explained in Chapter 18 *WAR Packaging*.

For Apache ANT, refer to the target WAR in the `$JONAS_ROOT/examples/earsample/build.xml` file.

Next, install the WAR file into the `$JONAS_ROOT/webapps` directory.

✎ **Note**

> Be aware that the WAR file must not be installed in the `$CATALINA_HOME/webapps` directory.

Then, check the configuration. Before running the web application, check that the `web` service is present in the `jonas.services` property. The `ejb` service may also be needed if the Web application uses enterprise beans.

The name of the WAR file can be added in the `jonas.service.web.descriptors` section.

Finally, `run` the application Server:

jonas start

The web components are deployed in a web container created during the startup. If the WAR file was not added in the `jonas.service.web.descriptors` list, the web components can be dynamically deployed using the `jonas admin` command or `JonasAdmin` tool.

15.4. Deploying and Installing a J2EE Application

Before deploying a J2EE application in the JOnAS application server, first package its components in an EAR file as explained in Chapter 23 *EAR Packaging*.

For Apache ANT, refer to the target EAR in the `$JONAS_ROOT/examples/earsample/build.xml` file.

Next, install the EAR file into the `$JONAS_ROOT/apps` directory.

Then, check the configuration. Before running the application, check that the `ejb`, `web` and `ear` services are present in the `jonas.services` property.

The name of the EAR file can be added in the `jonas.service.ear.descriptors` section.

Finally, `run` the application Server:

```
jonas start
```

The application components are deployed in EJB and web containers created during the startup. If the EAR file was not added in the `jonas.service.ear.descriptors` list, the application components can be dynamically deployed using the `jonas admin` command or `JonasAdmin` tool.

III. Web Application Programmer's Guide

This section contains information for the Web Application programmer; that is, the person in charge of developing the web components on the server side.

The Chapter 16 *Developing Web Components* guide explains how to construct Web components, as well as how to access Enterprise Beans from within the Web Components.

Deployment descriptor specification is presented in Chapter 17 *Defining the Web Deployment Descriptor*.

Web components can be used as *Web application* components or as *J2EE application* components. In both cases, a WAR file will be created, but the content of this file is different in the two situations. In the first case, the WAR contains the Web components and the Enterprise Beans. In the second case, the WAR does not contain the Enterprise Beans. The EJB JAR file containing the Enterprise Beans is packed together with the WAR file containing the Web components, into an EAR file.

Principles and tools for providing WAR files are presented in Chapter 18 *WAR Packaging* and Chapter 15 *Application Deployment and Installation Guide*.

Table of Contents

Chapter 16.

Developing Web Components

This chapter is for the Web Component provider; that is, the person in charge of developing the web components on the server side.

16.1. Introduction to Web Component Development

A Web Component is a generic term that denotes both JSP pages and servlets. Web components are packaged in a `.war` file and can be deployed in a JOnAS server via the `web` service. Web components can be integrated in a J2EE application by packing the `.war` file in an `.ear` file (refer to Chapter 22 *Defining the EAR Deployment Descriptor*).

The JOnAS distribution includes a Web application example called The EarSample example (see http://www.objectweb.org/jonas/current/examples/earsample).

The directory structure of this application is as follows:

`etc/xml`	Contains the `web.xml` file that describes the web application.
`etc/resources/web`	Contains HTML pages and images; JSP pages can also be placed here.
`src/org/objectweb/earsample/servlets`	Servlet sources
`src/org/objectweb/earsample/beans`	Beans sources

If beans from another application will be used, the `bean` directory is not needed.

16.2. The JSP Pages

Java Server Pages (JSP) is a technology that allows regular, static HTML, to be mixed with dynamically-generated HTML written in Java programming language for encapsulating the logic that generates the content for the page. Refer to the Java Server Pages (http://java.sun.com/products/jsp/) and the *Quickstart Guide* (http://java.sun.com/products/jsp/docs.html) for more details.

16.2.1. Example

The following example shows a sample JSP page that lists the content of a cart.

```
<!-- Get the session -->
<%@ page session="true" %>

<!-- The import to use -->
<%@ page import="java.util.Enumeration" %>
<%@ page import="java.util.Vector"      %>

<html>
<body bgcolor="white">
  <h1>Content of your cart</h1><br>
  <table>
    <!-- The header of the table -->
    <tr bgcolor="black">
```

```
      <td><font color="lightgreen">Product Reference</font></td>
      <td><font color="lightgreen">Product Name</font></td>
      <td><font color="lightgreen">Product Price</font></td>
    </tr>

    <!-- Each iteration of the loop display a line of the table -->
    <%
      Cart cart = (Cart) session.getAttribute("cart");
      Vector products = cart.getProducts();
      Enumeration enum = products.elements();
      // loop through the enumeration
      while (enum.hasMoreElements()) {
          Product prod = (Product) enum.nextElement();
    %>
    <tr>
      <td><%=prod.getReference()%></td>
      <td><%=prod.getName()%></td>
      <td><%=prod.getPrice()%></td>
    </tr>
    <%
    } // end loop
    %>
  </table>
</body>
</html>
```

It is a good idea to hide all the mechanisms for accessing EJBs from JSP pages by using a proxy Java bean, referenced in the JSP page by the usebean special tag. This technique is shown in the alarm example http://www.objectweb.org/jonas/current/examples/alarm/web/secured, where the .jsp files communicate with the EJB via a proxy Java bean ViewProxy.java http://www.objectweb.org/jonas/current/examples/alarm/beans/org/objectweb/alarm/beans/ViewProxy.java.

16.3. The Servlets

Servlets are modules of Java code that run in an application server for answering client requests. Servlets are not tied to a specific client-server protocol. However, they are most commonly used with HTTP, and the word "servlet" is often used as referring to an "HTTP servlet."

Servlets make use of the Java standard extension classes in the packages javax.servlet (the basic servlet framework) and javax.servlet.http (extensions of the servlet framework for servlets that answer HTTP requests).

Typical uses for HTTP servlets include:

• processing and/or storing data submitted by an HTML form,
• providing dynamic content generated by processing a database query,
• managing information of the HTTP request.

For more details refer to the Java Servlet Technology pages (http://java.sun.com/products/servlet/).

16.3.1. Example

The following example is a sample of a servlet that lists the content of a cart. This example is the servlet version of the previous JSP page example.

```java
import java.util.Enumeration;
import java.util.Vector;
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class GetCartServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

        res.setContentType("text/html");
        PrintWriter out = res.getWriter();

        out.println("<html><head><title>Your
            cart</title></head>");
        out.println("<body>");
        out.println("<h1>Content of your cart</h1><br>");
        out.println("<table>");

        // The header of the table
        out.println("<tr>");
        out.println("<td><font
            color="lightgreen">Product Reference</font></td>");
        out.println("<td><font
            color="lightgreen">Product Name</font></td>");
        out.println("<td><font
            color="lightgreen">Product Price</font></td>");
        out.println("</tr>");

        // Each iteration of the loop display a line of the table
        HttpSession session = req.getSession(true);
        Cart cart = (Cart) session.getAttribute("cart");
        Vector products = cart.getProducts();
        Enumeration enum = products.elements();
        while (enum.hasMoreElements()) {
            Product prod = (Product) enum.nextElement();
            int prodId = prod.getReference();
            String prodName = prod.getName();
            float prodPrice = prod.getPrice();
            out.println("<tr>");
            out.println("<td>" + prodId + </td>);
            out.println("<td>" + prodName + </td>);
            out.println("<td>" + prodPrice + </td>);
            out.println("</tr>");
            }

            out.println("</table>");
            out.println("</body>");
            out.println("</html>");
            out.close();
        }
}
```

16.4. Accessing an EJB from a Servlet or JSP Page

Starting with JOnAS 2.6 with its `web` container service, it is possible to access an enterprise Java bean and its environment in a J2EE-compliant way.

The following sections describe:

1. How to access the Remote Home interface of a bean.

2. How to access the Local Home interface of a bean.

3. How to access the environment of a bean.

4. How to start transactions in servlets.

Note

All the following code examples are taken from the The EarSample example provided in the JOnAS distribution.

16.4.1. Accessing the Remote Home Interface of a Bean:

In this example the servlet gets the Remote Home interface `OpHome` registered in JNDI using an EJB reference, then creates a new instance of the Session Bean:

```
import javax.naming.Context;
import javax.naming.InitialContext;

//remote interface
import org.objectweb.earsample.beans.secusb.Op;
import org.objectweb.earsample.beans.secusb.OpHome;

  Context initialContext = null;
  try {
    initialContext = new InitialContext();
  } catch (Exception e) {
    out.print("<li>Cannot get initial context for JNDI: ");
    out.println(e + "</li>");
    return;
  }
// Connecting to OpHome through JNDI
  OpHome opHome = null;
  try {
    opHome = (OpHome)
    PortableRemoteObject.narrow(initialContext.lookup \
      ("java:comp/env/ejb/Op"), OpHome.class);
  } catch (Exception e) {
    out.println("<li>Cannot lookup java:comp/env/ejb/Op: "
      + e + "</li>");
    return;
  }
  // OpBean creation
  Op op = null;
  try {
    op = opHome.create("User1");
  } catch (Exception e) {
    out.println("<li>Cannot create OpBean: " + e + "</li>");
    return;
  }
```

Note that the following elements must be set in the `web.xml` file tied to this web application:

```
<ejb-ref>
  <ejb-ref-name>ejb/Op</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>org.objectweb.earsample.beans.secusb.OpHome</home>
  <remote>org.objectweb.earsample.beans.secusb.Op</remote>
  <ejb-link>secusb.jar#Op</ejb-link>
</ejb-ref>
```

16.4.2. Accessing the Local Home of a Bean:

The following example shows how to obtain a local home interface `OpLocalHome` using an EJB local reference:

```
//local interfaces
import org.objectweb.earsample.beans.secusb.OpLocal;
import org.objectweb.earsample.beans.secusb.OpLocalHome;

  // Connecting to OpLocalHome thru JNDI
    OpLocalHome opLocalHome = null;
    try {
      opLocalHome = (OpLocalHome)
          initialContext.lookup("java:comp/env/ejb/OpLocal");
    } catch (Exception e) {
      out.println("<li>Cannot lookup java:comp/env/ejb/OpLocal: "
          + e + "</li>");
      return;
    }
```

This is found in the `web.xml` file:

```
<ejb-local-ref>
  <ejb-ref-name>ejb/OpLocal</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>org.objectweb.earsample.beans.secusb.OpLocalHome
    </local-home>
  <local>org.objectweb.earsample.beans.secusb.OpLocal</local>
  <ejb-link>secusb.jar#Op</ejb-link>
</ejb-local-ref>
```

16.4.3. Accessing the Environment of the Component

In this example, the servlet seeks to access the component's environment:

```
String envEntry = null;
  try {
    envEntry = (String)
      initialContext.lookup("java:comp/env/envEntryString");
  } catch (Exception e) {
    out.println("<li>Cannot get env-entry on JNDI " + e + "</li>");
    return;
  }
```

This is the corresponding part of the `web.xml` file:

```
<env-entry>
```

```
  <env-entry-name>envEntryString</env-entry-name>
  <env-entry-value>This is a string from env-entry</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

16.4.4. Starting Transactions in Servlets

The servlet wants to start transactions via the `UserTransaction`:

```
import javax.transaction.UserTransaction;

    // We want to start transactions from client: get UserTransaction
    UserTransaction utx = null;
    try {
        utx = (UserTransaction)
        initialContext.lookup("java:comp/UserTransaction");
    } catch (Exception e) {
        out.println("<li>Cannot lookup java:comp/UserTransaction: "
        + e + "</li>");
        return;
    }

    try {
        utx.begin();
        opLocal.buy(10);
        opLocal.buy(20);
        utx.commit();

    } catch (Exception e) {
        out.println("<li>exception during 1st Tx: " + e + "</li>");
        return;
    }
```

Defining the Web Deployment Descriptor

This chapter is for the Web component provider; that is, the person in charge of developing the Web components on the server side.

17.1. Principles

The Web component programmer is responsible for providing the deployment descriptor associated with the developed web components. The Web component provider's responsibilities and the application assembler's responsibilities are to provide an XML deployment descriptor that conforms to the deployment descriptor's XML schema as defined in the Java Servlet Specification Version 2.4. (Refer to $JONAS_ROOT/xml/web-app_2_4.xsd or http://jonas.objectweb.org/current/xml/web-app_2_4.xsd).

To customize the Web components, information not defined in the standard XML deployment descriptor may be needed. For example, the information may include the mapping of the name of referenced resources to its JNDI name. This information can be specified during the deployment phase, within another XML deployment descriptor that is specific to JOnAS. The JOnAS-specific deployment descriptor's XML schema is located in $JONAS_ROOT/xml/jonas-web-app_X_Y.xsd. The file name of the JOnAS-specific XML deployment descriptor must be the file name of the standard XML deployment descriptor prefixed by "jonas-".

The parser gets the specified schema via the classpath (schemas are packaged in the $JONAS_ROOT/lib/common/ow_jonas.jar file).

The standard deployment descriptor (web.xml) should contain structural information that includes the following:

- The servlet's description (including servlet's name, servlet's class or jsp-file, servlet's initialization parameters)
- Environment entries
- EJB references
- EJB local references
- Resource references
- Resource env references.

The JOnAS-specific deployment descriptor (jonas-web.xml) may contain information that includes:

- The JNDI name of the external resources referenced by a Web component

- The JNDI name of the external resources environment referenced by a Web component

- The JNDI name of the referenced beans by a Web component

- The name of the virtual host on which to deploy the servlets

- The name of the context root on which to deploy the servlets

- The compliance of the web application classloader to the Java 2 delegation model or not.

<host> element: If the configuration file of the web container contains virtual hosts, the host on which the WAR file is deployed can be set.

<context-root> element: You should specify the name of the context on which the application will be deployed. If it is not specified, the context-root used can be one of the following:

- If the WAR is packaged into an EAR file, the context-root used is the context specified in the `application.xml` file.
- If the WAR is standalone, the context-root is the name of the WAR file (that is, the context-root is `jonasAdmin` for `jonasAdmin.war`).

If the context-root is / or empty, the web application is deployed as ROOT context (that is, http://localhost:9000/).

`<java2-delegation-model>` element: Set the compliance to the Java 2 delegation model.

- If true: the web application context uses a classloader, using the Java 2 delegation model (ask parent classloader first).
- If false: the class loader searches inside the web application first, before asking parent class loaders.

17.2. Examples of Web Deployment Descriptors

- Example of a standard Web Deployment Descriptor (`web.xml`):

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>

<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>Op</servlet-name>
     <servlet-class>
       org.objectweb.earsample.servlets.ServletOp
     </servlet-class>
  </servlet>

  <servlet-mapping>
   <servlet-name>Op</servlet-name>
   <url-pattern>/secured/Op</url-pattern>
  </servlet-mapping>

  <security-constraint>
   <web-resource-collection>
     <web-resource-name>Protected Area</web-resource-name>
       <!-- Define the context-relative URL(s) to be protected -->
       <url-pattern>/secured/*</url-pattern>
       <!-- If you list http methods, only those methods
         are protected -->
       <http-method>DELETE</http-method>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
       <http-method>PUT</http-method>
   </web-resource-collection>
   <auth-constraint>
     <!-- Anyone with one of the listed roles
       may access this area -->
     <role-name>tomcat</role-name>
     <role-name>role1</role-name>
   </auth-constraint>
  </security-constraint>

  <!-- Default login configuration uses BASIC authentication -->
```

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Example Basic Authentication Area</realm-name>
</login-config>

<env-entry>
  <env-entry-name>envEntryString</env-entry-name>
  <env-entry-value>
    This is a string from the env-entry
  </env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<!-- reference on a remote bean without ejb-link-->
<ejb-ref>
  <ejb-ref-name>ejb/Op</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>org.objectweb.earsample.beans.secusb.OpHome</home>
  <remote>org.objectweb.earsample.beans.secusb.Op</remote>
</ejb-ref>

<!-- reference on a remote bean using ejb-link-->
<ejb-ref>
  <ejb-ref-name>ejb/EjbLinkOp</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>org.objectweb.earsample.beans.secusb.OpHome</home>
  <remote>org.objectweb.earsample.beans.secusb.Op</remote>
  <ejb-link>secusb.jar#Op</ejb-link>
</ejb-ref>

<!-- reference on a local bean -->
<ejb-local-ref>
  <ejb-ref-name>ejb/OpLocal</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home
    >org.objectweb.earsample.beans.secusb.OpLocalHome
  </local-home>
  <local>org.objectweb.earsample.beans.secusb.OpLocal</local>
  <ejb-link>secusb.jar#Op</ejb-link>
</ejb-local-ref>
</web-app>
```

- Example of a specific Web Deployment Descriptor (`jonas-web.xml`):

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<jonas-web-app xmlns="http://www.objectweb.org/jonas/ns"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.objectweb.org/jonas/ns
      http://www.objectweb.org/jonas/ns/jonas-web-app_4_0.xsd" >

  <!-- Mapping between the referenced bean and its JNDI name,
          override the ejb-link, if there is one in the associated
          ejb-ref in the standard Web Deployment Descriptor -->
  <jonas-ejb-ref>
    <ejb-ref-name>ejb/Op</ejb-ref-name>
    <jndi-name>OpHome</jndi-name>
  </jonas-ejb-ref>

  <!-- the virtual host on which deploy the web application -->
  <host>localhost</host>

  <!-- the context root on which deploy the web application -->
```

```
    <context-root>web-application</context-root>
</jonas-web-app>
```

17.3. Tips

Although some characters, such as ">", are legal, it is good practice to replace them with XML entity references. The following is a list of the predefined entity references for XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Chapter 18.
WAR Packaging

This chapter is for the Web component provider; that is, the person in charge of developing the web components on the server side. It describes how the web components should be packaged.

18.1. Principles

Web components are packaged for deployment in a standard Java programming language Archive file called a WAR file (Web ARchive), which is a JAR similar to the package used for Java class libraries. A WAR has a specific hierarchical directory structure. The top-level directory of a WAR is the document root of the application.

The document root is where JSP pages, client-side classes and archives, and static web resources are stored. The document root contains a subdirectory called `WEB-INF`, which contains the following files and directories:

- `web.xml`: The standard xml deployment descriptor in the format defined in the Java Servlet 2.4 Specification. Refer to `$JONAS_ROOT/xml/web-app_2_4.xsd`.
- `jonas-web.xml`: The optional JOnAS-specific XML deployment descriptor in the format defined in `$JONAS_ROOT/xml/jonas-web_X_Y.xsd`.
- `classes`: a directory that contains the servlet classes and utility classes.
- `lib`: a directory that contains JAR archives of libraries (tag libraries and any utility libraries called by server-side classes). If the Web application uses Enterprise Beans, it can also contain `ejb-jars`. This is necessary to give to the Web components the visibility of the EJB classes. However, if the `war` is intended to be packed in an `EAR`, the `ejb-jars` must not be placed here. In this case, they are directly included in the `EAR`. Due to the use of the class loader hierarchy, Web components have the visibility of the EJB classes. Details about the class loader hierarchy are described in Chapter 5 *JOnAS Class Loader Hierarchy*.

18.1.1. Example

Before building a WAR file, the Java source files must be compiled to obtain the class files (located in the `WEB-INF/classes` directory) and the two XML deployment descriptors must be written.

Then, the WAR file (`web-application.war`) is built using the `jar` command:

```
cd your_webapp_directory
jar cvf web-application.war *
```

During the development process, an "unpacked version" of the WAR file can be used. Refer to Section 3.5.4 *Configuring the WEB Container Service*.

IV. J2EE Client Application Programmer's Guide

This section contains information for the J2EE Client programmer; that is, the person in charge of developing the client components on the client side.

Table of Contents

Launching J2EE Client Applications

This chapter is for the Client Component provider; that is, the person in charge of developing the client components on the client side.

19.1. Launching Clients

The J2EE client application can be:

- A standalone client in a `.jar` file
- A class name, which must be found in the CLASSPATH
- A client bundle in an `.ear` file. An `.ear` file can contain many Java clients.

All the files required to launch the client container are in the `JONAS_ROOT/lib/client.jar` file. This `jar` includes a manifest file with the name of the class to launch. To launch the client container, simply type:

```
java –jar $JONAS_ROOT/lib/client.jar –?
```

This launches the client container and display usage information about this client container.

To launch the client container on a remote computer, copy the `client.jar` and invoke the client container by typing:

```
java –jar path_to_your/client.jar
```

The client that must be launched by the client container is given as an argument of the client container. For example:

```
java –jar client.jar myApplication.ear
```

or

```
java –jar client.jar myClient.jar
```

19.2. Configuring the Client Container

19.2.1. JNDI Access

Defining the JNDI access and the protocol to use is an important part of configuration. The JOnAS server, as well as the ClientContainer, uses the values specified in the `carol.properties` file. This file can be used at different levels. The `carol.properties` is searched with the following priority (high to low):

- The `carol.properties` specified by the `–carolFile` argument to the client container.
- The `carol.properties` packaged into the client application (the JAR client).
- If not located previously, it will use the `carol.properties` contained in the `JONAS_ROOT/lib/client.jar`.

A convenient way is to update the `carol.properties` of your `client.jar` with your customized `carol.properties` file. That is, `jar -uf client.jar carol.properties`

19.2.2. Trace Configuration

The client container `client.jar` includes a `traceclient.properties` file. This is the same file as the one in `JONAS_ROOT/conf` directory.

A different configuration file can be used for the traces by specifying the parameter `-traceFile` when invoking the client container.

The file in the `client.jar` can also be replaced with the command: `jar -uf client.jar traceclient.properties`

19.2.3. Classpath Configuration

Some jars/classes can be added to the client container. For example, if a class requires some extra libraries/classes, you can use the `-cp path/to/classes` option.

The classloader of the client container will use the libraries/classes provided by the `-cp` flag.

19.2.4. Specifying the Client to Use (EAR Case)

An EAR can contain many Java clients, which are described in the `application.xml` file inside the `<module><java>` elements.

To invoke the client container with an ear, such as `java -jar client.jar my.ear`, specify the Java client to use if there are many clients. Otherwise, it will take the first client.

To specify the JAR client to use from an EAR, use the argument `-jarClient` and supply the name of the client to use.

The `earsample` example in the JOnAS examples has two Java clients in its `ear` file.

19.2.5. Specifying the Directory for Unpacking the EAR (EAR Case)

By default, the client container will use the system property `java.io.tmpdir`. To use another temporary directory, specify the path by giving the argument `-tmpDir`to the client container.

19.2.6. Disable Automated WsGen

By default, the client container will apply WsGen (generation of web services artifacts) on all given archives. To disable that feature (because you already have applied WsGen on your application or because you do not have any web services in your client), add the `-nowsgen` option to the client container.

19.3. Examples

The `earsample` and `jaasclient` examples of the JOnAS examples are packaged for use by the client container. The first example demonstrates the client inside an `.ear` file. The second example demonstrates the use of a standalone client.

Chapter 20.

Defining the Client Deployment Descriptor

This chapter is for the Client component provider; that is, the person in charge of developing the Client components on the client side.

20.1. Principles

The Client component programmer is responsible for providing the deployment descriptor associated with the developed client components.

The client component provider's responsibilities and the Application Assembler's responsibilities are to provide an XML deployment descriptor that conforms to the deployment descriptor's XML DTD as defined in the Java Application Client Specification Version 1.4. (Refer to `$JONAS_ROOT/xml/application-client_1_4.xsd` or http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd.)

To customize the Client components, information not defined in the standard XML deployment descriptor may be needed. Such information might include, for example, the mapping of the name of referenced resources to its JNDI name. This information can be specified during the deployment phase within another XML deployment descriptor that is specific to JOnAS. The JOnAS-specific deployment descriptor's XML schema is located in `$JONAS_ROOT/xml/jonas-client_X_Y.xsd`. The file name of the JOnAS-specific XML deployment descriptor must be `jonas-client.xml`.

JOnAS interprets the `<!DOCTYPE>` tag at the parsing of the deployment descriptor XML files. The parser first tries to get the specified DTD via the classpath, then it uses the specified URL (or path).

The parser gets the specified schema via the classpath (schemas are packaged in the `$JONAS_ROOT/lib/common/ow_jonas.jar` file).

The standard deployment descriptor (`application-client.xml`) should contain structural information that includes the following:

- A Client description
- Environment entries
- EJB references
- Resource references
- Resource env references
- The callback handler to use.

The JOnAS-specific deployment descriptor (`jonas-client.xml`) may contain information that includes the following:

- The JNDI name of the external resources referenced by a Client component
- The JNDI name of the external resources environment referenced by a Client component
- The JNDI name of the beans referenced by a Client component
- The security aspects including the JAAS file, the JAAS entry, and a login/password to use for a specific callback handler.

20.2. Examples of Client Deployment Descriptors

- Example of a standard Client Deployment Descriptor (`application-client.xml`):

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

<application-client 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/application-client_1_4.xsd"
        version="1.4">

  <display-name>Client of the earsample</display-name>
  <description>client of the earsample</description>

  <env-entry>
    <env-entry-name>envEntryString</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Test of envEntry of application-client.xml
        file</env-entry-value>
  </env-entry>

  <ejb-ref>
    <ejb-ref-name>ejb/Op</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>org.objectweb.earsample.beans.secusb.OpHome</home>
    <remote>org.objectweb.earsample.beans.secusb.Op</remote>
    <ejb-link>secusb.jar#EarOp</ejb-link>
  </ejb-ref>

  <resource-ref>
    <res-ref-name>url/jonas</res-ref-name>
    <res-type>java.net.URL</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

  <callback-handler>
      org.objectweb.jonas.security.auth.callback.LoginCallbackHandler
      </callback-handler>

</application-client>
```

- Example of a specific Client Deployment Descriptor (`jonas-client.xml`):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<jonas-client xmlns="http://www.objectweb.org/jonas/ns"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.objectweb.org/jonas/ns
      http://www.objectweb.org/jonas/ns/jonas-client_4_0.xsd">

<jonas-client>

  <jonas-resource>
    <res-ref-name>url/jonas</res-ref-name>
    <jndi-name>http://jonas.objectweb.org</jndi-name>
  </jonas-resource>

  <jonas-security>
    <jaasfile>jaas.config</jaasfile>
    <jaasentry>earsample</jaasentry>
    <username>jonas</username>
    <password>jonas</password>
  </jonas-security>
```

```
</jonas-client>
```

20.3. Tips

Although some characters, such as ">", are legal, it is good practice to replace them with XML entity references.

The following is a list of the predefined entity references for XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Chapter 21.
Client Packaging

This chapter is for the Client component provider; that is, the person in charge of developing the client components on the client side. It describes how the client components should be packaged.

21.1. Principles

Client components are packaged for deployment in a standard Java programming language Archive file called a JAR file (Java ARchive). The document root contains a subdirectory called `META-INF`, which contains the following files and directories:

- `application-client.xml`: The standard XML deployment descriptor in the format defined in the J2EE 1.3 Specification. Refer to `$JONAS_ROOT/xml/application-client_1_4.xsd`.

- `jonas-client.xml`: The optional JOnAS-specific, XML deployment descriptor in the format defined in `$JONAS_ROOT/xml/jonas-client_X_Y.xsd`.

The manifest of this client JAR must contain the name of the class to launch (containing the main method). This is defined by the value of the `Main-Class` attribute of the manifest file. For a standalone client (not bundled in an EAR), all the EJB classes (except the skeleton) on which lookups will be performed must be included.

21.1.1. Client Packaging Example

Two examples of building a Java client are provided:

- The first is the `build.xml` of the `earsample` example with a Java client inside the EAR (see http://jonas.objectweb.org/current/examples/earsample/build.xml). Refer to the client1jar and client2jar targets.

- The second is the `build.xml` of the `jaasclient` example with a Java standalone client, which performs a lookup on an EJB.

  See http://jonas.objectweb.org/current/examples/jaasclient/build.xml; refer to the clientjars target.

V. J2EE Application Assembler's Guide

This section contains information for the J2EE Application Assembler; that is, the person in charge of combining one or more components (ejb-jars and/or wars) to create a J2EE application.

Table of Contents

Defining the EAR Deployment Descriptor

This chapter is for the Application Provider; that is, the person in charge of combining one or more components (EJB-JARs and/or WARs) to create a J2EE application.

22.1. Principles

The application programmer is responsible for providing the deployment descriptor associated with the developed application (Enterprise ARchive). The Application Assembler's responsibilities is to provide a XML deployment descriptor that conforms to the deployment descriptor's XML schema as defined in the J2EE specification version 1.4. Refer to `$JONAS_ROOT/xml/application_1_4.xsd` (http://jonas.objectweb.org/current/xml/application_1_4.xsd.)

To deploy J2EE applications on the application server, all information is contained in one XML deployment descriptor. The file name for the application XML deployment descriptor is `application.xml` and it must be located in the top level META-INF directory.

The parser gets the specified schema via the classpath (schemas are packaged in the `$JONAS_ROOT/lib/common/ow_jonas.jar` file).

Some J2EE application examples are provided in the JOnAS distribution:

- `$JONAS_ROOT/examples/alarm` for the Alarm demo
- `$JONAS_ROOT/examples/cmp2` for the Cmp2 example
- `$JONAS_ROOT/examples/earsample` for the EarSample example
- `$JONAS_ROOT/examples/petstore1.3` for the Blueprints Petstore application.

The standard deployment descriptor should contain structural information that includes the following:

- EJB components
- Web components
- Client components
- Alternate Deployment Descriptor for theses components
- Security role.

There is no JOnAS-specific deployment descriptor for the Enterprise ARchive.

22.2. Simple Example of Application Deployment Descriptor

```
<application 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/application_1_4.xsd"
     version="1.4">

  <display-name>Simple example of application</display-name>
  <description>Simple example</description>

  <module>
    <ejb>ejb1.jar</ejb>
```

```
    </module>
    <module>
      <ejb>ejb2.jar</ejb>
    </module>

    <module>
      <web>
        <web-uri>web.war</web-uri>
        <context-root>web</context-root>
      </web>
    </module>
</application>
```

22.3. Advanced Example

This is an advanced example of an Application Deployment Descriptors with alternative DD and security.

```
<application 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/application_1_4.xsd"
      version="1.4">

  <display-name>Ear Security</display-name>
  <description>Application with alt-dd and security</description>
  <module>
    <web>
      <web-uri>admin.war</web-uri>
      <context-root>admin</context-root>
    </web>
  </module>
  <module>
    <ejb>ejb.jar</ejb>
    <alt-dd>altdd.xml</alt-dd>
  </module>
  <security-role>
    <role-name>admin</role-name>
  </security-role>
</application>
```

22.4. Tips

Although some characters, such as ">", are legal, it is good practice to replace them with XML entity references.

The following is a list of the predefined entity references for XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Chapter 23.
EAR Packaging

This chapter is for the Application Assembler; that is, the person in charge of combining one or more J2EE components (EJB-JARs and/or WARs) to create a J2EE application. It describes how the J2EE components should be packaged to create a J2EE application.

23.1. Principles

J2EE applications are packaged for deployment in a standard Java programming language Archive file called an EAR file (Enterprise ARchive). This file can contain the following:

The web components (WAR)

> One or more WARs that contain the web components of the J2EE application. Due to the class loader hierarchy, when the WARs are packaged in a J2EE application, it is not necessary to package bean classes in the `WEB-INF/lib` directory.

> Details about this class loader hierarchy are described in Chapter 5 *JOnAS Class Loader Hierarchy*.

The beans (EJB-JAR)

> One or more EJB-JARs, which contain the beans of the J2EE application.

The libraries (JAR)

> One or more jars which contain the libraries (tag libraries and any utility libraries) used for the J2EE application.

The J2EE deployment descriptor

> The standard XML deployment descriptor in the format defined in the J2EE 1.4 specification (refer to `$JONAS_ROOT/xml/application_1_5.xsd` or http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd). This deployment descriptor must be stored with the name `META-INF/application.xml` in the EAR file.

23.1.1. EAR Packaging Example

Before building an EAR file for a J2EE application, the EJB-JARs and the WARs that will be packaged in the J2EE application must be built and the XML deployment descriptor (`application.xml`) must be written.

Then, the EAR file (`<j2ee-application>.ear`) can be built using the `jar` command:

```
cd <your_j2ee_application_directory>
jar cvf <j2ee-application>.ear *
```

VI. Advanced Topics

This section contains information for advanced JOnAS users.

Table of Contents

JOnAS Services

This chapter is intended for advanced JOnAS users who require that some "external" services run along with the EJB server. A service is something that may be initialized, started, and stopped. JOnAS itself already defines a set of services, some of which are cornerstones of the EJB Server. The JOnAS pre-defined services are listed in Section 3.5 *Configuring JOnAS Services*.

EJB application developers may need to access other services, for example another Web container or a Versant container, for their components. Thus, it is important that such services be able to run along with the EJB server. To achieve this, it is possible to define them as JOnAS services.

This chapter describes how to define a new JOnAS service and how to specify which service should be started with the EJB server.

24.1. Introducing a New Service

The customary way to define a new JOnAS service is to encapsulate it in a class whose interface is known by JOnAS. More precisely, such a class provides a way to initialize, start, and stop the service. Then, the jonas.properties file must be modified to make JOnAS aware of this service.

24.1.1. Defining the Service Class

A JOnAS service is represented by a class that implements the interface `org.objectweb.jonas.service.Service`, and, thus should implement the following methods:

- `public void init(Context ctx) throws ServiceException;`
- `public void start() throws ServiceException;`
- `public void stop() throws ServiceException;`
- `public boolean isStarted();`
- `public String getName();`
- `public void setName(String name);`

It should also define a public constructor with no argument.

These methods will be called by JOnAS for initializing, starting, and stopping the service. Configuration parameters are provided to the initialization method through a naming context. This naming context is built from properties defined in the `jonas.properties` file as explained in Section 24.1.2 *Modifying the jonas.properties File*.

The Service class should look like the following:

```
package a.b;
import javax.naming.Context;
import javax.naming.NamingException;
import org.objectweb.jonas.service.Service;
import org.objectweb.jonas.service.ServiceException;
.....
public class MyService implements Service {
    private String name = null;
    private boolean started = false;
    .....
    public void init(Context ctx) throws ServiceException {
        try {
```

```
            String p1 = (String) ctx.lookup("jonas.service.serv1.p1");
            .....
        } catch (NamingException e) {
            throw new ServiceException("....", e);
        }
        .....
    }
    public void start() throws ServiceException {
        .....
        this.started = true;
    }
    public void stop() throws ServiceException {
        if (this.started) {
            this.started = false;
            .....
        }
    }
    public boolean isStarted() {
        return this.started;
    }
    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

24.1.2. Modifying the jonas.properties File

The service is defined and its initialization parameters specified in the `jonas.properties` file. First, choose a name for the service (for example, "serv1"), then do the following:

- Add this name to the `jonas.services` property; this property defines the set of services (comma-separated) that will be started with JOnAS, *in the order of this list*.

- Add a `jonas.service.serv1.class` property specifying the service class.

- Add as many `jonas.service.serv1.XXX` properties specifying the service initialization parameters, as will be made available to the service class via the Context argument of the `init` method.

This is illustrated as follows:

```
jonas.services                  .......,serv1
jonas.service.serv1.class       a.b.MyService
jonas.service.serv1.p1          value
```

24.1.3. Using the New Service

The new service has been given a name in `jonas.properties`. With this name, it is possible to get a reference on the service implementation class by using the ServiceManager method: `getService(name)`. The following is an example of accessing a Service:

```
import org.objectweb.jonas.service.ServiceException;
import org.objectweb.jonas.service.ServiceManager;

  MyService sv = null;
```

```
    // Get a reference on MyService.
    try {
        sv = (MyService) ServiceManager.getInstance().getService("serv1");
    } catch (ServiceException e) {
        Trace.errln("Cannot find MyService:"+e);
    }
```

24.1.4. Adding the Class of the New Service to JOnAS

Package the class of the service into a `.jar` file and add the JAR in the `JONAS_ROOT/lib/ext` directory. All the libraries required by the service can also be placed in this directory.

24.2. Advanced Understanding

Refer to the JOnAS sources for more details about the classes mentioned in this section.

24.2.1. JOnAS built-in Services

The existing JOnAS services are the following:

Service name	Service class
registry	RegistryServiceImpl
ejb	EJBServiceImpl
ear	EarServiceImpl
dbm	DatabaseServiceImpl
jms	JmsServiceImpl
jmx	JmxServiceImpl
jtm	TransactionServiceImpl
mail	MailServiceImpl
resource	ResourceServiceImpl
security	JonasSecurityServiceImpl
ws	AxisWSService

If all of these services are required, they will be launched in the following order: `registry`, `jmx`, `security`, `jtm`, `dbm`, `mail`, `jms`, `resource`, `ejb`, `ws`, `web`, `ear`.

`jmx`, `security`, `dbm`, `mail`, `resource` are optional when you are using service `ejb`.

`registry` must be launched first.

◇ **Notes**

For compatibility with previous versions of JOnAS, if `registry` is not set as the first service to launch, JOnAS automatically launches the `registry` service. Thus, the `jtm` service must be launched before these services.

`dbm`, `jms`, `resource`, and `ejb` depend on `jtm`.

> `ear` depends on `ejb` and `web` (that provide the EJB and WEB containers), thus these services must be launched before the `ear` service.
>
> `ear` and `web` depend on `ws`, thus the `ws` service must be launched before the `ear` and `web` services.

It is possible to launch a stand-alone Transaction Manager with only the `registry` and `jtm` services.

In this case, a `jonas.properties` file looks like the following:

```
.....
jonas.services     registry,jmx,security,jtm,dbm,mail,jms,ejb,resource,serv1

jonas.service.registry.class  \
    org.objectweb.jonas.registry.RegistryServiceImpl
jonas.service.registry.mode        automatic

jonas.service.dbm.class       org.objectweb.jonas.dbm.DatabaseServiceImpl
jonas.service.dbm.datasources Oracle1

jonas.service.ejb.class       org.objectweb.jonas.container.EJBServiceImpl
jonas.service.ejb.descriptors ejb-jar.jar
jonas.service.ejb.parsingwithvalidation  true
jonas.service.ejb.minworkthreads      10
jonas.service.ejb.maxworkthreads      100

jonas.service.web.class       \
   org.objectweb.jonas.web.catalina.CatalinaJWebContainerServiceImpl
jonas.service.web.descriptors        war.war
jonas.service.web.parsingwithvalidation true

jonas.service.ear.class       org.objectweb.jonas.ear.EarServiceImpl
jonas.service.ear.descriptors j2ee-application.ear
jonas.service.ear.parsingwithvalidation true

jonas.service.jms.class       org.objectweb.jonas.jms.JmsServiceImpl
jonas.service.jms.mom         org.objectweb.jonas_jms.JmsAdminForJoram
jonas.service.jms.collocated  true
jonas.service.jms.url         joram://localhost:16010

jonas.service.jmx.class       org.objectweb.jonas.jmx.JmxServiceImpl

jonas.service.jtm.class       \
   org.objectweb.jonas.jtm.TransactionServiceImpl
jonas.service.jtm.remote      false
jonas.service.jtm.timeout     60

jonas.service.mail.class      org.objectweb.jonas.mail.MailServiceImpl
jonas.service.mail.factories  MailSession1

jonas.service.security.class  \
    org.objectweb.jonas.security.JonasSecurityServiceImpl

jonas.service.resource.class  \
    org.objectweb.jonas.resource.ResourceServiceImpl
jonas.service.resource.resources        MyRA

jonas.service.serv1.class     a.b.MyService
jonas.service.serv1.p1        John
```

24.2.2. The ServiceException

The `org.objectweb.jonas.service.ServiceException` exception is defined for Services. Its type is `java.lang.RuntimeException` and it can encapsulate any `java.lang.Throwable`.

24.2.3. The ServiceManager

The `org.objectweb.jonas.service.ServiceManager` class is responsible for creating, initializing, and launching the services. It can also return a service from its name and list all the services.

JOnAS and the Connector Architecture

This chapter is provided for advanced JOnAS users concerned with EAI (Enterprise Application Integration).

25.1. Introducing the Connector Architecture

The Java Connector Architecture (Connectors) defines a way for enterprise applications (based on EJB, Servlet, JSP, or J2EE clients) to communicate with existing external Enterprise Information Systems (EIS) through an application server such as JOnAS. This requires the use of a third-party software component called a Resource Adapter (RA) for each type of EIS. A Resource Adapter is an architecture component, comparable to a software driver, that connects the EIS, the application server, and the enterprise application. The RA is generally made available by an EIS vendor.

The RA provides an interface (the Common Client Interface or CCI) to the enterprise application (EJBs) for accessing the EIS. The RA also provides standard interfaces for plugging into the application server, so that the EIS and application server can collaborate to keep all system-level mechanisms transparent from the application components. The application performs "business logic" operations on the EIS data using the RA client API (CCI), while transactions, connections (including pooling), and security on the EIS is managed by JOnAS through the RA (system contract).
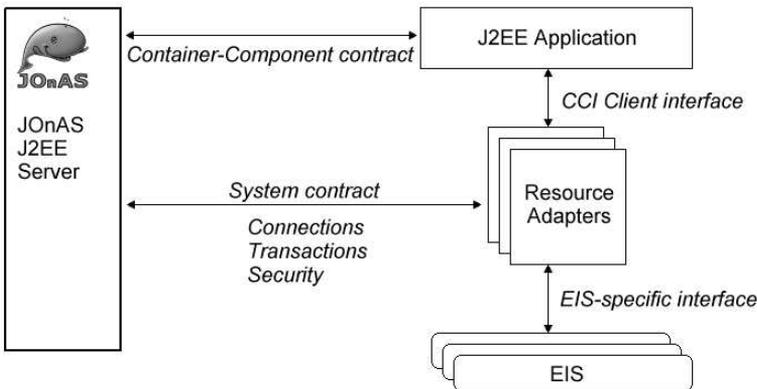

Figure 25-1. Connector Architecture

25.2. Defining the JOnAS Connector Deployment Descriptor

Using a Connector Resource Adapter with JOnAS involves the following steps:

1. The bean provider must specify the connection factory requirements by declaring a *resource manager connection factory reference* in its EJB deployment descriptor. For example:

```
<resource-ref>
<res-ref-name>eis/MyEIS</res-ref-name>
<res-type>javax.resource.cci.ConnectionFactory</res-type>
<res-auth>Container</res-auth>
</resource-ref>
```
The mapping of the *connection factory* to the actual JNDI name (here `adapt_1`) is done in the JOnAS-specific deployment descriptor with the following element:
```
<jonas-resource>
<res-ref-name>eis/MyEIS</res-ref-name>
<jndi-name>adapt_1</jndi-name>
</jonas-resource>
```

This means that the bean programmer will have access to a connection factory instance using the JNDI interface via the `java:comp/env/eis/MyEIS` name:
```
// obtain the initial JNDI naming context
Context inictx = new InitialContext();

// perform JNDI lookup to obtain the connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)
        inictx.lookup("java:comp/env/eis/MyEIS<");
```

The bean programmer can then get a connection by calling the method `getConnection` on the connection factory.
```
javax.resource.cci.Connection cx = cxf.getConnection();
```

The returned connection instance represents an application-level handle to a physical connection for accessing the underlying EIS.

After finishing with the connection, you must close it using the `close` method on the `Connection` interface:
`cx.close();`

2. The resource adapters must be deployed before being used by the application. Deploying the resource adapter requires that you build a *JOnAS-specific resource adapter configuration* file that will be included in the resource adapter.

   This jonas-ra XML file is used to configure the resource adapter in the operational environment and reflects the values of all properties declared in the deployment descriptor for the resource adapter, plus additional JOnAS-specific configuration properties. JOnAS provides a deployment tool (Section 6.7 *RAConfig*) that is capable of building this XML file from an RA deployment descriptor inside a RAR file. For example:
   `RAConfig -j adap_1 ra`

   These properties may be specific for each resource adapter and its underlying EIS. They are used to configure the resource adapter via its `managedConnectionFactory` class. It is mandatory that this class provide a getter and setter method for each of its supported properties (as it is required in the Connector Architecture specification). Refer to Chapter 56 *Configuring Resource Adapters* for a complete description of the JOnAS-specific configuration file, `jonas-ra.xml`.

3. The JOnAS *resource service* must be configured and started at JOnAS launching time:

   In the `jonas.properties` file:

   a. Insert the name `resource` in the `jonas.services` property.

   b. Use one of the following methods to deploy an RAR file:

   - The name of the `resource adapter` files (the `.rar` suffix is optional) must be added in the list of Resource Adapter to be used in the `jonas.service.resource.resources` RAR file. If the `.rar` suffix is not used on the property, it will be used when trying to allocate the specified Resource Adapter.
     `jonas.service.resource.resources  MyEIS.rar, MyEIS1`

- Place the RAR file in the autoload directory of $JONAS_BASE; default value is $JONAS_BASE/rars/autoload. Note that it may be different if jonas.service.resource.autoload in jonas.properties is configured differently.

- Add the RAR:
  jonas admin -a xxx.rar

Chapter 26.
JMS User's Guide

This chapter is provided for advanced JOnAS users concerned with JMS (Java Message Service).

As required by the J2EE v1.4 specification, application components (servlets and enterprise beans) can use JMS for Java messaging. Furthermore, applications can use Message-driven Beans (MDBs) for asynchronous EJB method invocation, as specified by the EJB 2.1 specification.

JOnAS supports the JMS 1.1 specification, which offers a domain-independent approach to programming the client application. Prior to JMS 1.1, client programming for point-to-point and publish/subscribe domains was achieved using similar, but separate, class hierarchies. With JMS 1.1, it is now possible to engage queues and topics in the same transaction.

Enterprise Bean providers can use JMS connection factory resources via *resource references*, and JMS destination resources (JMS queues and JMS topics) via *resource environment references*. Thus, they are able to provide JMS code, inside an EJB method or web component method, for sending or synchronously receiving messages to/from a JMS queue or topic.

The EJB container and the Web container can allow for JMS operations within a global transaction, which may include other resources such as databases.

JOnAS integrates a third-party JMS implementation, JORAM (http://joram.objectweb.org/), which is the default JMS service, and for which a J2EE1.4 compliant Resource Adapter archive file is also provided. Other JMS providers, such as SwiftMQ (http://www.swiftmq.com/) and WebSphere MQ (http://www-3.ibm.com/software/integration/mqfamily/), can easily be integrated.

A JMS provider can be integrated within JOnAS by deploying a corresponding *resource adapter*. This is the preferred method as the JMS service will eventually become deprecated in future JOnAS releases. Also, this method allows deployment of 2.1 MDBs (which is impossible with the JMS service).

To perform JMS operations, application components use JMS-administered objects such as connection factories and destinations. Refer to Section 26.4 *JMS Administration* for an explanation of how to create those objects.

26.1. JMS is Pre-installed and Configured

To use JMS with JOnAS, no additional installation or configuration operations are required. JOnAS contains:

- The Java[TM] Message Service API 1.1, currently integrated with the JOnAS distribution,
- A JMS implementation. Currently, the OpenSource JORAM (http://joram.objectweb.org), is integrated with the JOnAS distribution; thus no installation is necessary.

Additionally, another JMS implementation, the SwiftMQ product, has been used with JOnAS.

26.2. Writing JMS Operations Within an Application Component

To send (or synchronously receive) JMS messages, an application component requires access to JMS-administered objects (that is, to connection factories for creating connections to JMS resources, and to destination objects (queue or topic), which are the JMS entities used as destinations within JMS sending operations). Both are made available through JNDI by the JMS provider administration facility.

You can find a sample JMS application in `$JONAS_ROOT/examples/src/jms/`; it is described in Section 26.6 *A JMS EJB Example*.

26.2.1. Accessing the Connection Factory

The EJB specification introduces the concept of *resource manager connection factory references*. This concept also appears in the J2EE v1.4 specification. It is used to create connections to a resource manager. To date, three types of *resource manager connection factories* are considered:

- DataSource objects (`javax.sql.DataSource`) represent connection factories for JDBC connection objects.

- JMS connection factories. The connection factories for JMS connection objects are:

  - `javax.jms.ConnectionFactory`

  - `javax.jms.QueueConnectionFactory`

  - `javax.jms.TopicConnectionFactory`.

- Java Mail connection factories. The connection factories for Java Mail connection objects are `javax.mail.Session` or `javax.mail.internet.MimePartDataSource`.

The connection factories of interest here are the second type, which should be used to get JMS connection factories.

Note that starting with JMS 1.1, it is recommended that you use only the `javax.jms.ConnectionFactory` (rather than `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`). However, the JMS 1.1 implementation is fully backwards-compatible and existing applications will work as-is.

The standard deployment descriptor should contain the following `resource-ref` element:

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

This means that the programmer will have access to a `ConnectionFactory` object using the JNDI name `java:comp/env/jms/conFact`. The source code for obtaining the factory object is the following:

```
ConnectionFactory qcf = (ConnectionFactory)
          ctx.lookup("java:comp/env/jms/conFact");
```

The mapping to the actual JNDI name of the connection factory (as assigned by the JMS provider administration tool), `CF` in the example, is defined in the JOnAS-specific deployment descriptor with the following element:

```
<jonas-resource>
<res-ref-name>jms/conFact</res-ref-name>
<jndi-name>CF</jndi-name>
</jonas-resource>
```

26.2.2. Accessing the Destination Object

Accessing a JMS destination within the code of an application component requires using a `Resource Environment Reference`, which is represented in the standard deployment descriptor as follows:

```
<resource-env-ref>
<resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
<resource-env-ref-type>javax.jms.Queue<resource-env-ref-type>
</resource-env-ref>
```

The application component's source code should contain:

```
Queue q = (Queue) ctx.lookup("java:comp/env/jms/stockQueue");
```

The mapping to the actual JNDI name (for example, "myQueue") is defined in the JOnAS-specific deployment descriptor in the following way:

```
<jonas-resource-env>
<resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
<jndi-name>myQueue<jndi-name>
</jonas-resource-env>
```

26.2.3. Writing JMS Operations

A typical method performing a message-sending JMS operation looks like the following:

```
void sendMyMessage() {

   ConnectionFactory cf = (ConnectionFactory)
         ctx.lookup("java:comp/env/jms/conFact");
   Queue queue = (Queue) ctx.lookup("java:comp/env/jms/stockQueue");
   Connection conn = cf.createConnection();
   Session sess = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);

   MessageProducer  mp = sess.createProducer((Destination)queue);
   ObjectMessage msg = sess.createObjectMessage();
   msg.setObject("Hello");
   sender.send(msg);
   sess.close();
   conn.close();
 }
```

It is also possible for an application component to *synchronously* receive a message. Here is an EJB method that performs synchronous message reception on a queue:

```
public String recMsg() {
    ConnectionFactory cf = (ConnectionFactory)
          ctx.lookup("java:comp/env/jms/conFact");
    Queue queue = (Queue) ctx.lookup("java:comp/env/jms/stockQueue");
    Connection conn = cf.createConnection();
    Session sess = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer mc = sess.createConsumer((Destination)queue);
    conn.start();
    ObjectMessage msg = (ObjectMessage) mc.receive();
    String msgtxt =  (String) msg.getObject();
    sess.close();
    conn.close();
    return msgtxt;
}
```

A method that performs JMS operations should always contain the session create and close statements, as follows:

```
public void doSomethingWithJMS (...) {
  ...
  session = connection.createSession(...);
  ... // JMS operations
  session.close();
}
```

The contained JMS operations will be a part of the transaction, if there is one, when the JOnAS server executes the method.

Note

Never send and receive a particular message in the same transaction because JMS sending operations are performed only at commit time.

The previous examples illustrate point-to-point messaging. However, application components can also be developed using the publish/subscribe JMS API (that is, using the `Topic` instead of the `Queue` destination type). This offers the capability of broadcasting a message to several message consumers at the same time.

The following example illustrates a typical method for publishing a message on a JMS topic and demonstrates how interfaces have been simplified since JMS 1.1.

```
public void sendMsg(java.lang.String s) {
    ConnectionFactory cf = (ConnectionFactory)
                  ictx.lookup("java:comp/env/jms/conFactSender");
    Topic topic = (Topic) ictx.lookup("java:comp/env/jms/topiclistener");
    Connection conn = cf.createConnection();
    Session session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer mc = session.createConsumer((Destination)topic);
    ObjectMessage message = session.createObjectMessage();
    message.setObject(s);
    mc.send(message);
    session.close();
    conn.close();
}
```

26.2.4. Transactions and JMS Sessions Within an Application Component

JMS session creation within an application component will result in different behaviors, depending on whether the session is created at execution time within or outside a transaction. In fact, the parameters of the `createSession(boolean transacted, int acknowledgeMode)` method are *never* taken into account.

- If the session creation occurs outside a transaction, the parameters is considered as being `transacted = false` and `acknowl`for example, `Mode = AUTO_ACKNOWLEDGE`. This means that each operation of the session is immediately executed.

- If the session creation occurs inside a transaction, the parameters have no meaning, the session may be considered as transacted, and the commit and rollback operations are handled by the JOnAS server at the level of the associated XA resource.

26.2.5. Authentication with JMS

If your JMS implementation performs user authentication, the following methods can be used on connection factories:

- The `createConnection(String userName, String password)` method can be used on `ConnectionFactory`
- The `createQueueConnection(String userName, String password)` method can be used on `QueueConnectionFactory`
- The `createTopicConnection(String userName, String password)` method can be used on `TopicConnectionFactory`

Note

> Starting with JMS 1.1, it is recommended that you use only the `javax.jms.ConnectionFactory` (rather than `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`). However, the JMS 1.1 implementation is fully backwards-compatible and existing applications will work as-is.

26.3. Some Programming Rules and Restrictions When Using JMS within EJB

This section presents some programming restrictions and rules for using JMS operations within entity components.

26.3.1. Connection Management

Depending on the JMS implementation and the application, it may be desirable to keep the JMS connections open for the life of the bean instance or for the duration of the method call. These two programming modes are illustrated in the following example (this example illustrates a stateful Session Bean):

```
public class EjbCompBean implements SessionBean {
...
QueueConnectionFactory qcf = null;
Queue queue = null;

public void ejbCreate() {
   ....
   ictx = new InitialContext();
   qcf = (QueueConnectionFactory)
      ictx.lookup("java:comp/env/jms/conFactSender");
   queue = (Queue) ictx.lookup("java:comp/env/jms/queue1");
}

public void doSomethingWithJMS (...) {
   ...
   Connection conn = qcf.createConnection();
   Session session = conn.createSession(...);
   ... // JMS operations
   session.close();
   conn.close();
```

```
 }
...
}
```

To keep the connection open during the life of a bean instance, the programming style shown in the following example is preferred, since it avoids many connection opening and closing operations:

```
public class EjbCompBean implements SessionBean {
...
ConnectionFactory qcf = null;
Queue queue = null;
Connection conn = null;

public void ejbCreate() {
   ....
   ictx = new InitialContext();
   cf = (ConnectionFactory)
      ictx.lookup("java:comp/env/jms/conFactSender");
   queue = (Queue) ictx.lookup("queue1");
   conn = cf.createConnection();
}

public void doSomethingWithJMS (...) {
   ...
   Session session = conn.createSession(...);
   ... // JMS operations
   session.close();
}

public void ejbRemove() {
   conn.close();
}

...
}
```

Be aware that maintaining JMS objects in the bean state is not always possible, depending on the type of bean.

- For a stateless Session Bean, the bean state is not maintained across method calls. Therefore, the JMS objects should always be initialized and defined in each method that performs JMS operations.
- For an Entity Bean, an instance may be passivated, and only the persistent part of the bean state is maintained. Therefore, it is recommended that the JMS objects be initialized and defined in each method performing JMS operations. If these objects are defined in the bean state, they can be initialized in the ejbActivate method (if the connection is created in the ejbActivate method, be sure to close it in the ejbPassivate method).
- For a stateful Session Bean (as shown in the previous example), JMS objects can be defined in the bean state. Stateful Session Bean instances can be passivated (not in the current version of JOnAS, but it is scheduled for the summer of 2004). Since connection factories and destinations are serializable objects, they can be initialized only in ejbCreate. However, be aware that a connection must be closed in ejbPassivate (with the state variable set to null) and recreated in ejbActivate.

Note that, due to a known problem with the Sun JDK 1.3 on Linux, the close of the connection can block. The problem is fixed with JDK 1.4.

26.3.2. Starting Transactions after JMS Connection or Session Creation

Currently, it is not possible to start a bean-managed transaction after the creation of a JMS session and have the JMS operations involved in the transaction. In the following code example, the JMS operations will not occur within the `ut` transaction:

```
public class EjbCompBean implements SessionBean {
...

public void doSomethingWithJMS (...) {
   ...
   Connection conn = cf.createConnection();
   Session session = conn.createSession(...);
   ut = ejbContext.getUserTransaction();
   ut.begin();
   ... // JMS operations
   ut.commit();
   session.close();
   conn.close();
 }

...
}
```

To have the session operations involved in the transaction, the session creation and close should be inside the transaction boundaries, and the connection creation and close operations can either be both outside the transaction boundaries or both inside the transaction boundaries, as follows:

```
public class EjbCompBean implements SessionBean {
...

public void doSomethingWithJMS (...) {
   ...
   Connection conn = qcf.createConnection();
   ut = ejbContext.getUserTransaction();
   ut.begin();
   Session session = conn.createSession(...);
   ... // JMS operations
   session.close();
   ut.commit();
   conn.close();
 }

...
}
```

or

```
public class EjbCompBean implements SessionBean {
...

public void doSomethingWithJMS (...) {
   ...
   ut = ejbContext.getUserTransaction();
   ut.begin();
   Connection conn = cf.createConnection();
   Session session = conn.createSession(...);
   ... // JMS operations
   session.close();
   conn.close();
   ut.commit();
```

```
  }
...
}
```

Programming EJB components with bean-managed transactions can result in complex code. Using container-managed transactions can help avoid problems such as those previously described.

26.4. JMS Administration

Applications using messaging require some JMS-administered objects: *connection factories* and *destinations*. These objects are created via the proprietary administration interface (not standardized) of the JMS provider. For simple cases, it is possible to have either the jms service, or the JMS resource adapter, performing administration operations during startup.

As provided, the default JMS service and JORAM adapter configurations automatically create six connection factories and two destination objects.

The six connection factories automatically created are described in the following table:

JNDI name	JMS type	Usage
CF	ConnectionFactory	To be used by an application component to create a connection.
QCF	QueueConnectionFactory	To be used by an application component to create a QueueConnection.
TCF	TopicConnectionFactory	To be used by an application component to create a TopicConnection.
JCF	ConnectionFactory	To be used by any other Java component (for instance a client) to create a connection.
JQCF	QueueConnectionFactory	To be used by any other Java component (for instance a client) to create a QueueConnection.
JTCF	TopicConnectionFactory	To be used by any other Java component (for instance a client) to create a TopicConnection.

The CF, QCF and TCF connection factories are *managed* connection factories. The application components should use only managed connection factories to allow JOnAS to manage the JMS resources created via these connection factories (the JMS sessions). In contrast, JCF, JQCF and JTCF are *non-managed* connection factories. They are used by Java components implementing a JMS client behavior, but running outside the application server.

The two destinations automatically created are described in the following table:

JNDI name	JMS type	Usage
sampleQueue	Queue	Can be equally used by an EJB component or a Java component.
sampleTopic	Topic	Can be equally used by an EJB component or a Java component.

26.4.1. JMS Service Administration

To use the JMS service in the default configuration, all that is necessary is requiring the use of the JMS service in the `jonas.properties` file:

```
jonas.services        security,jtm,dbm,jms,ejb
```

JOnAS will not create additional connection factories when using the default configuration. However, JOnAS can create requested destination objects at server launching time, if specified in the `jonas.properties` file. To do this, specify the JNDI names of the Topic and Queue destination objects to be created in a `jonas.service.jms.topics` and `jonas.service.jms.queues` property respectively, as follows:

```
// JOnAS server creates 2 topic destinations (t1,t2)
jonas.service.jms.topics t1,t2

// JOnAS server creates 1 queue destination (myQueue)
jonas.service.jms.queuesmyQueue
```

It is recommended that programmers use *resource references* and *resource environment references* to access the connection factories and destination objects created by JOnAS, as presented in Section 26.2 *Writing JMS Operations Within an Application Component*.

26.4.2. JMS Resource Adapter Configuration

Starting with JOnAS release 4.1, it is recommended that you deploy a JMS resource adapter instead of using the `jms` service. How to do this is explained in Section 3.8 *Configuring JMS Resource Adapters*.

26.5. Running an EJB Performing JMS Operations

All that is necessary to have an Enterprise Bean perform JMS operations is:

```
jonas start
```

The Message-Oriented Middleware (the JMS provider implementation) is automatically started (or at least accessed) and the JMS-administered objects that will be used by the Enterprise Beans are automatically created and registered in JNDI.

Then, the EJB can be deployed as usual with:

```
jonas admin –a XX.jar
```

26.5.1. Accessing the Message-Oriented Middleware as a Service

If the JOnAS property `jonas.services` contains the `jms` service, the JOnAS JMS service will be launched and will eventually try to launch a JMS implementation (for example, the JORAM MOM or the SwiftMQ MOM) through the JMS service in the JOnAS properties file.

For launching the MOM, consider the following possibilities:

- Launching the MOM automatically in the JOnAS JVM.

  This is done using the default values for the configuration options (that is, keeping the JOnAS property `jonas.service.jms.collocated` value `true` in the `jonas.properties` file (see the `jonas.properties` file provided in `$JONAS_ROOT/conf` directory)).
  ```
  jonas.service.jms.collocated true
  ```

In this case, the MOM will be launched automatically at server launching time (command **jonas start**).

✎ **Note**

> To use the JMS resources from a distant host, the `hostname` property value in the default `a3servers.xml` configuration file must be changed from `localhost` to the actual host name. See case 4 (Launching the MOM on another port number) for details on the JORAM configuration.

- Launching the MOM in a separate JVM on the same host.

  To launch JORAM MOM with its default options, use the command: **JmsServer**

  For other MOMs, use the proprietary command.

  In this case, the JOnAS property `jonas.service.jms.collocated` must be set to `false` in the `jonas.properties` file:
  ```
  jonas.service.jms.collocated false
  ```

- Launching the MOM on another host.

  The MOM can be launched on a separate host. In this case, the JOnAS server must be notified that the MOM is running on another host via the JOnAS property `jonas.service.jms.url` in the `jonas.properties` file. For JORAM, its value should be the JORAM URL `joram://host:port` where `host` is the host name, and `port` the default JORAM port number, which is: 16010
  ```
  jonas.service.jms.collocated false
  jonas.service.jms.url      joram://host2:16010
  ```

  For SwiftMQ, the value of the URL is similar to: `smqp://host:4001/timeout=10000`

- Launching the MOM on another port number (for JORAM)

  Changing the default JORAM port number requires a JORAM-specific configuration operation (modifying the `a3servers.xml` configuration file located in the directory where JORAM is explicitly launched). A default `a3servers.xml` file is provided in the `$JONAS_ROOT/conf` directory; this `a3servers.xml` file specifies that the MOM runs on the localhost using the JORAM default port number.

  To launch the MOM on another port number, change the args attribute of the service `fr.dyade.aaa.mom.ConnectionFactory` element in the `a3servers.xml` file and update the `jonas.service.jms.url` property in the `jonas.properties` file.

  The default `a3servers.xml` file is located in `$JONAS_ROOT/conf`. To change the location of this file, the system property `-Dfr.dyade.aaa.agent.A3CONF_DIR="your directory for a3.xml"` must be passed.

  To learn how to change other MOM configuration settings (distribution, multi-servers, and so on), refer to the JORAM documentation on http://joram.objectweb.org.

✎ **Note**

> The MOM may be directly launched by the proprietary command. The command for JORAM is:
>
> ```
> java -DTransaction=NullTransaction fr.dyade.aaa.agent.AgentServer 0 ./s0
> ```
>
> This command corresponds to the default options used by the `JmsServer` command.

The JMS messages are not persistent when launching the MOM with this command. If persistent messages are required, the `-DTransaction=NullTransaction` option should be replaced with

the `-DTransaction=ATransaction` option. Refer to the JORAM documentation for more details about this command.

26.5.2. Accessing the Message-Oriented Middleware as a J2EE1.4 Adapter

With JOnAS, a JMS server can be accessed through a resource adapter that can be deployed.

To deploy such a resource adapter, put the corresponding archive file (`*.rar`) in JOnAS's `rars/autoload` directory, declare it at the end of the `jonas.properties` file, or deploy it manually through the `jonasAdmin` tool.

Configuring and deploying such adapters is explained in Section 3.8 *Configuring JMS Resource Adapters*.

26.6. A JMS EJB Example

This example shows an EJB application that combines an Enterprise Bean sending a JMS message and an Enterprise Bean writing a Database (an Entity Bean) within the same global transaction. It is composed of the following elements:

• A Session Bean, EjbComp, with a method for sending a message to a JMS topic.

• An Entity Bean, Account (the one used in the sample eb with container-managed persistence), which writes its data into a relational database table and is intended to represent a sent message (that is, each time the EjbComp bean sends a message, an Entity Bean instance will be created).

• An EJB client, EjbCompClient, which calls the sendMsg method of the EjbComp bean and creates an Account entity bean, both within the same transaction. For a transaction commit, the JMS message is actually sent and the record corresponding to the entity bean in the database is created. For a rollback, the message is not sent and nothing is created in the database.

• A pure JMS client MsgReceptor, outside the JOnAS server, the role of which is to receive the messages sent by the Enterprise Bean on the topic.

You can find the sample JMS application in `$JONAS_ROOT/examples/src/jms/`; it is described in Section 26.6 *A JMS EJB Example*.

26.6.1. The Session Bean Performing JMS Operations

The bean should contain code for initializing the references to JMS administered objects that it will use. To avoid repeating this code in each method performing JMS operations, it can be introduced in the `ejbCreate` method.

```
public class EjbCompBean implements SessionBean {
...
ConnectionFactory cf = null;
Topic topic = null;

public void ejbCreate() {
   ....
   ictx = new InitialContext();
   cf = (ConnectionFactory)
      ictx.lookup("java:comp/env/jms/conFactSender");
   topic = (Topic) ictx.lookup("java:comp/env/jms/topiclistener");
}
...
}
```

All code that is not necessary for understanding the JMS logic (such as exception management) has been removed from the above example.

The JMS-administered objects `ConnectionFactory` and `Topic` have been made available to the bean by a *resource reference* in the first example, and by a *resource environment reference* in the second example.

The standard deployment descriptor should contain the following element:

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

<resource-env-ref>
  <resource-env-ref-name>jms/topiclistener</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>
```

The JOnAS-specific deployment descriptor should contain the following element:

```
<jonas-resource>
  <res-ref-name>jms/conFactSender</res-ref-name>
  <jndi-name>TCF</jndi-name>
</jonas-resource>

<jonas-resource-env>
  <resource-env-ref-name>jms/topiclistener</resource-env-ref-name>
  <jndi-name>sampleTopic</jndi-name>
</jonas-resource-env>
```

Note that the EjbComp SessionBean will use the administered objects automatically created by JOnAS in the default JMS configuration.

Because the administered objects are now accessible, it is possible to perform JMS operations within a method. The following occurs in the `sendMsg` method:

```
public class EjbCompBean implements SessionBean {
...
public void sendMsg(java.lang.String s) {
    // create Connection, Session and MessageProducer
    Connection conn = null;
    Session session = null;
    MessageProducer mp = null;
    try {
        conn = cf.createConnection();
        session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
        mp = session.createProducer((Destination)topic);
    }
    catch (Exception e) {e.printStackTrace();}

    // send the message to the topic
    try {
        ObjectMessage message;
        message = session.createObjectMessage();
        message.setObject(s);
        mp.send(message);
        session.close();
        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
```

```
    }
}
...
}
```

This method sends a message containing its String argument.

26.6.2. The Entity Bean

The example uses the simple Entity Bean Account for writing data into a database. Refer to the sample eb, which is described in Chapter 2 *Getting Started with JOnAS* and in the *JOnAS Tutorial*.

26.6.3. The Client Application

The client application calls the sendMsg method of the EjbComp bean and creates an AccountImpl Entity Bean, both within the same transaction.

```
public class EjbCompClient {
...
public static void main(String[] arg) {
...
utx = (UserTransaction)
  initialContext.lookup("javax.transaction.UserTransaction");
...
home1 = (EjbCompHome) initialContext.lookup("EjbCompHome");
home2 = (AccountHome) initialContext.lookup("AccountImplHome");
...
EjbComp aJmsBean = home1.create();
Account aDataBean = null;
...
utx.begin();
aJmsBean.sendMsg("Hello commit"); // sending a JMS message
aDataBean = home2.create(222, "JMS Sample OK", 0);
utx.commit();

utx.begin();
aJmsBean.sendMsg("Hello rollback"); // sending a JMS message
aDataBean = home2.create(223, "JMS Sample KO", 0);
utx.rollback();
...
}
}
```

The result of this client execution will be that:

- The "Hello commit" message will be sent and the [222, 'JMS Sample OK', 0] record will be created in the database (corresponding to the Entity Bean 109 creation).
- The "Hello rollback" message will never be sent and the [223, 'JMS Sample KO', 0] record will not be created in the database (since the Entity Bean 110 creation will be canceled).

26.6.4. A Pure JMS Client for Receiving Messages

In this example, the messages sent by the EJB component are received by a simple JMS client that is running outside the JOnAS server, but listening for messages sent on the JMS topic "sampleTopic." It uses the ConnectionFactory automatically created by JOnAS named "JCF".

```
public class MsgReceptor {

static Context ictx = null;
static ConnectionFactory cf = null;
static Topic topic = null;

public static void main(String[] arg) {

    ictx = new InitialContext();
    cf = (ConnectionFactory) ictx.lookup("JCF");
    topic = (Topic) ictx.lookup("sampleTopic");
    ...
    Connection conn = cf.createConnection();
    Session session =
            conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer mc = session.createConsumer((Destination)topic);

    MyListenerSimple listener = new MyListenerSimple();
        mc.setMessageListener(listener);
    conn.start();

    System.in.read(); // waiting for messages

    session.close();
    conn.close();
    ...
}
}

public MyListenerSimple implements javax.jms.MessageListener {
   MyListenerSimple() {}

   public void onMessage(javax.jms.Message msg) {
  try {
  if(msg==null)
    System.out.println("Message: message null ");
  else {
    if(msg instanceof ObjectMessage) {
        String m = (String) ((ObjectMessage)msg).getObject();
        System.out.println ("JMS client: received message ======> " + m);
    } else if(msg instanceof TextMessage) {
        String m = ((TextMessage)msg).getText();
        System.out.println ("JMS client: received message ======> " + m);
    }
  }catch(Exception exc) {
     System.out.println("Exception caught :" + exc);
     exc.printStackTrace();
  }
   }
}
```

Ant EJB Tasks: Using EJB-JAR

The <jonas> nested element uses the GenIC tool to build JOnAS-specific stubs and skeletons and constructs a JAR file that can be deployed to the JOnAS Application Server. The build process always determines if the EJB stubs/skeletons and the EJB-JAR file are up-to-date and performs the minimum amount of work required.

A naming convention for the EJB descriptors is most commonly used to specify the name for the completed JAR file. For example, if the EJB descriptor ejb/Account-ejb-jar.xml is located in the descriptor directory, the <jonas> element searches for a JOnAS-specific EJB descriptor file named ejb/Account-jonas-ejb-jar.xml, and a JAR file named ejb/Account.jar will be written in the destination directory. The <jonas> element can also use the JOnAS naming convention. Using the same example, the EJB descriptor can also be named ejb/Account.xml (no base name terminator here) in the descriptor directory. The <jonas> element will then search for a JOnAS-specific EJB descriptor file called ejb/jonas-Account.xml. This convention does not strictly follow the EJB-JAR naming convention recommendation, but it is supported for backward compatibility with previous version of JOnAS.

Note that when the EJB descriptors are added to the JAR file, they are automatically renamed META-INF/ejb-jar.xml and META-INF/jonas-ejb-jar.xml.

Furthermore, this naming behavior can be modified by specifying attributes in the ejbjar task (for example, basejarname, basenameterminator, and flatdestdir) as well as the iplanet element (for example, suffix).

27.1. ejbjar Parameters

Attribute	Description	Required
destdir	The base directory into which the generated JAR files will be written. Each JAR file is written in directories that correspond to their location within the descriptordir namespace.	Yes
jonasroot	The root directory for JOnAS.	Yes
jonasbase	The base directory for JOnAS. If omitted, it defaults to jonasroot.	No
classpath	The classpath used when generating EJB stubs and skeletons. If omitted, the classpath specified in the "ejbjar" parent task will be used. If specified, the classpath elements will be prefixed to the classpath specified in the parent "ejbjar" task. A nested "classpath" elements can also be used. Note that the needed JOnAS JAR files are automatically added to the classpath.	No
keepgenerated	true if the intermediate Java source files generated by GenIC must not be deleted. If omitted, it defaults to false.	No
nofastrmic	If true, the external RMIC compiler is used by GenIC. The default is false, which means the internal fast RMIC compiler is used.	No
nocompil	true if the generated source files must not be compiled via the Java and RMI compilers. If omitted, it defaults to false.	No

Attribute	Description	Required
novalidation	`true` if the XML deployment descriptors must be parsed without validation. If omitted, it defaults to `false`.	No
javac	Java compiler to use. If omitted, it defaults to the value of `build.compiler` property.	No
javacopts	Options to pass to the Java compiler.	No
protocols	Comma-separated list of protocols (chosen within jrmp, iiop, cmi) for which stubs should be generated. Default is `jrmp`.	No
rmicopts	Options to pass to the rmi compiler.	No
verbose	Indicates whether or not to use -verbose switch. If omitted, it defaults to `false`.	No
additionalargs	Add additional args to GenIC.	No
keep generic	`true` if the generic JAR file used as input to GenIC must be retained. If omitted, it defaults to `false`.	No
suffix	String value appended to the JAR filename when creating each JAR. If omitted, it defaults to `.jar`.	No
nogenic	If this attribute is set to `true`, JOnAS's GenIC will not be run on the EJB JAR. Use this if you prefer to run GenIC at deployment time. If omitted, it defaults to `false`.	No
jvmopts	Additional args to pass to the GenIC JVM.	No
invokecmd	If this attribute is set to `true`, GenIC uses the Javac Sun class to avoid using the 'javac' command line. This is useful for users getting 'java.io.Exception CreateProcess' errors because of too long command lines. Defaults to `false`.	No

As noted above, the jonas element supports additional <classpath> nested elements.

> **Note**
>
> To avoid `java.lang.OutOfMemoryError`, you can use the element `jvmopts` to change the default memory usage.

27.1.1. ejbjar Example

This example shows `ejbjar` being used to generate deployment jars using a JOnAS EJB container. This example requires the naming standard to be used for the deployment descriptors. Using this format creates a EJB JAR file for each variation of `*-jar.xml` that is located in the deployment descriptor directory.

```
<ejbjar srcdir="${build.classes}"
        descriptordir="${descriptor.dir}">
  <jonas destdir="${deploymentjars.dir}"
        jonasroot="${jonas.root}" orb="RMI"/>
  <include name="**/*.xml"/>
  <exclude name="**/jonas-*.xml"/>
  <support dir="${build.classes}">
      <include name="**/*.class"/>
```

```
   </support>
</ejbjar>
```

This example shows `ejbjar` being used to generate a single deployment JAR using a JOnAS EJB container. This example does require the deployment descriptors to use the naming standard. This creates only one EJB-JAR file: `TheEJBJar.jar`

```
<ejbjar srcdir="${build.classes}"
        descriptordir="${descriptor.dir}"
        basejarname="TheEJBJar">
  <jonas destdir="${deploymentjars.dir}"
         jonasroot="${jonas.root}"
         suffix=".jar" orb="${genic.org}"/>
  <include name="**/ejb-jar.xml"/>
  <exclude name="**/jonas-ejb-jar.xml"/>
</ejbjar>
```

27.2. GenIC Ant Task

27.2.1. Why is there a new task?

Previous versions of the JOnAS EjbJar Ant task have some limitations—especially when you are developing webservices.

In previous versions of JOnAS, jonas-ejb-jar constructed the JAR for you, using information gathered from the ejb-jar.xml and from the classes themselves (dependencies using BCEL). But if you have a Session Bean exposed as a webservice, you want to have more files in your archive (`webservices.xml`, `JAX-RPC mapping file`, `WSDL` + imported `WSDL definitions` + imported `XML Schema`).

The older task did not package these files inside the archive, and therefore, when GenIC loaded the descriptors, some dependencies were missing, causing GenIC to throw an exception.

The solution is to let the developer create the JAR file, so that the exact content of the file can be controlled.

The file should have at least the following content:

- META-INF/ejb-jar.xml
- META-INF/jonas-ejb-jar.xml
- **/*.class

Notice that webservices files are optional.

27.2.2. Task Attributes

The `genic` task supports most attributes of the `jonas-ejb-jar` task.

Differences:

- Removed `destdir`: JAR files are directly modified
- Removed `classpath`: classpath is now set as inner element
- Removed `novalidation`: validation attribute is used now

- Removed `keepgeneric`: not meaningful (input JAR is already specific)
- Removed `suffix`: not meaningful (no JAR generated)
- Removed `nogenic`: not meaningful (we want to run GenIC)

Attribute	Description	Required
jonasroot	The root directory for JOnAS.	Yes (Can be read from project properties if not set)
jonasbase	The base directory for JOnAS. If omitted, it defaults to `jonasroot`.	No
keepgenerated	`true` if the intermediate Java source files generated by GenIC must not be deleted. If omitted, it defaults to `false`.	No
nofastrmic	If `true`, the external RMIC compiler is used by GenIC. The default is `false`, which means the internal fast RMIC compiler is used.	No
nocompil	`true` if the generated source files must not be compiled via the Java and RMI compilers. If omitted, it defaults to `false`.	No
validation	`true` if the XML deployment descriptors must be parsed with validation. If omitted, it defaults to `true`.	No
javac	Java compiler to use. If omitted, it defaults to the value of `build.compiler` property.	No
javacopts	Options to pass to the Java compiler.	No
protocols	Comma-separated list of protocols (chosen from jeremie, jrmp, iiop, cmi) for which stubs should be generated. Default is `jrmp,jeremie`.	No
rmicopts	Options to pass to the RMI compiler.	No
verbose	Indicates whether or not to use -verbose switch. If omitted, it defaults to `false`.	No
additionalargs	Add additional arguments to GenIC.	No
jvmopts	Additional arguments to pass to the GenIC JVM.	No
jvmdebug	Indicates whether you want to debug the forked JVM; it defaults to `false`. The JVM will be suspended and waiting a connection on port 12345.	No
invokecmd	If this attribute is set to `true`, GenIC uses the Javac Sun class to avoid using the 'javac' command line. This is useful for users getting 'java.io.Exception CreateProcess' errors because of too long command lines. Defaults to `false`.	No

Nested Element	Description	Required

Nested Element	Description	Required
classpath	The additional classpath entries used when generating EJB stubs and skeletons.	No
fileset	Points out the ejb-jars that will be processed by GenIC. Note that you can add as many filesets as you want (useful if your JARs are in different directories).	Yes (at least 1)

1. First, define this new task:

```
<taskdef name="genic"
    classname="org.objectweb.jonas.ant.GenICTask"
    classpath="${jonas.root}/lib/common/ow_jonas_ant.jar" />
```

2. Then use it:

  a. Create the JAR file.
```
<jar destfile="${temp.ejbjars.dir}/my-ejbjar.jar">
    <metainf dir="${etc.dir}/META-INF" />
    <fileset dir="${classes.dir}">
        <include name="org/objectweb/jonas/myejbjar/*.class" />
    </fileset>
</jar>
```

  b. Process this JAR with the GenIC task.
```
<genic keepgenerated="true"
        protocols="${protocols.names}">
    <fileset dir="${temp.dir}">
        <include name="ejbjars/my-ejbjar.jar" />
    </fileset>
</genic>
```

Example 27-1. Example

Chapter 28.

Login Modules in a Java Client

This chapter describes how to configure an environment to use login modules with Java clients, and provides an example of this.

28.1. Configuring an Environment to Use Login Modules with Java Clients

The login modules for use by clients are defined in the `$JONAS_ROOT/conf/jaas.config` file. For example:

```
jaasclient {
    // Login Module to use for the example jaasclient.

    //First, use a LoginModule for the authentication
    // Use the resource memrlm_1
    org.objectweb.jonas.security.auth.spi.JResourceLoginModule required
    resourceName="memrlm_1"
        ;

    // Use the login module to propagate security to the JOnAS server
    // globalCtx is set to true in order to set the security context
    // for all the threads of the client container instead of only
    // on the current thread.
    // Useful with multithread applications (like Swing Clients)
    org.objectweb.jonas.security.auth.spi.ClientLoginModule  required
    globalCtx="true"
        ;
};
```

This file is used when a Java client is launched with `jclient`, as a result of the following property being set by `jclient`:

```
-Djava.security.auth.login.config==$JONAS_ROOT/conf/jaas.config
```

For more information about the JAAS authentication, refer to the JAAS authentication tutorial (see http://java.sun.com/j2se/1.4.1/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html).

28.2. Example of a Client

- First, declare the CallbackHandler to use. It can be a simple command line prompt, a dialog, or even a login/password to use.

  Example of CallbackHandler that can be used within JOnAS:
  ```
  CallbackHandler handler1 = new LoginCallbackHandler();
  CallbackHandler handler2 = new DialogCallbackHandler();
  CallbackHandler handler3 =
    new NoInputCallbackHandler("jonas_user", "jonas_password");
  ```
- Next, call the LoginContext method with the previously defined CallbackHandler and the entry to use from the `JONAS_ROOT/conf/jaas.config` file.

  This example uses the dialog callbackhandler.

```
LoginContext loginContext = new LoginContext("jaasclient", handler2);
```

- Finally, call the login method on the LoginContext instance.

```
loginContext.login();
```

If there are no exceptions, the authentication is successful.

If the supplied password is incorrect, authentication can fail.

Chapter 29.

Web Services with JOnAS

Web Services can be used within EJBs or servlets/JSPs. This integration conforms to the JSR 921(Web Service for J2EE v1.1) specification (see http://www.jcp.org/en/jsr/details?id=921).

> **Note**
>
> While JSR 921 is in a maintenance review phase, we provide a link to the 109(v1.0) specification (http://www.jcp.org/en/jsr/details?id=109).

29.1. Web Services

29.1.1. Some Definitions

WSDL (Web Service Description Language v1.1)

An XML-based format for specifying the interface to a Web Service. The WSDL details the service's available methods and parameter types, as well as the actual SOAP endpoint for the service. In essence, WSDL is the "user's manual" for the Web Service. (See http://www.w3.org/TR/wsdl.)

SOAP (Simple Object Access Protocol v1.2)

An XML-based protocol for sending request and responses to and from Web Services. It consists of three parts: an envelope defining message contents and processing, encoding rules for application-defined data types, and a convention for representing remote procedure calls and responses. (See http://www.w3.org/tr/SOAP.)

JAX-RPC (Java API for XML RPC v1.1)

The Java API for XML based RPC. RPC (Remote Procedure Call) enables a client to execute procedures on other systems. The RPC mechanism is often used in a distributed client/server model in which the server defines a service as a collection of procedures that may be called by remote clients. In the context of Web Services, RPCs are represented by the XML-based protocol SOAP when transmitted across systems.

In addition to defining the envelope structure and encoding rules, the SOAP specification defines a convention for representing remote procedure calls and responses. An XML-based RPC server application can define, describe and export a Web Service as an RPC-based service. WSDL (Web Service Description Language) specifies an XML format for describing a service as a set of endpoints operating on messages. With the JAX-RPC API, developers can implement clients and services described by WSDL. (See http://www.jcp.org/en/jsr/detail?id=101.)

29.1.2. Overview of a Web Service

Strictly speaking, a Web Service is a well-defined, modular, encapsulated function used for loosely coupled integration between applications' or systems' components. It is based on standard technologies, such as XML, SOAP, and UDDI.

Web Services are generally exposed and discovered through a standard registry service. With these standards, Web Services consumers (whether they be users or other applications) can access a broad range of information—personal financial data, news, weather, and enterprise documents—through applications that reside on servers throughout the network.

Web Services use a WSDL Definition as a contract between client and server (which are called *endpoints*). WSDL defines the types to serialize through the network (described with XMLSchema), the messages to send and receive (composition, parameters), the portTypes (abstract view of a Port), the bindings (concrete description of PortType: SOAP, GET, POST, ...), the services (set of Ports), and the Port (the port is associated with a unique endpoint URL that defines the location of the Web Service).

A Web Service for J2EE is a component with some methods exposed and accessible by HTTP (through servlets). Web Services can be implemented as Stateless Session Beans or as JAX-RPC classes (a simple Java class, no inheritance needed).
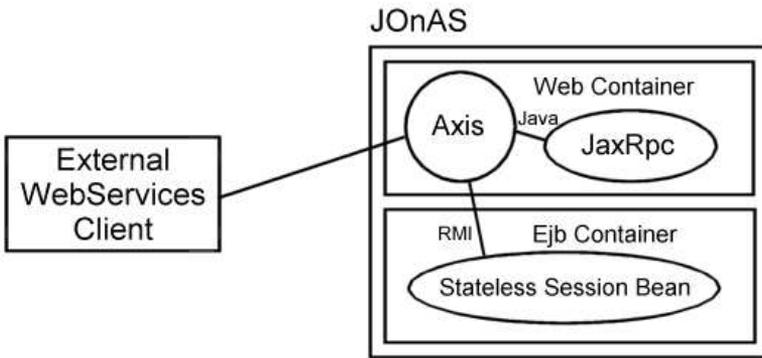
Figure 29-1. Web Services endpoints deployed within JOnAS (an external client code can access the endpoint via AxisServlet)
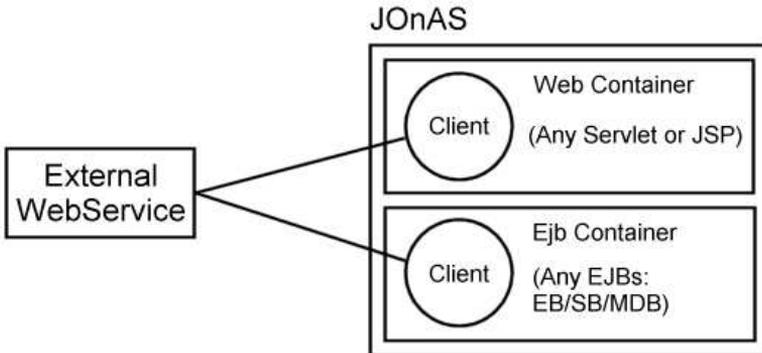
Figure 29-2. Web Services client deployed within JOnAS (can access external Web Services)

The servlet is used to respond to a client request and dispatch the call to the designated instance of servant (the SSB or JAX-RPC class exposed as Web Service). It handles the deserialization of incoming SOAP message to transform SOAP XML into a Java Object, perform the call, and serialize the call result (or the thrown exception) into SOAP XML before send the response message to the client.

29.2. Exposing a J2EE Component as a Web Service

There are two types of J2EE components that can be exposed as Web Services endpoints: Stateless Session Beans and JAX-RPC classes. Web Services' endpoints must not contain state information.

A new standard Deployment Descriptor has been created to describe Web Services endpoints. This Descriptor is named `webservices.xml` and can be used in a Web Application (in WEB-INF/) or in an EJB-JAR (in META-INF/). This Descriptor has its JOnAS-specific Deployment Descriptor (`jonas-webservices.xml` is optional).

29.2.1. A JAX-RPC Endpoint

A JAX-RPC endpoint is a simple class running in the Tomcat servlet container. SOAP requests are dispatched to an instance of this class and the response is serialized and sent back to the client.

A JAX-RPC endpoint must be in a Web Application (the WAR file must contain a `WEB-INF/webservices.xml`).

29.2.2. Stateless Session Bean Endpoint

An Stateless Session Bean (SSB) is an EJB that will be exposed (all or some of its methods) as a Web Service endpoint.

In the `ejb-jar.xml` standard descriptor, a Session Bean, exposed as a Web Service, must now use the new service-endpoint tag. Here the developer defines the fully qualified interface name of the Web Service. Notice that no other interfaces (home, remote, localhome, local) are needed with a Session Bean exposed as Web Service.

Typically, an SSB must be in an EJB-JAR, and a `META-INF/webservices.xml` is located in the EJB-JAR file.

29.2.3. Usage

In this Descriptor, the developer describes the components that will be exposed as Web Services' endpoints; these are called the port-components. A set of port-components defines a webservice-description, and a webservice-description uses a WSDL Definition file for a complete description of the Web Services' endpoints.

Each port-component is linked to the J2EE component that will respond to the request (service-impl-bean with a servlet-link or ejb-link child element) and to a WSDL port (wsdl-port defining the port's QName). A list of JAX-RPC Handlers is provided for each port-component. The optional service-endpoint-interface defines the methods of the J2EE components that will be exposed (no inheritance needed).

A JAX-RPC Handler is a class used to read or modify the SOAP Message before transmission or after reception (refer to the *JAX-RPC v1.1 spec. chap#12 "SOAP Message Handlers"*). The Session Handler is a simple example that will read/write SOAP session information in the SOAP Headers. Handlers are identified with a unique name (within the application unit), are initialized with the init-

param(s), and work on processing a list of SOAP Headers defined with soap-headers child elements. The Handler is run as the SOAP actors defined in the list of SOAP-roles.

A webservice-description defines a set of port-components, a WSDL Definition (describing the Web Service) and a mapping file (WSDL-2-Java bindings). The wsdl-file element and the jaxrpc-mapping-file element must specify a path to a file contained in the module unit (that is, the WAR/JAR file). Note that a URL cannot be set here. The specification also requires that the WSDLs be placed in a wsdl subdirectory (that is, WEB-INF/wsdl or META-INF/wsdl); there is no such requirement for the jaxrpc-mapping-file. All the ports defined in the WSDL must be linked to a port-component. This is essential because the WSDL is a contract between the endpoint and a client (if the client use a port not implemented/linked with a component, the client call will systematically fail).

As for all other Deployment Descriptors, a standard XML Schema is used to constrain the XML.

29.2.4. Simple Example (expose a JAX-RPC Endpoint) of webservices.xml

```
<webservices 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://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
   version="1.1">
   <display-name>Simple Web Service Endpoint
      DeploymentDesc</display-name>

  <webservice-description>
    <!-- name must be unique in an Application unit-->
    <!-- Should not contains spaces !!! -->
    <webservice-description-name>
      SimpleWebServiceEndpoint
    </webservice-description-name>

    <!-- Link to the WSDL file describing the endpoint -->

<wsdl-file>WEB-INF/wsdl/warendpoint.wsdl</wsdl-file>

    <!-- Link to the mapping file describing binding
        between WSDL and Java -->

<jaxrpc-mapping-file>WEB-INF/warEndpointMapping.xml
</jaxrpc-mapping-file>

    <!-- The list of endpoints -->
    <port-component>

    <!-- Unique name of the port-component -->
    <!-- Must not contains spaces !!! -->

<port-component-name>WebappPortComp1</port-component-name>

      <!-- The QName of the WSDL Port the
J2EE port-component is linked to -->
      <!-- Must Refers to a port in
associated WSDL document -->
      <wsdl-port xmlns:ns="
      http://wsendpoint.servlets.ws.objectweb.org ">
         ns:wsendpoint1
      </wsdl-port>

      <!-- The endpoint interface defining methods exposed -->
      <!-- for the endpoint                              -->
```

```
    <service-endpoint-interface>
       org.objectweb.ws.servlets.wsendpoint.WSEndpointSei
    </service-endpoint-interface>

    <!-- Link to the J2EE component (servlet/EJB)    -->
    <!-- implementing methods of the SEI            -->
    <service-impl-bean>
    <!-- name of the servlet defining the JAX-RPC endpoint   -->
    <!-- can be ejb-link if SSB is used (only in EjbJar !)   -->
          <servlet-link>WSEndpoint</servlet-link>
    </service-impl-bean>

    <!-- The list of optional JAX-RPC Handlers -->
    <handler>
    <handler-name>MyHandler</handler-name>
    <handler-class>org.objectweb.ws.handlers.SimpleHandler</handler-class>
    <!-- A list of init-param for Handler configuration -->
       <init-param>
          <param-name>param1</param-name>
          <param-value>value1</param-value>
       </init-param>
       <init-param>
          <param-name>param2</param-name>
          <param-value>value2</param-value>
       </init-param>
    </handler>
   </port-component>
  </webservice-description>
</webservices>
```

29.2.5. The Optional jonas-webservices.xml

The `jonas-webservices.xml` file is an optional deployment descriptor that is used only for state-less session bean endpoints. This descriptor is used by WsGen to set some specifics for the generated webapp that is in charge of the SOAP request dispatching. Its role is to allow for the specifications of the webapp's name, context root, security settings, and endpoint URL customization.

War Name:

Specifies the name of the generated WAR. Setting the <war> element is optional; if it is not specified, then the default naming convention is used to retrieve the WAR name from the EJB-JAR's name.

Default Naming:

<*ejbjar-name*>.jar will have an <*ejbjar-name*>.war WebApp.

Context Root:

Specifies the context root of the generated WAR. Setting the <context-root> element is optional; if it is not specified, then the default naming convention is used and the context root will be the same as the WAR's name.

Default Naming:

<*webapp-name*>.war will have a <*webapp-name*> context root.

Specifies the endpoint URL of a given jonas-port-component. Setting the <endpoint-uri> element is optional; if it is not specified, the default naming convention is used and URL will looks like: /<context-root>/<port-component-name>/<wsdl:port-name>

Security:

The security settings specify what security settings, if any, the generated webapp should use. Setting up security within this file is very similar to how you would setup security in a normal web.xml.

> **Note**
>
> When using the <endpoint-security-constraint>, <endpoint-login-config>, or <endpoint-security-role> elements, you must include the j2ee namespace in the jonas-webservices element (that is, xmlns:j2ee:"http://java.sun.com/xml/j2ee"), and you will have to specify the namespace for all elements within these tags (that is, the 'j2ee:' in the examples below).

Security Constraint:

The <endpoint-security-constraint> specifies the <security-constraint> element to be used in the web.xml of the generated webapp.

Login Config:

The <endpoint-login-config> is used to specify the <login-config> element to be used in the web.xml of the generated webapp.

Security Role:

The <endpoint-security-role> specifies the <security-role> element to be used in the web.xml of the generated webapp.

Realm:

The <endpoint-realm> specifies the security realm the generated webapp should use.

Realm Name:

The <endpoint-realm-name> specifies the name of the security realm the generated webapp should use.

```
<?xml version="1.0"?>
<jonas-webservices xmlns="http://www.objectweb.org/jonas/ns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:j2ee:"http://java.sun.com/xml/ns/j2ee"
    xsi:schemaLocation="http://www.objectweb.org/jonas/ns
    http://www.objectweb.org/jonas/ns/jonas_j2ee_web_services_4_5.xsd">

  <!-- the name of the webapp in the EAR            -->
  <!-- (it is the same as the one in application.xml) -->
  <war>dispatchingWebApp.war</war>

  <!-- the context root the webapp should use -->
  <context-root>endpoint</context-root>

  <jonas-port-component>
```

```
    <port-component-name>ssbEndpoint</port-component-name>
    <!-- We can specify with endpoint-uri the URL we want to have  -->
    <!-- for the given port. -->
    <!-- Will be something like: -->
    <!--
http://<host>:<port>/<context-root>/services1/webservice -
->
    <endpoint-uri>/services1/webservice</endpoint-uri>
  </jonas-port-component>

  <!-- everthing in this element will appear in the
security-constraint -->
  <!-- element in the web.xml of the generated
        -->
  <endpoint-security-constraint>
    <j2ee:web-resource-collection>
      <j2ee:web-resource-name>Stateless Session Bean
Endpoint</j2ee:web-resource-name>
      <j2ee:url-pattern>/*</j2ee:url-pattern>
    </j2ee:web-resource-collection>
    <j2ee:auth-constraint>
      <j2ee:role-name>admin</j2ee:role-name>
    </j2ee:auth-constraint>
  </endpoint-security-constraint>

  <!-- everything in this element will appear in the login-config -->
  <!-- element in the web.xml of the generated webapp         -->
  <endpoint-login-config>
    <j2ee:auth-method>BASIC</j2ee:auth-method>
    <j2ee:realm-name>Endpoint Authentication
Area</j2ee:realm-name>
  </endpoint-login-config>

  <!-- everything in this element will appear in the security-role -->
  <!-- element in the web.xml of the generated webapp         -->
  <endpoint-security-role>
    <j2ee:role-name>admin</j2ee:role-name>
  </endpoint-security-role>
  <!-- the realm the generated webapp should use -->
  <endpoint-realm>memrlm_1</endpoint-realm>
  <!-- the name of the realm the generated webapp should use -->
  <endpoint-realm-name>Endpoint Authentication
Area</endpoint-realm-name>
</jonas-webservices>
```

Example 29-1. Example:

29.2.6. Changes to jonas-web.xml

JOnAS allows the developer to fully configure an application by setting the hostname, the context-root, and the port used to access the Web Services. This is done in the `jonas-web.xml` file of the dispatching Web Application.

host

Configure the hostname to use in URL (must be an available web container host).

port

Configure the port number to use in URL (must be an available HTTP/HTTPS connector port number).

When these values are not set, JOnAS will attempt to determine the default values for host and port.

Limitations:

- The host can be determined only when only one host is set for the web container.
- The port can be determined only when only one connector is used by the web container.

29.3. The Web Services Client

An EJB or a servlet that wants to use Web Services (as a client) must declare a dependency on the Web Service with a service-ref element (same principle as for all *-ref elements).

29.3.1. The service-ref Element

The service-ref element declares a reference to a Web Service used by a J2EE component in the web, EJB, and client Deployment Descriptor.

The component uses a logical name called a service-ref-name to look up the service instance. Thus, any component that performs a lookup on a Web Service must declare a dependency (the service-ref element) in the standard deployment descriptor (`web.xml` `application-client.xml`, or `ejb-jar.xml`).

Example of service-ref:

```
<service-ref>
  <!-- (Optional) A Web Services description that can be used
    in administration tool. -->
  <description>Sample WebService Client</description>

  <!-- (Optional) The WebService reference name
    (used in Administration tools) -->
  <display-name>WebService Client 1</display-name>

  <!-- (Optional) An icon for this WebService. -->
  <icon> <!-- ... --> </icon>

  <!-- The logical name for the reference that is used
    in the client source code. It is recommended, but not required,
    that the name begin with 'services/' -->
  <service-ref-name>services/myService</service-ref-name>

  <!-- Defines the class name of the JAX-RPC Service interface
    that the client depends on. In most cases, the value
    will be javax.xml.rpc.Service, but a generated specific Service
    Interface class may be specified (requires WSDL knowledge and
    so on for the wsdl-file element). -->
  <service-interface>javax.xml.rpc.Service</service-interface>

  <!-- (Optional) Contains the location (relative to the root of
    the module) of the Web Service WSDL description.
      -needs to be in the wsdl directory
      -required if generated interface and sei are declared. -->
  <wsdl-file>WEB-INF/wsdl/stockQuote.wsdl</wsdl-file>
```

```
<!-- (Optional) A file specifying the correlation of the WSDL
  definition to the interfaces (Service Endpoint Interface,
  Service Interface).
    -required if generated interface and sei (Service Endpoint
     Interface) are declared.-->
<jaxrpc-mapping-file>WEB-INF/myMapping.xml</jaxrpc-mapping-file>

<!-- (Optional) Declares the specific WSDL service element
  that is being referred to. It is not specified if no wsdl-file
  is declared or if WSDL contains only 1 service element.
  A service-qname is composed by a namespaceURI and a localpart.
  You must define it if more than 1 service is declared
  in the WSDL. -->
<service-qname>
  <namespaceURI>http://beans.ws.objectweb.org</namespaceURI>
  <localpart>MyWSDLService</localpart>
</service-qname>

<!-- Declares a client dependency on the container to resolving
  a Service Endpoint Interface to a WSDL port. It optionally
  associates the Service Endpoint Interface with a particular
  port-component. -->
<port-component-ref>
  <service-endpoint-interface>
    org.objectweb.ws.beans.ssbendpoint.MyService
  </service-endpoint-interface>
  <!-- Define a link to a port component declared in another unit
  of the application -->
  <port-component-link>
    ejb_module.jar#PortComponentName
  </port-component-link>
</port-component-ref>


<!--A list of Handler to use for this service-ref  -->
<handler>

  <!-- Must be unique within the module. -->
  <handler-name>MyHandler</handler-name>

  <handler-class>org.objectweb.ws.handlers.myHandler</handler-class>

  <!-- A list of init-param (couple name/value) for Handler
    initialization -->
  <init-param>

    <param-name>param_1</param-name>
    <param-value>value_1</param-value>
  </init-param>

  <!-- A list of QName specifying the SOAP Headers the handler
    will work on.
    -namespace and localpart values must be found inside the WSDL. -->
  <soap-header>
    <namespaceURI>http://ws.objectweb.org</namespaceURI>
    <localpart>MyWSDLHeader</localpart>
  </soap-header>

  <!-- A list of SOAP actor definition that the Handler will play
    as a role. A soap-role is a namespace URI. -->
  <soap-role>http://actor.soap.objectweb.org</soap-role>
```

```
    <!-- A list of port-name element defines the WSDL port-name
      that a handler should be associated with. If no port-name is
      specified, the handler is assumed to be associated with all ports
      of the service-ref. -->
    <port-name>myWSDLPort</port-name>
  </handler>
</service-ref>
```

29.3.2. The jonas-service-ref Element

A jonas-service-ref must be specified for each service-ref declared in the standard Deployment Descriptor. The jonas-service-ref adds JOnAS-specific (and Web Service engine-specific) information to service-ref elements.

Example of jonas-service-ref:

```
<jonas-service-ref>
   <!-- Define the service-ref contained in the component
      deployment descriptor (web.xml, ejb-jar.xml, or
      application-client.xml) used as a key to associate a
      service-ref to its corresponding jonas-service-ref-->
   <service-ref-name>services/myService</service-ref-name>

   <!-- Define the physical name of the resource. -->
   <jndi-name>webservice_1</jndi-name>

   <!-- A list of init-param used for specific configuration of
      the service -->
   <jonas-init-param>
      <param-name>param</param-name>
      <param-value>name</param-value>
   </jonas-init-param>
</jonas-service-ref>
```

29.4. WsGen

WsGen is a new JOnAS tool that works in the same way as GenIC. It takes an archive file (EJB-JAR, WAR, JAR client, or EAR) and generates all the necessary Web Services files:

- Creates vendor-specific Web Services deployment files for the server side and, when needed, the client side (Axis will use its own WSDD format).
- Creates a Web Application for each EJB-JAR exposing Web Service.
- Generates and compiles client-side artifacts (Services and Port Bindings implementation classes).

For example, to provide an EJB-exposing method as a Web Service, a developer creates a webservices.xml file packaged in EJB-JAR's META-INF directory. WsGen automatically creates a configured Web Application (using an Axis servlet) and wraps it (ejbjar + webapp) in an EAR file.

With a JaxRpc class, WsGen adds a servlet (an Axis servlet) inside the existing web deployment descriptor and generates an Axis-specific configuration file.

When using service-ref (from EJB-JARs, Web Applications, or clients), WsGen automatically generates a stub from WSDL (if a generated service interface name is provided).

29.4.1. Usage

WsGen is used usually from an ant build file. Simply add this taskdef under the EJB-JAR taskdef:

```
<taskdef name="wsgen" classname="org.objectweb.jonas.ant.WsGenTask"
   classpath="${jonas.root}/lib/common/ow_jonas_ant.jar" />
<wsgen srcdir="${temp.dir}"
   destdir="${dist.dir}"
   verbose="false"
   debug="false">
      <include name="webapps/wswarsample.war"/>
</wsgen>
```

See the `$JONAS_ROOT/examples/webservices` samples for complete build scripts.

Note that the EJB-JAR/Web Application/client archive must include WSDL, jax-rpc-mappings used in service-ref, or `webservices.xml`. When these files are used from a service-ref, they are added into the generic EJB-JAR with the EJB-JAR Ant task of JOnAS; you must ensure that they have been placed inside the srcdir given to the EJB-JAR task (otherwise, the EJB-JAR task cannot find them and will produce an error).

This task is a directory-based task and, as such, forms an implicit fileset (see http://ant.apache.org/manual/index.html. This defines which files, relative to the `srcdir`, will be processed. The WsGen task supports all the attributes of fileset to refine the set of files to be included in the implicit fileset.

Attribute	Description	Required
srcdir	Directory where file archive (EJB-JAR, WAR, Client, EAR) can be found.	Yes
destdir	Directory where generated files will be placed	No
verbose	Verbose mode (Defaults to false)	No
debug	Debug mode (Defaults to false)	No
javacopts	List of options given to the Java compiler	No
jonasroot	Directory where JOnAS is installed	No
jonasbase	Directory where JOnAS configuration is stored	No

Table 29-1. wsgen Task Support

Wsgen is also usable from the command line with WsGen script (available on *nix and Windows).

VII. How-to Documents

This section contains information for a variety of JOnAS users.

Table of Contents

Chapter 30.

JOnAS Migration

The guide contains the following:

30.1. JOnAS 4.3.x to JOnAS 4.4.1

Applications developed for JOnAS 4.3.x do not require changes; The main changes occur within the JOnAS configuration files, and it is recommended that you report your customizations in the new JOnAS 4.4.1 configuration files, especially for the ones mentioned below.

30.1.1. Application redeployment

This JOnAS's version offers a new mechanism to ensure application redeployment during the migration. At the deployment time, GenIC is automatically launched by the server whether a version change is detected.

30.1.2. Configuration changes

The most visible configuration change is Tomcat 5.0 moves to Tomcat 5.5. See the wiki about Tomcat https://wiki.objectweb.org/jonas/Wiki.jsp?page=Tomcat-To-JOnAS.

30.1.3. Update your $JONAS_BASE

- If starting from an existing JONAS_BASE, it must be updated in order to upgrade to the last built-in provided EAR/WAR/JAR/RAR files (for example, new versions of the JORAM or JDBC RARs).
  ```
  cd $JONAS_ROOT
  ant update_jonasbase
  ```
- mejb is packaged as an ear in the new version while it was delivered in an ejbjar in the previous version. So the old ejbjar must be deleted.
  ```
  cd $JONAS_BASE/ejbjars/autoload
  rm mejb.jar
  ```

30.1.4. JORAM 4.3.0

JOnAS 4.4.1 integrates the new JORAM 4.3.0 which contains:

- Performance optimizations
- New feature: SSL for client-server communications
- New feature: SoftReference Messages allowing swap out, the ability to send large messages, the destinations can store more and more messages, and the messages are not held in memory.

The latest implementation required changes to the message persistance module. Therefore, JORAM's stored messages from the previous versions are not compliant with this new version; you must delete JORAM's persistance directory (s0 by default) before the migration.

30.2. JOnAS 4.1 to JOnAS 4.3.x

Applications developed for JOnAS 4.1 do not require changes; however, they should be redeployed (GenIC). The main changes occur within the JOnAS configuration files, and it is recommended that you document your customizations in the new JOnAS 4.3.2 configuration files, especially for the files mentioned below.

30.2.1. Configuration changes

The most visible configuration changes are the following:

1. By default, JORAM uses RAR instead of the JMS service

2. JOnAS no longer uses the JRE ORB implementation; it uses the JacORB implementation. The default iiop model now used is the POA model (http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/rmiiiopexample.html). Thus, GenIC should be relaunched on all previously generated beans.

Configuration files with significant changes:

`conf/jonas.properties`

The jms service is removed from the `jonas.services` property.

`conf/joram-admin.cfg`

This file is used for specifying the creation of JMS-administered objects when using the JORAM connector. The JMS destinations previously defined in the `jonas.properties` file (jonas.service.jms.queues and jonas.service.jms.topics) must be moved into this file.

`$JONAS_BASE/conf/jacorb.properties`

This is the configuration file for JacORB.

30.2.2. Update your $JONAS_BASE

- If starting from an existing JONAS_BASE, it must be updated in order to upgrade to the last built-in provided EAR/WAR/JAR/RAR files (for example, new versions of the JORAM or JDBC RARs).
  ```
  cd $JONAS_ROOT
  ant update_jonasbase
  ```
- If you want to keep the jms service and not use the JORAM's RAR, you have to remove the `$JONAS_BASE/rars/autoload/joram_for_jonas_ra.rar`.

30.3. JOnAS 3.3.x to JOnAS 4.1

Applications developed for JOnAS 3.3.x do not require changes; however, they should be redeployed (GenIC). The main changes occur within the JOnAS configuration files, and it is recommended to report your customizations in the new JOnAS 4.1 configuration files, especially for the ones mentioned below.

30.3.1. Configuration changes

The two most visible configuration changes are the following:

1. HTTP port numbers have moved from the 8000 range to the 9000 range. For example, the JOnAS server index page with default configuration on a given host is now http://localhost:9000/index.jsp

2. The three RMI communication protocols, jrmp, jeremie, and iiop can now be used simultaneously; the incompatibility between Jeremie and rmi/iiop and the "ant installiiop" step have been suppressed. In any case, the "ant install" phase (in JONAS_ROOT) is no longer needed.

Configuration files with significant changes:

`conf/server.xml`

> This file is a customized Tomcat 5 configuration file, while in JOnAS 3.3.x it was a Tomcat 4 configuration file. Moreover, package names of JOnAS-related security files have changed. For example, org.objectweb.jonas.security.realm.web.catalina50.JACC replaces org.objectweb.jonas.security.realm.JRealmCatalina41. The JAAS classname realm is org.objectweb.jonas.security.realm.web.catalina50.JAAS.

`conf/jetty5.xml`

> Replaces `conf/jetty.xml`. In the `web-jetty.xml` files (in the WAR), the package name of the Realm class has changed, for example, org.objectweb.jonas.security.realm.web.jetty50.Standard replaces org.objectweb.jonas.security.realm.JRealmJetty42 class. The JAAS classname realm is org.objectweb.jonas.security.realm.web.jetty50.JAAS.

`conf/jonas.properties`

> There are many changes:
>
> - Some properties for web services
> - Some package names have changed (for example, for the Web JOnAS service)
> - The XML validation is activated by default for EJBs
> - New properties for the service 'db' (by default it uses HSQL as java database)

`conf/joram-admin.cfg`

> This is a new configuration file used for specifying the creation of JMS- administered objects when using the JORAM connector (J2EE CA 1.5 JMS resource adapter). The default file corresponds to the default-administered objects created when using the JOnAS JMS service.

30.3.2. Running EJB 2.1 Message-driven Beans

The use of EJB 2.1 Message-driven beans (MDBs) requires changing the JOnAS configuration. While for EJB 2.0 MDBs the JOnAS JMS service was required, EJB 2.1 MDBs can only be used through a JMS Connector (J2EE CA 1.5 resource adapter). Currently the JOnAS JMS service and the JMS connector cannot work at the same time. Therefore, it is necessary to suppress the "jms" service from the list of JOnAS services (jonas.services in jonas.properties) and to add the JORAM connector in the list of resource adapters to be deployed by the JOnAS resource service (jonas.service.resource.resources in jonas.properties). Note that it is currently not possible to run EJB 2.0 MDBs and EJB 2.1 MDBs simultaneously in the same server. It is anticipated that a JMS connector able to handle both EJB 2.0 and EJB 2.1 MDBs will be available soon, at which time the JOnAS JMS service will become deprecated. For more details, refer to Section 3.5.10 *Configuring the JMS Service* and Section 3.8 *Configuring JMS Resource Adapters*.

30.3.3. Deploying Resource Adapters

The Resource Adapter configuration tool (Section 6.7 *RAConfig*) did not generate the DOCTYPE information in JOnAS 3.3.x versions. If you are using resource adapters that were customized through RAConfig, it is recommended that the tool be run again on these Resource Adapters.

30.4. JOnAS 3.1 to JOnAS 3.1.4

Applications developed for JOnAS 3.1 do not require changes; however, they should be redeployed (GenIC). The migration affects only certain customized configuration files and `build.xml` files.

The main changes are in the area of *communication protocols* support, due to the integration of CAROL. This implies the following configuration changes:

- The `jndi.properties` file is replaced by a `carol.properties` file (in `JONAS_BASE/conf` or `JONAS_ROOT/conf`) and is no longer searched for within the classpath.
- The OBJECTWEB_ORB environment variable no longer exists.
- Security context propagation is specified in the `jonas.properties` file, which replaces the `-secpropag` option of GenIC or the `secpropag` attribute of the JOnAS EJB-JAR ANT task.
- EJBs can be deployed for several protocols, which is specified by the new option `-protocols` of GenIC or new attribute `protocols` of the JOnAS ejb-jar ANT task; previously, the protocol was chosen through the OBJECTWEB_ORB environment variable.
- The `${OBJECTWEB_ORB}_jonas.jar` files, that is, `RMI_jonas.jar` or `JEREMIE_jonas.jar`, no longer exist; there is only one `jonas.jar` file.
- The previous items involve changes in application `build.xml` files.

Refer to Section 3.3 *Configuring the Communication Protocol and JNDI* for details about communication protocol configuration.

Other configuration changes are due to *security* enhancements:

- The files `tomcat-users.xml`, `jonas-users.properties`, and `jettyRealm.properties` are suppressed and replaced by a `jonas-realm.xml` file. This file contains the list of users/password/roles for the Memory realm, as well as the access configuration for Database and LDAP realms. Realms declared in this file have corresponding resources bound in the registry, and MBeans to be managed.
- The security service should be launched after the dbm service (This is set be the order in which they appear in the jonas.services property).
- A new realm with a reference to the JOnAS resource specified in the `jonas-realm.xml` file is used in the `server.xml` file (Tomcat) or in the `web-jetty.xml` file (Jetty).
- The `jonas.properties` file contains a new line specifying the jndi name of a resource (ejbrealm) that provides Java access to the user identification repository (memory, ldap, or database) of the corresponding realm (specified in the `jonas-realm.xml` file). This is primarily used by Java clients that intend to build their SecurityContext.

Refer to Section 3.5.9 *Configuring the Security Service* for details about security configuration.

The *preferred steps* for migrating from JOnAS 3.1 are the following:

1. Create a new JOnAS_BASE (for example, through the ANT `create_jonasbase` target).
2. Copy the new as well as any customized files from the old JONAS_BASE to the new one, conforming to the new configuration rules (`jndi.properties` replaced by `carol.properties`, security-context propagation and realm specified in `jonas.properties`, new realm specification in `server.xml`, changes in your `build.xml` files, and the content of `tomcat-users.xml`, `jonas-users.properties`, or `jettyRealm.properties` should migrate into `jonas-realm.xml`).

Details for migrating a configuration are provided in the following sections.

30.4.1. carol.properties

Modify this file according to the content of the old `jndi.properties` file. If the OBJECTWEB_ORB was RMI, set `carol.protocols` to `jrmp`; if the OBJECTWEB_ORB was JEREMIE, set `carol.protocols` to `jeremie`. Then, configure the URL with host name and port number. Example:

```
carol.protocols=jrmp
carol.jrmp.url=rmi://localhost:1099
```

30.4.2. jonas.properties

If EJB security was used, the security context propagation should be activated. A realm resource can be chosen to be accessed from Java; this is now specified in the `jonas.properties` file:

```
jonas.security.propagation  true
jonas.service.security.ejbrealm memrlm_1
jonas.services   registry,jmx,jtm,dbm,security,jms,ejb,web,ear
```

30.4.3. server.xml

Choose the memory, database, or ldap realm resource for Tomcat authentication.

```
<Realm className="org.objectweb.jonas.security.realm.JRealmCatalina41"
   debug="99" resourceName="memrlm_1"/>
```

30.4.4. web-jetty.xml

This file is located in the WEB-INF directory of a WAR file and contains a reference to the JOnAS Realm to be used for authentication.

```
<Call name="setRealmName">
   <Arg>Example Basic Authentication Area</Arg>
</Call>
<Call name="setRealm">
   <Arg>
      <New class="org.objectweb.jonas.security.realm.JRealmJetty42">
         <Arg>Example Basic Authentication Area</Arg>
         <Arg>memrlm_1</Arg>
      </New>
   </Arg>
</Call>
```

30.4.5. Deployment

For existing scripts that call GenIC for deploying EJBs, the `-secpropag` option no longer exists (security propagation is activated from the jonas.properties file as illustrated previously), and a new option `-protocols` specifies a comma-separated list of protocols (chosen from jeremie, jrmp, iiop, cmi) for which stubs will be generated. The default value is `jrmp,jeremie`.

```
GenIC –protocols jrmp,jeremie,iiop
```

Refer to the following for the deployment Ant task.

30.4.6. build.xml files

The `build.xml` files for building JOnAS examples have been upgraded according to the new configuration scheme. Existing `build.xml` files must be updated the same way:

- `<property name="orb" value="${myenv.OBJECTWEB_ORB}" />` is no longer used and must be suppressed.
- In the target building the classpath, replace `${orb}_jonas.jar` by `jonas.jar`.
- In the jonas deployment task, suppress the attributes `orb="${orb}"` `secpropag="yes,"` and add the attribute `protocols="${protocols.names}."` The `build.properties` file of the JOnAS examples now contains `protocols.names=jrmp,jeremie`.

30.5. JOnAS 3.0 to JOnAS 3.1

Applications developed for JOnAS 3.0 can be redeployed without any changes.

The differences in the execution environment are the following:

- JOnAS is available under three different packagings.
- The location and the JOnAS configuring policy has changed.
- The location and the policy to deploy applications has changed.

JOnAS is still available as a "single ejb container" as before. Additionally, two new packagings as "J2EE server" are available:

- jonas3.1-tomcat4.1.24 package
- jonas3.1-jetty4.2.9 package.

For these two new packagings, it is no longer necessary to set the environment variable CATALINA_HOME or JETTY_HOME. These packagings have JOnAS examples compiled for use with RMI.

The location and the policy for JOnAS configuration has changed:

- Configuration files are located under `$JONAS_ROOT/conf` (in previous versions they were located under `$JONAS_ROOT/config`).
- The old policy used to search for configuration files (working directory, home directory, `$JONAS_ROOT/config`) is no longer used. The new policy is the following:

  If the environment variable JONAS_BASE is set, configuration files are searched for in `$JONAS_BASE/conf`, if not, under `$JONAS_ROOT/conf`.

The location and the policy for deploying applications has changed:

- If the environment variable JONAS_BASE is set, the application to be deployed with `jadmin` or `jonas admin` is searched for in `$JONAS_BASE/(ejbjars|apps|webapps)`, if is not found, the search continues under `$JONAS_ROOT/(ejbjars|apps|webapps)`.

30.6. JOnAS 2.6.4 to JOnAS 3.0

30.6.1. Application with entity beans CMP 1.1

The standard deployment descriptor must be updated for applications deployed in previous versions of JOnAS using entity beans that have container-managed persistence CMP 1.1. For such entity beans, a `<cmp-version>` tag with value `1.x` must be added. For example, a deployment descriptor that looks like the following:

```
<persistence-type>container</persistence-type>
<cmp-field>
   <field-name>fieldOne</field-name>
</cmp-field>
<cmp-field>
   <field-name>fieldTwo</field-name>
</cmp-field>
```

must be changed as follows:

```
<persistence-type>container</persistence-type>
<cmp-version>1.x</cmp-version>
<cmp-field>
   <field-name>fieldOne</field-name>
</cmp-field>
<cmp-field>
   <field-name>fieldTwo</field-name>
</cmp-field>
```

The EJB 2.0 specification states that the default value of the `<cmp-version>` tag is 2.x.

30.7. JOnAS 2.6 to JOnAS 2.6.1

30.7.1. Installation procedure

Before using JOnAS when it is installed on a host, the `ant install` must be run in the JOnAS_ROOT directory to rebuild global libraries based on your environment (RMI/JEREMIE, web environment). This must be run again if the orb (that is, from RMI to JEREMIE) is switched, or if the web environment (that is, switching from CATALINA to JETTY) changes.

30.7.2. EJB container creation

It is still possible to create an EJB container from an EJB deployment descriptor identified by its `xml` file name in JOnAS version 2.6.1 (as required by JOnAS users with versions prior to JOnAS 2.6).

30.8. JOnAS 2.5 to JOnAS 2.6

30.8.1. Use of Make discontinued for building

The `make` tool is no longer used to build JOnAS and the JOnAS examples. *Common makefiles* previously used to build the JOnAS examples via `make` are no longer delivered in `$JONAS_ROOT/gmk`.

Ant is the only build tool used in JOnAS 2.6.

30.8.2. Building with Ant 1.5

In JOnAS 2.6, the required version of Ant is 1.5 (instead of Ant 1.4). In addition, starting with this version, an ant task *ejbjar* for JOnAS is delivered in the JOnAS distribution. Refer to the JOnAS example to see how this new task is used.

30.8.3. New jonas command

A new `jonas` command is delivered that provides the capability to:

- Start a JOnAS server
- Stop a JOnAS server
- Aadministrate a JOnAS server
- Check the JOnAS environment
- Print the JOnAS version.

The `EJBServer`, `JonasAdmin` and `CheckEnv` commands are now deprecated.

Refer to Chapter 6 *JOnAS Command Reference* for details.

30.8.4. EJB container creation

In previous versions of JOnAS, an EJB container could be created from an EJB-JAR file or from an EJB deployment descriptor identified by its XML file name.

In JOnAS version 2.6, an EJB container can only be created from an EJB-JAR file.

This means that an XML file name can no longer be specified as:

- A value of the `jonas.service.ejb.descriptors` jonas property
- An argument of the `-a` option of the `jonas admin` command.

30.8.5. Tomcat 4.0.x Support

Tomcat version 4.0.1 is no longer supported. The current supported versions are Tomcat 4.0.3 and Tomcat 4.0.4.

30.8.6. Tomcat Service and Web Container Service

JOnAS 2.6 is a full-featured J2EE application server, thus Tomcat can be used with JOnAS as the Web container. This functionality is set up via the JOnAS service *web*. In earlier version of JOnAS, a service *tomcat* was provided. This service `tomcat` is now deprecated because it was not compliant with J2EE specification. Note that if the `tomcat` service was used previously and the `web` container service is now being used, the WAR files (and expanded directories) that were deployed here (they are now deployed in `JONAS_ROOT/webapps`) should be suppressed from the `CATALINA_HOME/webapps` directory.

Refer to Chapter 32 *Integrating Tomcat and JOnAS* for details.

30.9. JOnAS 2.4.4 to JOnAS 2.5

30.9.1. Tomcat 4.0.x Support

JOnAS 2.5 supports two versions of Tomcat for the tomcat service: Tomcat 3.3.x (as in JOnAS 2.4.4) and Tomcat 4.0.x. The default support of JOnAS 2.4.4 is for Tomcat 3.3.x, therefore nothing needs to be changed if the tomcat service was being used with JOnAS 2.4.4. For Tomcat 4.0.x, the `jonas.service.tomcat.class` property of the `jonas.properties` file must be set to `org.objectweb.jonas.tomcat.EmbeddedTomcatImpl40`. Refer to Chapter 31 *Integrating Tomcat and JOnAS (for JOnAS version < JOnAS 2.6)* for more details.

30.9.2. trace.properties

Starting   with   JOnAS   2.5,   the   Objectweb   Monolog (http://www.objectweb.org/monolog/doc/index.html)   logging   API   is   used   for JOnAS traces. The JOnAS traces configuration file `JONAS_ROOT/conf/trace.properties` has changed. Refer to Chapter 3 *JOnAS Configuration* for details.

30.10. JOnAS 2.4.3 to JOnAS 2.4.4

The main differences between these two versions are the two new services provided in JOnAS 2.4.4: `registry` and `tomcat`.

jonas services

JOnAS 2.4.3	from JOnAS 2.4.4
jonas.services *jmx,security,jtm,dbm,resource,* *jms,ejb*	jonas.services *registry,tomcat,jmx,security,jtm,dbm,* *resource,jms,ejb*

Registry service configuration

JOnAS 2.4.3	from JOnAS 2.4.4
-	jonas.service.registry.class org.objectweb.jonas.registry.RegistryServiceImpl *jonas.service.registry.mode* *automatic*

Tomcat service configuration

JOnAS 2.4.3	from JOnAS 2.4.4
-	jonas.service.tomcat.class org.objectweb.jonas.tomcat.EmbeddedTomcatImpl33 jonas.service.tomcat.args

30.11. JOnAS 2.3 to JOnAS 2.4

30.11.1. Introduction

This is intended for JOnAS users migrating applications from JOnAS 2.3 to JOnAS 2.4 and later versions. This migration does not affect EJB components' files. However, two configuration files are slightly different: the `jonas.properties` file and the `jonathan.xml` file.

`jonas.properties`

> Due to the new JOnAS architecture regarding services (refer to the advanced topic Chapter 24 *JOnAS Services*), the structure of the properties defined in this file has changed. It is necessary to upgrade a jonas.properties file written for a version 2.x (x<4) to reuse it for JOnAS 2.4.

`jonathan.xml`

> For applications using the JEREMIE distribution mechanism, it is necessary to upgrade this configuration file because JOnAS has embedded a new version of Jonathan.

30.11.2. Upgrading the jonas.properties file

JOnAS EJB servers are configured via the `jonas.properties` file. This configuration file may be located in three different places:

1. `$JONAS_ROOT/config/jonas.properties`

2. `$HOME/jonas.properties`: the home directory

3. `./jonas.properties`: the directory from which the EJB server is launched.

An EJB server reads the three potential files in the order listed, each one possibly overwriting properties defined in a previous file. Therefore, existing `jonas.properties files` from previous JOnAS versions must be upgraded in order to retain the configuration settings, by making the following structural changes:

jonas services (new)

JOnAS 2.4.3	from JOnAS 2.4.4
-	jonas.services *jmx,security,jtm,dbm,resource,jms,ejb*

JMX service configuration

JOnAS 2.4.3	from JOnAS 2.4.4
-	jonas.service.jmx.class *org.objectweb.jonas.jmx.JmxServiceImpl*

JOnAS EJB service configuration (beans to be loaded)

JOnAS 2.4.3	from JOnAS 2.4.4
jonas.beans.descriptors ...	jonas.service.ejb.class *org.objectweb.jonas.container.EJBServiceImpl* jonas.service.ejb.descriptors ...

JOnAS DBM Service Configuration

JOnAS 2.4.3	from JOnAS 2.4.4
jonas.datasources ...	jonas.service.dbm.class *org.objectweb.jonas.dbm.DataBaseServiceImpl* jonas.service.dbm.datasources ...

JOnAS JTM Service Configuration

JOnAS 2.4.3	from JOnAS 2.4.4
jonas.tm.remote *false* jonas.tm.timeout *60*	jonas.service.jtm.class *org.objectweb.jonas.jtm.TransactionServiceImpl* jonas.service.jtm.remote *false* jonas.service.jtm.timeout *60*

JOnAS Security Service Configuration

JOnAS 2.4.3	from JOnAS 2.4.4
-	jonas.service.security.class *org.objectweb.jonas.security.JonasSecurityServiceImpl*

JOnAS JMS Service Configuration

JOnAS 2.4.3	from JOnAS 2.4.4

JOnAS 2.4.3	from JOnAS 2.4.4
jonas.jms.mom *org.objectweb.jonas_jms.JmsAdminForJoram*	jonas.service.jms.class *org.objectweb.jonas.jms.JmsServiceImpl*
jonas.jms.collocated *true*	jonas.service.jms.mom
jonas.jms.url *joram://localhost:16010*	*org.objectweb.jonas_jms.JmsAdminForJoram*
jonas.jms.threadpoolsize *10*	jonas.service.jms.collocated *true*
jonas.jms.topics *sampleTopic*	jonas.service.jms.url
jonas.jms.queues ...	*joram://localhost:16010*
	jonas.service.ejb.mdbthreadpoolsize *10*
	jonas.service.jms.topics *sampleTopic*
	jonas.service.jms.queues ...

JOnAS Resource Service Configuration (Resource Adapters to be installed)

JOnAS 2.4.3	from JOnAS 2.4.4
-	jonas.service.resource.class *org.objectweb.jonas.resource.ResourceServiceImpl*
	jonas.service.resource.resources ...

The main transformation rule is that most of the properties are now part of a JOnAS service. For each service XXX, the class property `jonas.service.XXX.class` containing the name of the service class (all these class properties are set in the `$JONAS_ROOT/config/jonas.properties` file) must be specified, and each additional property *p* related to the service is named *jonas.service.XXX.p*. The list of services to be launched with the server is specified in the `jonas.services` property. These services are EJB (in which are defined the beans to be loaded), JTM (in which are defined the transaction monitor properties), DBM (in which are defined the datasources), SECURITY, JMS (the messaging service), and JMX (a new service for management).

30.11.3. Upgrading the Jonathan configuration file

In the new version of Jonathan, `jonathan.prop` has been replaced by `jonathan.xml`.

Integrating Tomcat and JOnAS (for JOnAS version $<$ JOnAS 2.6)

This guide describes how to integrate JOnAS with Tomcat from Apache (http://jakarta.apache.org/tomcat/index.html). JOnAS will be used to run your beans inside EJB containers while Tomcat will be used to run your servlets or JSPs. Inside these servlets or JSPs, you can remotely access beans inside the JOnAS EJBServer.

For cases where you need security, Tomcat will be used also for user authentication. Refer to Chapter 33 *Example for Using Security in JOnAS with Tomcat* for more information on how to use security with JOnAS and Tomcat.

The following steps describe how to configure JOnAS and Tomcat to make them work together.

1. Section 31.1 *Installing Tomcat*

2. Section 31.2 *Configuring Tomcat for JOnAS*

3. Section 31.3 *Structuring your Web Application*

4. Section 31.4 *Writing a servlet accessing a bean*

5. Section 31.5 *Writing a JSP accessing a bean*

6. Section 31.6 *Compiling your Web Application*

7. Section 31.7 *Running your Web Application*

8. Section 31.8 *Troubleshooting*

The two families of Tomcat products—Tomcat 3.3.x and Tomcat 4.0.x—differ on several points, but we describe how to work with each Tomcat family.

31.1. Installing Tomcat

1. Download Tomcat from the Apache site (http://jakarta.apache.org/site/binindex.html).

2. Install Tomcat by unzipping the download file and putting it in a specific directory.

   - For Tomcat 3.3.x: This directory will be referred to as `$TOMCAT_HOME`. Set the TOMCAT_HOME variable in your environment and export it.

   - For Tomcat 4.0.x: This directory will be referred to as `$CATALINA_HOME`. If you want to share a single copy of Tomcat 4 among multiple users on the same server, you must configure a CATALINA_BASE environment variable (in addition to CATALINA_HOME) that points to a directory that is unique to your instance. Set both variables in your environment and export them.

3. On Windows, set the JAVA_HOME variable in your environment to your JDK installation directory.

31.2. Configuring Tomcat for JOnAS

As of JOnAS 2.4.4, a new JOnAS service `tomcat` is provided for launching a servlet container Tomcat inside the JOnAS JVM. As of JOnAS 2.5, two implementations of this service are provided: one for Tomcat 3.3.x and another for Tomcat 4.0.x. All you have to do is to add

`tomcat` to the `jonas.services` property of the `jonas.properties` file and set the `jonas.service.tomcat.class` property of the `jonas.properties` file to:

1. `org.objectweb.jonas.tomcat.EmbeddedTomcatImpl33` for Tomcat 3.3.x (this is the default value)

2. `org.objectweb.jonas.tomcat.EmbeddedTomcatImpl40` for Tomcat 4.0.x

Note that if you want to run the servlet container and the EJB container in separate JVMs, you can run two instances of the JOnAS server, one with only the `tomcat` service, and another with all services but tomcat.

31.3. Structuring your Web Application

A Web Application is made of several parts:

- An EJBServer running the different JOnAS services and possibly the registry
- A Tomcat server running servlets and JSPs
- Your browser remotely accessing HTML pages, servlets, and JSPs.

The directory structure of your application should look like this:

`etc`

> The `web.xml` file describing your web application.

`web`

> HTML files, images, and JSPs that will be copied to the Tomcat directory.

`src`

> Servlet sources that should mimic the package hierarchy.

`beans`

> Bean sources that should mimic the package hierarchy.

> **Note**
>
> If your beans come from another application, you do not need the `beans` directory.

Two examples using Tomcat and JOnAS are provided with the JOnAS distribution. They can be found in the `example` directory:

- The Alarm demo
- The websample example

31.4. Writing a servlet accessing a bean

You must write in your servlet code the same code you would write in a simple Java application:

• Create an initial context
• Look up the bean home
• Find or create a bean and access it.

An example of Java servlet can be found in the websample example.

31.5. Writing a JSP accessing a bean

Writing JSPs in place of a java servlet allows you to easily create web content that has both static and dynamic components. JSP offers all the dynamic capabilities of a Java Servlet but provides a more natural approach to creating static content. More details about using JSPs can be found on java.sun.com (http://java.sun.com/products/jsp/).

It could be a good idea to hide all the mechanisms to access EJBs from JSPs by using a proxy java bean, referred in the JSP by the special `usebean` tag. This technique is shown in the alarm example; just look at one of the `.jsp` files. A proxy java bean `ViewProxy.java` has been used to contain the glue necessary to access EJB.

31.6. Compiling your Web Application

With Ant (http://jakarta.apache.org/ant/) you can compile the Web applications that you will find under `$JONAS_ROOT/examples`. You have to run `build install` to compile examples and install the `.war` file in tomcat.

If you use gnu `make`, you just have to run:

• `make` to compile the whole application
• `make install` to install the `.war` file in Tomcat.

You can also run compilers manually via `compile.sh` or `compile.bat`.

Looking at the alarm example, the following steps are performed run `rebuild all` on the web application:

1. Run `make install` in the bean directory to build the `alarm.jar` file and install it in `$JONAS_ROOT/ejbjars`
2. Create in `output/webapps` an `alarm` directory that will be used to build the `alarm.war` file that will be installed in `$TOMCAT_HOME/webapps` (`$CATALINA_HOME/webapps`)
3. Copy `$JONAS_ROOT/ejbjars/alarm.jar` to `output/webapps/alarm/WEB-INF/lib`. This is the way servlets or JSPs will see the bean classes. Another possibility would be to make this classes accessible by the classpath.
4. Copy `etc/web.xml` to `output/webapps/alarm/WEB-INF`
5. Copy all files under the `web` directory in `output/webapps/alarm`
6. Make a `.war` file that has all the directory hierarchy under `output/webapps/alarm`

Install the application by copying `alarm.war` to `$TOMCAT_HOME/webapps` (`$CATALINA_HOME/webapps`) and copying `apps-alarm.xml` to `$TOMCAT_HOME/conf` (for Tomcat 4.0.x, `apps-alarm.xml` is not used).

31.7. Running your Web Application

You are now ready to run your web application:

1. Start the `EJBServer` with the `tomcat` service

2. Use your browser to access the welcome page, usually index.html

31.8. Troubleshooting

Under certain circumstances with JOnAS (JEREMIE), you may have a `java.lang.ClassCastException` in a servlet that makes:

```
home =
   (XXHome)PortableRemoteObject.narrow(initialContext.lookup("XXHome"),
   XXHome.class);
```

The workaround with Tomcat 3.3.x is to make visible from the CLASSPATH the `ejb-jar` file in which the class XXHome is defined.

The workaround with Tomcat 4.0.x is to remove the `ejb-jar` file from `$TOMCAT_HOME/webapps/WEB-INF/lib` and to make it visible from the CLASSPATH.

Chapter 32.
Integrating Tomcat and JOnAS

Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. This guide describes how to integrate JOnAS with Tomcat from Apache (see http://jakarta.apache.org/tomcat/index.html).

Since JOnAS 2.6 provides a full J2EE application server, Tomcat can be used with JOnAS as Web container. This functionality is set up via the JOnAS `web` container service. At this time Tomcat 4.1.x is supported as the implementation of the JOnAS `web` container service. Note that in earlier versions of JOnAS a service `tomcat` was provided with two implementations one for Tomcat 3.3.x and one for Tomcat4.0.x. This service `tomcat` is now deprecated because it was not compliant to J2EE specification. So now, Tomcat 3.3.x is no longer supported with JOnAS (except through the deprecated service `tomcat`).

In the JOnAS J2EE server you will be able to run your beans inside EJB containers via the `ejb` service or `ear` service while you will be able to run your servlets or JSPs via the `web` container service or `ear` service.

JOnAS will be used to run your beans inside EJB containers while Tomcat will be used to run your servlets or JSPs. Inside these servlets or JSPs, you will be allowed to remotely access beans inside the JOnAS EJBServer.

For the cases where security is needed, Tomcat will be used also for user authentication. Refer to the security guide (see http://www.objectweb.org/jonas/doc/howto/SecurityExample.html) for more information on how to use security with JOnAS and Tomcat.

The following steps configure JOnAS and Tomcat to work together:

1. Section 32.1 *Configuring Tomcat for JOnAS*
2. Section 32.2 *Structuring your Web Application*

32.1. Configuring Tomcat for JOnAS

Since JOnAS 2.6 a new JOnAS service `web` is provided for launching a servlet container inside the JOnAS JVM.

At this time, a Tomcat 4.1.x implementation of this service is provided as a servlet container.

All you have to do is to add `web` in the `jonas.services` property of `jonas.properties` file and to set the `jonas.service.web.class` property of `jonas.properties` file to:

```
org.objectweb.jonas.web.catalina41.CatalinaJWebContainerServiceImpl
```

(It is the default value.)

Then you configure Tomcat as usual by updating if needed the `$CATALINA_HOME/conf/server.xml` or the `$CATALINA_BASE/conf/server.xml` file.

Configure Tomcat with a JOnAS realm. See Chapter 33 *Example for Using Security in JOnAS with Tomcat*.

If you want to customize a web application that will be deployed by the `web` container service or the `ear` service you can do this in the `server.xml` file by setting a <Context/> element with attribute className set to `org.objectweb.jonas.web.catalina41.JOnASStandardContext`

You can fiund a `server.xml` example in `JONAS_ROOT/conf/server.xml`

Of course this is only a subset of what is possible to configure in Tomcat. For more information, see Tomcat configuration documentation

32.2. Structuring your Web Application

How to use servlets and JSPs is explained in Chapter 16 *Developing Web Components* or Chapter 22 *Defining the EAR Deployment Descriptor*.

Chapter 33.

Example for Using Security in JOnAS with Tomcat

In the JOnAS distribution, there is an example named `earsample`, showing how to use the security. This chapter describes in more details this example, in order to illustrate how the security capabilities of JOnAS should be used. The example is a Session Bean whose methods are protected. The user calling this bean must be authorized to access these methods.

Since there is at the present time no authentification in JOnAS, we rely on Tomcat to make the identification and authentification steps.

33.1. Example Description

This example shows how to access an EJB whose methods are protected.

1. The user must be *identified* and *authenticated*. We rely on Tomcat to do these steps.

2. Once done, the user has access to a servlet in Tomcat. This servlet calls some methods of the EJB with a *security context* in which the identity of the user approved by Tomcat is kept (this identity is called a *principal*).

3. The security service of JOnAS checks if the principal of the user is authorized to access the called method of the EJB.

4. If the access is allowed, the method is normally executed, otherwise an exception is sent back to the servlet which informs the user.

33.2. Setting Up the Example

In order to make this example convenient to run, all the deployment phase is hidden. Now we will have a closer look of what is needed to set up a secured EJB in JOnAS.

33.2.1. Setting up Tomcat

To run this example, the first thing to do is to configure Tomcat since JOnAS relies on it to identify and authenticate the user of a servlet.

This is done in two steps:

1. The first one is to protect the access to the servlet so that the user has to identify and authenticate prior to access the servlet. In order to do that, we have to modify the `web.xml` http://www.objectweb.org/jonas/current/examples/earsample/etc/xml/web.xml file. This file contains information relative to the web application. Among it, there is the protected access to the servlets. What is interesting concerning security is the following one:

```
...
<security-constraint>
   <web-resource-collection>
<web-resource-name>Protected Area</web-resource-name>
<url-pattern>/secured/*</url-pattern>
...
   </web-resource-collection>
   <auth-constraint>
<role-name>tomcat</role-name>
```

```
...
   </auth-constraint>
</security-constraint>
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>Example Basic Authentication Area</realm-name>
</login-config>
...
```

This means that:

- All URLs of the form `http://<hostname>:<port>/earsample/secured/...` are protected.

- Only the `tomcat` *role* can access these URLs.

- The login configuration is `BASIC` that is to say that a dialog box will be opened when you will access these URLs, asking you to enter a *User ID* and a *password*.

2. We have to define who is in the role `tomcat`. This is specified in the `tomcat-users.xml` file located in your `$CATALINA_HOME/conf/` directory for Tomcat 4.1.x. This file contains:
```
<tomcat-users>
   <user name="tomcat" password="tomcat" roles="tomcat" />
   ...
</tomcat-users>
```

   This means that the user with the `name` (or `User ID`) `tomcat` is in the *role* `tomcat` and is authentified thanks to the *password* `tomcat`.

3. Then, we must indicate to tomcat to use a JOnAS `AccessInterceptor` in order to make the propagation of security context mechanism available.

   - For Tomcat 4.1.x, this is done by updating the `$CATALINA_HOME/conf/server.xml` file. Replace the line:
```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
     debug="0" resourceName="UserDatabase"/>
```
   by
```
<Realm
  className="org.objectweb.security.catalina41.realm.UserDatabaseRealm"
  debug="0" resourceName="UserDatabase"/>
```
   or
```
<Realm
  className="org.objectweb.security.catalina41.realm.MemoryRealm"
  debug="0"/>
```
   or another `org.objectweb.security.catalina41.realm.*` class

33.2.2. Setting up JOnAS

Now, let's take a look on how to configure the security in JOnAS (More details are provided in Chapter 13 *Security Management*. It is very similar to the setting up of Tomcat. It includes the following steps:

1. Define which methods of the EJB are protected and who can access them, this is done in the standard deployment descriptor of the EJB (the ejb-jar.xml file).
```
<ejb-jar>
   ...
   <session>
<ejb-name>Op</ejb-name>
```

```
...
   </session>
   ...
    <assembly-descriptor>
    <security-role>
<role-name>tomcat</role-name>
    </security-role>
    <method-permission>
<role-name>tomcat</role-name>
<method>
<ejb-name>Op</ejb-name>
<method-name>*</method-name>
</method>
    </method-permission>
    ...
   </assembly-descriptor>
</ejb-jar>
```

This means that:

- a *role* tomcat is defined.

- this *role* can access *all* the methods of the Op Session Bean (* is a shortcut to indicate all the methods of the EJB).

2. Now, we have to indicate to JOnAS who can be in the tomcat role. This is done in the jonas-users.properties file:
   ```
   tomcat = tomcat
   ```

   This file is very similar to the tomcat-users.xml file (even if it is a properties file and not an XML file). It stipulates that the tomcat name (on the left side of =) is in the tomcat role (on the right side). More generally, a name may be in several roles which are all on the right side separated by commas. However, there is only *one* name on the left side. That is to say if two users are in the same role, they have to be on two lines and not on one only. Conversely to the tomcat-users.xml file, there is no password since JOnAS is not currently able to authenticate user.

33.3. Some warnings...

In this example, there are some important points:

- the tomcat role defined on the tomcat-users.xml file is local to Tomcat. It is used to authorize access to the servlet.

- the tomcat role defined on the jonas-users.properties file is local to JOnAS. It is used to authorize access to the methods of the bean. We have used the same name, tomcat for both, for convenience but it was not mandatory.

- the names defined in the tomcat-users.xml and jonas-users.properties files *must* be the same. Indeed the bean is considering the servlet as its client so that the name of the client in the servlet must be used to control access of the bean.

33.4. Do not forget...

Using security with RMI requires that you deploy the bean by using the -secpropag option of GenIC.

How to Install a jUDDI Server on JOnAS

34.1. UDDI Server

A UDDI server is basically a web services registry.

Providers of web services put technical information (such as the WSDL definition used to access the web service, its description, ...) inside these registries. Web services users can browse these registries to choose a web service that fits their needs.

34.2. What is jUDDI?

jUDDI (pronounced "Judy") is a Java-based implementation of the "Universal Description, Discovery, and Integration" (UDDI) specification (v2.0) for Web services. It is implemented as a pure Java web application and can be deployed with a minimum amount of work inside JOnAS.

For more information, see http://ws.apache.org/juddi.

34.3. Where Can I Find the Latest Version?

JOnAS already includes jUDDI v0.8 as a preconfigured Web application (CVS Snapshot (21-Jan-2005) + 2 patchs). However, you can find the *latest* jUDDI version at http://ws.apache.org/juddi.

34.4. Installation Steps

jUDDI needs a minimum of configuration steps in order to be successfully deployed inside JOnAS.

> **Note**
>
> The first step can be ignored if you use the JOnAS-provided `juddi.war`.

34.4.1. Create the jUDDI Web Application

34.4.1.1. Compilation

1. Go to the directory where the jUDDI sources are located (`JUDDI_HOME`).

2. Customize the `JUDDI_HOME/conf/juddi.properties` configuration file.
   ```
   # jUDDI Proxy Properties (used by RegistryProxy)
   juddi.proxy.adminURL = http://localhost:9000/juddi/admin
   juddi.proxy.inquiryURL = http://localhost:9000/juddi/inquiry
   juddi.proxy.publishURL = http://localhost:9000/juddi/publish
   juddi.proxy.transportClass = org.apache.juddi.proxy.AxisTransport
   juddi.proxy.securityProvider = com.sun.net.ssl.internal.ssl.Provider
   juddi.proxy.protocolHandler = com.sun.net.ssl.internal.www.protocol
   ```

```
# jUDDI HTTP Proxy Properties
juddi.httpProxySet = true
juddi.httpProxyHost = proxy.viens.net
juddi.httpProxyPort = 8000
juddi.httpProxyUserName = sviens
juddi.httpProxyPassword = password
```

3. Launch the compilation with Ant WAR. This produces a `juddi.war` inside the `JUDDI_HOME/build/` directory.

34.4.1.2. Customization

JOnAS provides a lightweight `juddi.war` file from which all unnecessary libraries have been removed.

The original `juddi.war` (created from JUDDI_HOME) has a lot of libraries inside its `WEB-INF/lib` that are already provided by JOnAS. These files, listed below, can safely be removed:

• axis.jar

• commons-discovery.jar

• commons-logging.jar

• jaxrpc.jar

• saaj.jar

• wsdl4j.jar

By default, jUDDI includes a `jonas-web.xml` descriptor (in `JUDDI_HOME/conf`). This descriptor specifies the *jndi name* of the DataSource used in jUDDI; its default value is `jdbc/juddiDB`. This value will be used in the <datasource>.properties of JOnAS.

34.4.2. Create the Database

34.4.2.1. Retrieve the SQL scripts for your database

jUDDI comes with SQL files for many databases (MySQL, DB2, HSQL, Sybase, PostgreSQL, Oracle, TotalXML, JDataStore). The SQL scripts are different for each version of jUDDI:

• MySQL

  • 0.9rc1 : create_database.sql insert_publishers.sql

• DB2

  • 0.9rc1 : create_database.sql insert_publishers.sql

• HSQL

  • 0.9rc1 : create_database.sql insert_publishers.sql

• Sybase

  • 0.9rc1 : create_database.sql insert_publishers.sql

• PostgreSQL

- 0.9rc1 : create_database.sql insert_publishers.sql

- Oracle

  - 0.9rc1 : create_database.sql insert_publishers.sql

- TotalXML

  - 0.9rc1 : create_database.sql insert_publishers.sql

- JDataStore

  - 0.9rc1 : create_database.sql insert_publishers.sql

34.4.2.2. Set Up the Database

Execute the given scripts (table creation, tmodels insertions and publishers insertions). Then, a publisher (the user who has the rights to modify the UDDI server) must be added manually. This is currently the only way to add a publisher for jUDDI.

34.4.3. Configure JOnAS Datasource

As jUDDI uses a DataSource to connect to the database, JOnAS must be configured to create this DataSource:

Create a file (named `ws-juddi-datasource.properties` for example) and fill it in according to the database you use.

```
##################### MySQL DataSource configuration example
# datasource.name is the jndi-name set in jonas-web.xml
datasource.name           jdbc/juddiDB
# datasource.url is the URL where the database can be accessed
datasource.url            jdbc:mysql://localhost/db_juddi
# datasource.classname is the JDBC Driver classname
datasource.classname      com.mysql.Driver
# Set the DB username and password here
datasource.username       XXX
datasource.password       XXX
# available values:
#   rdb,rdb.postgres,rdb.oracle,rdb.oracle8,rdb.mckoi,rdb.mysql
datasource.mapper         rdb.mysql
# Add the datasource properties filename
# (without the suffix .properties)
# in ($JONAS_BASE|$JONAS_ROOT)/conf/jonas.properties
jonas.service.dbm.datasources <datasource-filename>
```

34.4.4. Deploy and Test jUDDI

Deploy jUDDI on JOnAS with a command similar to the following:

```
$ jonas admin -a ~/juddi/juddi.war
```

You should see the following output:

```
11:53:57,984 : RegistryServlet.init : jUDDI Starting:
   Initializing resources and subsystems.
```

```
11:53:58,282 : AbsJWebContainerServiceImpl.registerWar : War
/home/sauthieg/sandboxes/ws-projects/ws-juddi/build/juddi.war
   available at the context /juddi.
```

Open your web browser and go to the URL http://localhost:9000/juddi/happyjuddi.jsp to confirm that the jUDDI setup is successful.

If the URL opens, you can access your UDDI server through any UDDIv2.0 compliant browser.

- inquiryURL = http://localhost:9000/juddi/inquiry
- publishURL = http://localhost:9000/juddi/publish

34.5. Links

- UDDI web site (http://sourceforge.net/projects/uddi4j)
- jUDDI web site (http://ws.apache.org/juddi)
- UDDI4J Java implementation (http://sourceforge.net/projects/uddi4j)
- IBM UDDI Test Registry (https://uddi.ibm.com/testregistry/registry.html)
- Microsoft UDDI Test Registry (http://uddi.microsoft.com/)
- XMethods web Site (http://www.xmethods.net/)
- UDDI Browser (http://www.uddibrowser.org/)

Chapter 35.

Howto: JSR 160 Support in JOnAS

This chapter is for the management application developer or administrator who intends to use standard JMX RMI connectors to access an MBean Server running on a JOnAS server.

35.1. What is JSR 160?

The JSR 160 is a Java Management Extension (JMX) Remoting specification that extends the JSR 3 specification by providing a standard API to connect to remote JMX-enabled applications.

Currently, JSR 160 has defined a mandatory connector based on RMI (that supports both RMI/JRMP and RMI/IIOP).

Support for JSR 160 connectors in JOnAS is based on the MX4J JMX implementation (see http://mx4j.sourceforge.net/).

35.2. Connector servers created by JOnAS

Previous and current JOnAS versions implement a proprietary remote object that enables connections to the MBean server. This object is registered in JNDI under the name 'RMIConnector_*jonasServerName*'. It can be accessed using any of the communication protocols support by JOnAS (RMI/JRMP, RMI/IIOP, or JEREMIE - see Section 3.3.1 *Choosing the Protocol*).

JSR 160 support implies providing standard connector server objects. At start-up, the JMX service creates one or more such objects, depending on the JOnAS configuration (in this case, depending on the content of `carol.properties` file). To create a client connector, the client side needs to know the URL of the connector server. The following sections present the URLs that can be used by the clients, depending on the protocol chosen.

Currently, JSR-160 connectors can use only the RMI/JRMP and RMI/IIOP protocols.

35.2.1. Using a RMI/JRMP Connector

To use this connector, the `jrmp` protocol has to be in the `carol.protocols` list.

The client has to construct a `JMXServiceURL` using the following `String`, possibly modified according to the JOnAS specific configuration:

```
service:jmx:rmi:///jndi/rmi://host:1099/jrmpconnector_jonasServerName
```

where `host` is the host on which is running the JOnAS server to be managed. The `port` number is given in the `carol.properties` file.

Then, a `JMXConnector` has to be created and connected to the connector server using the JMX Remote API.

35.2.2. Example 1:

```
Hashtable environment = null;
JMXServiceURL address = new
 JMXServiceURL("service:jmx:rmi:///jndi/rmi://host:1099/jrmpconnector_jonas");
JMXConnector connector =
```

```
 JMXConnectorFactory.newJMXConnector(address, environment);
connector.connect(environment);
```

35.2.3. Using a RMI/IIOP Connector

You can use this connector if the `iiop` protocol is set in the `carol.protocols` list.

This is done in the same way as for JRMP, but the `String` to be used to construct the `JMXServiceURL` must adhere to the following model: `"service:jmx:iiop:///jndi/iiop://host:2000/iiopconnector_`*jonasServerName*`"`

35.2.4. Example 2:

```
Hashtable environment = null;
JMXServiceURL address = new JMXServiceURL
  ("service:jmx:iiop:///jndi/iiop://host:2001/iiopconnector_jonas");
JMXConnector connector =
  JMXConnectorFactory.newJMXConnector(address, environment);
connector.connect(environment);
```

Chapter 36.
Using the MC4J JMX Console

This guide describes how to connect the MC4J JMX Console to the JMX server running in a JOnAS server (see http://mc4j.sourceforge.net/).

Recall that the current JMX implementation used by JOnAS is MX4J (http://mx4j.sourceforge.net/), and that JOnAS supports JSR 160 for remote connections to the JMX server (see Chapter 35 *Howto: JSR 160 Support in JOnAS*).

36.1. Connecting to the JMX server

Launch the MC4J console and create a JSR160 connection using the connection wizard.

- Choose JSR160 connector type.
- Provide a name for your connection.
- Change the server URL. The URL to use depends on the following configuration parameters of JOnAS:
  - JONAS server name.
  - The host on which JOnAS is running.
  - The protocol used by the JOnAS name service (use only `jrmp` or `iiop`)
  - The port number used by the JOnAS name service for that protocol.

- For example, in the case of a JOnAS server having a default configuration and default name, you would use the following URL:
  `service:jmx:rmi:///jndi/rmi://host:1099/jrmpconnector_jonas`

  See Chapter 35 *Howto: JSR 160 Support in JOnAS* for specific JOnAS configurations.

Once connected, you can manage the connection (delete, disconnect or reconnect) and browse the MBeans registered in the JMX server.

Working with Management Beans

This guide contains:

37.1. Target Audience and Rationale

This chapter is intended for advanced JOnAS users who are interested in understanding management facilities that JOnAS provides, and possibly extending these facilities for their application needs.

JOnAS management facilities are based on Management Beans (MBeans), which are compliant with the JMX Specification at http://java.sun.com/products/JavaManagement/. Moreover, JOnAS implements the JSR 77 specification, which defines the management model for J2EE platforms.

37.2. About JOnAS MBeans and their use in JonasAdmin

MBeans provide access to management functions such as configuration, deployment/undeployment, monitoring of resources and application modules.

MBeans are created not only by a variety of JOnAS services, but also by components integrated in JOnAS (Web server Tomcat or Jetty, JORAM MOM, etc.). They are registered in the current MBean Server, which is started by each JOnAS server instance. Remote access to the MBean Server is facilitated by JMX remote connectors compliant with the JSR 160 specification. See more information about connectors in Chapter 35 *Howto: JSR 160 Support in JOnAS*.

The JonasAdmin application implements the management functions listed above using the various MBeans registered in the MBeanServer of the JOnAS instance that is currently being managed. This is usually the server on which JonasAdmin is deployed, but it may be another server running in the same management domain.

JonasAdmin also presents, in a structured way, all the registered MBeans, their attributes, and their operations. In the future, JonasAdmin will probably be extended to allow setting attributes and invoking operations.

37.3. Using JOnAS MBeans in a Management Application

In order to invoke a management operation on a MBean, the caller must access to the MBean server.

37.3.1. Local MBean Server

When the caller is located in the same JVM as the MBean Server, it can use the `javax.management.MBeanServerFactory` class to obtain a reference to the MBean Server:

```
List mbeanServers = MBeanServerFactory.findMBeanServer(null);
if (mbeanServers != null && mbeanServers.size() > 0) {
        return (MBeanServer) mbeanServers.get(0);
```

```
}
```

37.3.2. Using Remote Connectors

When the caller is remote, it can use a JMX remote connector to establish a connection with the MBean Server and obtain a `javax.management.MBeanServerConnection` object.

Suppose that the connector server has the following address: `service:jmx:jrmp://host/jndi/jrmp://host:1099/jrmpconnector_jonas`, which is the default for a JOnAS server called `jonas`.

```
JMXServiceURL connURL = new JMXServiceURL(
    "service:jmx:jrmp://host/jndi/jrmp://host:1099/jrmpconnector_jonas");
JMXConnector connector =
    JMXConnectorFactory.newJMXConnector(connURL, null);
connector.connect(null);
MBeanServerConnection conn = connector.getMBeanServerConnection();
    return conn;
```

37.3.3. Using a Management EJB

A remote caller can also use the MEJB provided by the JOnAS distribution. A Management EJB implementation, compliant to the JSR 77, is packed in the `mejb.ear` installed in the `JONAS_ROOT/ejbjars/autoload` directory. Thus, the MEJB is automatically deployed at JOnAS start-up. Its `Home` is registered in JNDI under the name `ejb/mgmt/MEJB`. The JOnAS distribution also contains an example using the MEJB in a J2EE application, the `j2eemanagement` sample.

37.3.4. Using the Management Web Service Endpoint

A remote caller can use the Management Web Service endpoint that is packaged as a part of the `mejb.ear`, which is installed in the `JONAS_ROOT/ejbjars/autoload` directory. Thus, the management endpoint is automatically deployed at JOnAS start-up. Check http://<hostname>:<port>/mejb/ManagementEndpoint/ManagementEndpoint?wsdl on a running JOnAS instance for the WSDL file. The endpoint allows light-weight clients to query JOnAS MBeans from virtually any platform by leveraging the platform-independant nature of Web Services.

37.4. Registering User MBeans

Application MBeans can be created and registered in the MBean server by calling `registerMBean` method on the `MBeanServer` object or `createMBean` method on the `MBeanServerConnection` object. Also, MBeans can be loaded using the m-let service, a dynamic loading mechanism provided by the JMX implementation. This mechanism allows loading MBeans from a remote URL. The information on the MBeans to load is provided in an m-let text file. Refer to the JMX implementation documentation for details concerning this file. In addition, Chapter 38 *JOnAS and JMX: Registering and Manipulating MBeans* illustrates the use of this m-let mechanism. In order to make an m-let text file accessible to JOnAS applications, install it in the `JONAS_ROOT/conf` directory.

JOnAS and JMX: Registering and Manipulating MBeans

JMX (Java Management eXtensions) is an API for managing, among other things, J2EE applications. JOnAS (version 4 and above) integrates the MX4J open-source JMX server and registers a number of MBeans. The web-based JonasAdmin application acts as a JMX client, enabling viewing and manipulation of these MBeans.

It maybe desirable for an application to expose its own MBeans via the JMX server to allow application management (using, for example, MC4J). JOnAS currently does not provide a prebuilt method for registering MBeans with its JMX server. This chapter illustrates one method of registering application specific MBeans with the JOnAS JMX server based on the m-let service.

38.1. ServletContextListener

The basic task of registering an MBean with the JOnAS JMX server is accomplished by the following implementation of the ServletContextListener interface (http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletContextListener.html). This implementation reads a number of MLet files, which specify the MBeans to be registered, and attempts to register those beans during the web application context initialization.

```
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

import javax.management.InstanceAlreadyExistsException;
import javax.management.InstanceNotFoundException;
import javax.management.ReflectionException;
import javax.management.MBeanServer;
import javax.management.MBeanException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServerFactory;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectInstance;
import javax.management.ObjectName;
import javax.management.loading.MLet;
import javax.management.loading.MLetMBean;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextListener;

import org.apache.log4j.Logger;

/**
 * ServletContextListener designed to register JMX MBeans into
 * an existing JMX server when the application starts up. The
 * MBeans to be loaded are specified in the MLet files whose
 * names are given in the servlet context parameter with the name
 * mletFiles, in a semi-colon delimited list (This is not really
```

```
 * needed as multiple mlets can be specified in one file, it might
 * be useful). Note that the filename are relative to the WEB-INF
 * directory of the servlet context being listened to.
 *
 * Note, that this listener should precede (in web.xml) any other
 * that depend on the MBeans being registered.
 *
 * Note that the MBean registration is sensitive to classloader
 * issues. For example, when registering the MBeans in the JMX server
 * provided by the Jonas application server any libraries required by
 * the MBeans need to be in the central lib directory (lib/ext).
 *
 *
 * @author Jonny Wray
 *
 */
public class MBeanRegistrationListener implements
     ServletContextListener {
         private static final String MLET_DOMAIN =
             "MBeanRegistrationListener";
         private static final String MLET_BEAN_NAME =
             MLET_DOMAIN+":Name=MLet";
         private static final String MLETFILE_INITPARAM_NAME =
             "mletFiles";
         private static final Logger log =
             Logger.getLogger(MBeanRegistrationListener.class);


         private MBeanServer lookForExistingServer(){
                 List mbeanServers =
                     MBeanServerFactory.findMBeanServer(null);
                 if(mbeanServers != null && mbeanServers.size()
                     > 0){
                         return (MBeanServer)mbeanServers.get(0);
                 }
                 return null;
         }

         private MBeanServer getMBeanServer(){
                 MBeanServer server = lookForExistingServer();
                 if(server == null){
                     server =
                     MBeanServerFactory.createMBeanServer(MLET_DOMAIN);
                 }
                 return server;
         }

         public void contextDestroyed(ServletContextEvent arg0) {
                 log.info("Destroy event");
                 // Anything that needs to be done here on deregistering
                 // of the web application?
         }


private List getMLetURLs(String filenames){
    List urls = new ArrayList();
        StringTokenizer tok = new StringTokenizer(filenames, ";");
            while(tok.hasMoreTokens()){String filename = tok.nextToken();
URL configURL =
  Thread.currentThread().getContextClassLoader().getResource(filename);
if(configURL == null){
```

```
    log.error("Cound not load MLet file resource from "+filename+"
      using current thread context classloader");
    }
    else{
        urls.add(configURL);
        }
    }
        return urls;
    }

    private List getMLetURLs(ServletContext context, String filenames){
        List urls = new ArrayList();
        StringTokenizer tok =
                new StringTokenizer(filenames, ";");
        while(tok.hasMoreTokens()){
            String filename = "/WEB-INF/"+tok.nextToken();
            URL configURL = null;
            try {
                 configURL = context.getResource(filename);
            }
            catch (MalformedURLException e) {
                   log.error("URL for "+filename+" is malformed", e);
            }
            if(configURL == null){
                log.error("Cound not find MLet file resource from
                    "+filename+" in servlet context");
            }
            else{
                urls.add(configURL);
            }
        }
        return urls;
    }

        /**
         * Dynamically register the MBeans specified in the list
         * of MLet files (relative to /WEB-INF/) specified in servlet
         * context parameter mletFiles as a semi-colon deliminated list
         * of file names.
         *
         * The algorithm looks for already running JMX servers and uses
         * the first it comes across. If no servers are running, then
         * it creates one.
         *
         * Note, the interface does not define any exceptions to be
         * thrown. Currently, any exceptions thrown during registration
         * are logged at error level and then ignored. This seems
         * reasonable, as these may or may not be a fatal event. In
         * this way the registration process reports its failure and
         * the application context initialization continues.
         */
public void contextInitialized(ServletContextEvent arg0) {
    log.info("Initializing event");
    String filenames =
    arg0.getServletContext().getInitParameter(MLETFILE_INITPARAM_NAME);
    if(filenames != null && filenames.length() > 0){
        MBeanServer server = getMBeanServer();
        if(server != null){
            try{
                ObjectName name = new ObjectName(MLET_BEAN_NAME);
                if(!server.isRegistered(name)){
                    log.info("Creating new MLetMBean
```

```
                        for dynamic registration");
                MLetMBean mletService = new MLet();
                server.registerMBean(mletService, name);
            }
            List urls = getMLetURLs(arg0.getServletContext(), filenames);
            for(int i=0;i < urls.size();i++){
                URL url = (URL)urls.get(i);
                try {
                    log.info("Registering MBeans from MLet file
                        "+url);
                    Set loadedMBeans =
                        (Set)server.invoke(name, "getMBeansFromURL",
                            new Object[]{url},
                            new String[]{URL.class.getName()});

                    processRegisteredMBeans(loadedMBeans);
                }
                catch (InstanceNotFoundException e) {
                    log.error("Unable to register MBeans from MLet
                        file "+url, e);
                }
                catch (MBeanException e) {
                    log.error("Unable to register MBeans from MLet
                        file "+url, e);
                }
                catch (ReflectionException e) {
                    log.error("Unable to register MBeans from MLet
                        file "+url, e);
                }
            }
        }
        catch(MalformedObjectNameException e){
                log.error("Unable to register the MLetMBean", e);
        }
        catch(NotCompliantMBeanException e){
                log.error("Unable to register the MLetMBean", e);
        }
        catch(MBeanRegistrationException e){
                log.error("Unable to register the MLetMBean", e);
        }
        catch(InstanceAlreadyExistsException e){
                log.error("Unable to register the MLetMBean", e);
        }
    }
    else{
        log.error("MBeanServer not found and could not be created.
            Not registering MBeans.");
    }
    }
    else{
        log.error("No mletFiles servlet context parameter found.");
    }
}

private void processRegisteredMBeans(Set loadedMBeans) {
        log.debug("Loaded beans: "+loadedMBeans.size());
        Iterator it = loadedMBeans.iterator();
        while(it.hasNext()){
                Object o = it.next();
                if(o instanceof ObjectInstance){
                        ObjectInstance inst = (ObjectInstance)o;
                        log.info("Registered: "+inst.getObjectName());
```

```
                }
                else if(o instanceof Throwable){
                        Throwable err = (Throwable)o;
                        log.error("Error registering MBeans", err);
                }
        }
}

}
```

38.2. Configuration

In order to use the above ServletContextListener it must be configured in the `web.xml` of the web application that wants to register the MBeans. For example, the following lines added to `web.xml` will result in the registration of the MBeans specified in the `application.mlet` file under the WEB-INF directory. Multiple MLet files can be specified in a comma-separated list.

```
<context-param>
    <param-name>mletFiles</param-name>
    <param-value>application.mlet</param-value>
</context-param>

<listener>
   <listener-class>net.fiveprime.jmx.MBeanRegistrationListener
      </listener-class>
</listener>
```

An example MLet file to load an extension (detailed below) of the HibernateService MBean is:

```
<mlet code="ConfigurableHibernateService"
 name="HibernateService:Name=HibernateService"
 archive="mbeans.jar">
  <arg type="java.lang.String" value="hibernate.cfg.xml">
  <arg type="java.lang.String" value="hibernate/HibernateSessionFactory">
  <arg type="java.lang.String" value="DefaultDS">
</mlet>
```

38.3. Library dependencies

Registration of MBeans results in their construction by the JMX server. As such, any classes the MBean is dependent on must be available to the JMX server, in `lib/ext`.

38.4. HibernateService Extension

The Hibernate distribution provides an implementation of HibernateServiceMBean in the class HibernateService. The MLet file above specifies an extension of this class that allows the HibernateService to be configured from an external file, such as the standard `hibernate.cfg.xml` file. There are a number of situations where it is desirable to use the Hibernate mapped classes outside of Jonas running a JMX server. This enables the Hibernate mapping files and properties to be specified in one place and used in multiple situations. If this is not needed, then the `HibernateService` class could be used directly.

```
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
```

```java
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.jmx.HibernateService;

import org.apache.commons.digester.Digester;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.xml.sax.SAXException;
/**
 * Extension of the HibernateService class to add configuration
 * ability from a Hibernate XML configuration file.
 *
 * @author Jonny Wray
 *
 */
public class ConfigurableHibernateService extends HibernateService {

private static Log log =
    LogFactory.getLog(ConfigurableHibernateService.class);

/**
 * Configure this HibernateService from an XML file
 *
 * @param filename The Hibernate XML configuration file, for example
 * hibernate.cfg.xml
 * @param jndiName The JNDI name that the session factory will be
 * registered under @param datasourceName
 * The name of the datasource used by the session factory
 * @throws HibernateException
 * If there's a problem reading the configuration file
 */
public ConfigurableHibernateService(String filename,
    String jndiName, String datasourceName)
        throws HibernateException{

        init(filename, jndiName, datasourceName);
        start();
}

private void init(String filename, String jndiName,
    String datasourceName) throws HibernateException {
        if(log.isDebugEnabled()){
            log.debug("Configuring Hibernate JMX MBean with
                filename "+filename+ ", JNDI name "+jndiName+"
                and datasource "+datasourceName);
    }
    try{
        URL url =
            this.getClass().getClassLoader().getResource(filename);
        Digester mappingDigester = configureMappingDigester();
        List results =
            (List)mappingDigester.parse(url.openStream());
        Iterator it = results.iterator();
        while(it.hasNext()){
            StringBuffer buffer = (StringBuffer)it.next();
            addMapResource(buffer.toString());
            log.debug("Adding mapping resource "+buffer.toString());
        }
```

```
        Digester propertyDigester = configurePropertyDigester();
        Map resultMap = (Map)propertyDigester.parse(url.openStream());
        it = resultMap.keySet().iterator();
        while(it.hasNext()){
            String key = (String)it.next();
            String value = (String)resultMap.get(key);
            setProperty("hibernate."+key, value);
            log.debug("Adding property ("+key+","+value+")");
        }
        setJndiName(jndiName);
        setDatasource(datasourceName);
    }
    catch(IOException e){
        throw new HibernateException(e);
    }
    catch(SAXException e){
        throw new HibernateException(e);
    }
}

private Digester configureMappingDigester(){
    Digester digester = new Digester();
    digester.setClassLoader(this.getClass().getClassLoader());
    digester.setValidating(false);
    digester.addObjectCreate("hibernate-configuration/session-factory",
        ArrayList.class);

    digester.addObjectCreate(
        "hibernate-configuration/session-factory/mapping",
        StringBuffer.class);
    digester.addCallMethod(
        "hibernate-configuration/session-factory/mapping",
        "append", 1);
    digester.addCallParam(
        "hibernate-configuration/session-factory/mapping",
        0, "resource");
    digester.addSetNext(
        "hibernate-configuration/session-factory/mapping",
        "add");

    return digester;
}

private Digester configurePropertyDigester(){
    Digester digester = new Digester();
    digester.setClassLoader(this.getClass().getClassLoader());
    digester.setValidating(false);
    digester.addObjectCreate("hibernate-configuration/session-factory",
        HashMap.class);

    digester.addCallMethod(
        "hibernate-configuration/session-factory/property", "put", 2);
    digester.addCallParam(
        "hibernate-configuration/session-factory/property", 0, "name");
    digester.addCallParam(
        "hibernate-configuration/session-factory/property", 1);

    return digester;
}
}
```

How to Use JOnAS through a Firewall

This chapter is for JOnAS administrators who intend to use RMI/IIOP to run JOnAS through a firewall.

The example that follows describes how to modify JOnAS configuration files in order to specify a range of ports to be used by RMI when using the IIOP protocol. This example considers a range of 10 ports, starting at port 33000, and gives the IP address 150.11.135.17 for the host running JOnAS.

1. Uncomment the following properties of the `$JONAS_BASE/jacorb.properties` file and specify the appropriate values for the port range and host IP address:
   ```
   jacorb.net.server_socket_factory=
       org.jacorb.orb.factory.PortRangeServerSocketFactory
   jacorb.net.server_socket_factory.port.min=33000
   jacorb.net.server_socket_factory.port.max=33010
   OAIAddr=150.11.135.17
   ```

2. In the `$JONAS_BASE/carol.properties` file, specify the use of the IIOP protocol and set the port number for the registry to be the first port number of the range to be used:
   ```
   carol.protocols=iiop
   carol.iiop.url=iiop://150.11.135.17:33000
   ```

3. Launch JOnAS:
   ```
   service jonas start
   ```

   Note

   This configuration for specifying the port numbers will not work if you launch JOnAS in background, so you must launch JOnAS in the foreground.

How To Configure and Use xdoclet for JOnAS

This guide contains the following sections:

40.1. Downloading and installing xdoclet

You can find Xdoclet for JOnAS on ObjectWeb SourceForge / Tools / xdoclet / xdocletForJOnAS42.zip

This ZIP file contains:

- `test`: A folder that contains tests for JOnAS tags. You can use this as an example for your applications.
- `objectweb-tags.html`: Tags for the creation of JOnAS-specific resources (for use with xdoclet-objectweb-module-1.2.2.jar).
- `xdoclet-objectweb-module-1.2.2.jar`: Updated ObjectWeb libraries for xdoclet.
- `Readme.txt`: This readme file.

40.1.1. How to use this archive:

- Download xdoclet from: http://xdoclet.sourceforge.net/install.html
- Install xdoclet and replace your local xdoclet-objectweb-module-1.2.2.jar file with this new version of that file.

> **Note**
>
> The JOnAS examples (xdoclet and olstore) use the $JONAS_ROOT/lib/example/xdoclet/xdoclet-objectweb-module-1.2.2.jar.

40.2. Ejbdoclet Ant Target

The following ejbdoclet Ant target illustrates how to launch an xdoclet task to generate a deployment descriptor for JOnAS. The `defaultentityjndiname` is not mandatory, but you can use it if you do not want to specify the `@bean.jndi-name` for your Entity beans.

```
<target name="ejbdoclet" >
    <taskdef
      name="ejbdoclet"
      classname="xdoclet.modules.ejb.EjbDocletTask"
```

```
        classpathref="project.class.path"
    />

    <ejbdoclet
        destdir="${build.generate.dir}"
        excludedtags="@version,@author"
        addedtags="@xdoclet-generated at ${TODAY}"
        ejbspec="2.0"
        verbose="true"
    >
     ...
        <jonas version="4.2"
            validateXML="true"
            destdir="${build.generate.dir}"
            defaultentityjndiname="jdbc_1"
        />
    </ejbdoclet>
</target>
```

40.3. Xdoclet tags

40.3.1. Field-Level Tags

40.3.1.1. jonas.bean

The `jonas.bean` element declares the JOnAS-specific information for an enterprise bean.

Parameter	Type	Applicability	Description	Mandatory
ejb-name	text		The enterprise bean's name specified in the standard EJB deployment descriptor.	`true`
jndi-name	text		The JNDI name of the enterprise bean's home. Concerns only the Entity and Session beans. Mandatory if version < 2.5, but optional for Session beans for 2.5 onwards.	false
cleanup	text		Determines the jonas-specific behavior for table management at deploy time.	false
lock-policy	text		Determine the jonas-specific lock policy for database access.	false
automatic-pk-field-name	text		The jdbc column name for automatic primary key; auto-generated.	false
inactivity-timeout	text		Optional inactivity-timeout value (integer value)	false
prefetch	text		Optional prefetch (boolean value)	false

40.3.1.2. jonas.resource

The `jonas.resource` element declares the JOnAS-specific information for an external resource referenced by a bean.

Parameter	Type	Applicability	Description	Mandatory
res-ref-name	text		The name of the resource reference specified in the standard EJB deployment descriptor.	`true`
jndi-name	text		The JNDI name of the resource.	`true`

40.3.1.3. jonas.resource-env

The `jonas.resource-env` element declares the JOnAS-specific information for an external resource environment referenced by a bean.

Parameter	Type	Applicability	Description	Mandatory
resource-env-ref-name	text		The name of the resource environment reference specified in the standard EJB deployment descriptor.	`true`
jndi-name	text		The JNDI name of the resource environment.	`true`

40.3.1.4. jonas.ejb-ref

The `jonas.ejb-ref` element declares the JOnAS-specific information for a reference to another enterprise bean's home.

Parameter	Type	Applicability	Description	Mandatory
ejb-ref-name	text		The name of the EJB reference specified in the standard EJB deployment descriptor.	`true`
jndi-name	text		The JNDI name of the ejb.	`true`

40.3.1.5. jonas.ejb-ref

The `jonas.ejb-ref` element specifies the value of timeout in seconds for expiration of session instances.

Parameter	Type	Applicability	Description	Mandatory
session-timeout	int		The value of timeout in seconds for expiration of session instances.	`true`

40.3.1.6. jonas.is-modified-method-name

The `jonas.is-modified-method-name` element specifies the name of the is-modified method of an entity.

Parameter	Type	Applicability	Description	Mandatory
is-modified-method-name	text		The name of the is-modified method of an entity.	`true`

40.3.1.7. jonas.shared

The `jonas.shared` element specifies whether the bean state can be accessed outside JOnAS. This tag was introduced in version 2.4.

Parameter	Type	Applicability	Description	Mandatory
shared	bool		True if the bean state can be accessed outside JOnAS. The default is False.	`true`

40.3.1.8. jonas.passivation-timeout

The `jonas.passivation-timeout` element specifies the value of timeout in seconds for passivation of entity instances when no transaction are used.

Parameter	Type	Applicability	Description	Mandatory
passivation-timeout	int		The value of timeout in seconds for passivation of entity instances when no transaction are used.	`true`

40.3.1.9. jonas.max-cache-size

The `jonas.max-cache-size` element defines the max number of instances (int value) that can be held in memory. The default value is infinite. This tag was introduced in version 2.4.

40.3.1.10. jonas.min-pool-size

The `jonas.min-pool-size` element specifies the number of instances that will be created to populate the pool when the bean is loaded for the first time. The default value is 0. This tag was introduced in version 2.4.

40.3.1.11. jonas.jdbc-mapping

The `jonas.jdbc-mapping` element declares the mapping of an entity with container-managed persistence to the underlying database.

Parameter	Type	Applicability	Description	Mandatory
jndi-name	text		The JNDI name of the datasource.	`true`
automatic-pk	bool		True or False for use automatic generation of primary key.	false
jdbc-table-name	text		The name of the relational table.	`true`

40.3.1.12. jonas.finder-method-jdbc-mapping

The `jonas.finder-method-jdbc-mapping` element declares the SQL `WHERE` clause associated to a finder method of a container-managed persistence entity.

Parameter	Type	Applicability	Description	Mandatory
method-name	text		The method's name.	`true`
method-params	text		Identifies a single method among multiple methods with an overloaded method name.	false
jdbc-where-clause	text		The SQL WHERE clause.	`true`

40.3.1.13. jonas.jonas-message-driven-destination

The `jonas.jonas-message-driven-destination` element declares the JOnAS-specific information for a the message-driven bean destination.

Parameter	Type	Applicability	Description	Mandatory
jndi-name	text		The JNDI name of the message driven destination.	`true`

40.3.2. Method Level tags

40.3.2.1. jonas.ejb-relation

The `jonas.ejb-relation` element declares the CMR fields Mapping to primary-key-fields to the underlying database.

Parameter	Type	Applicability	Description	Mandatory
pk-composite	text		true if the pk is composite (default value = false)	false
ejb-relation-name	text		The name of the relationship.	`true`

Parameter	Type	Applicability	Description	Mandatory
jdbc-table-name	text		The optional name of the relation joint table.	false
ejb-relationship-role-name1	text		The name of the first relationship role.	`true`
foreign-key-jdbc-name1	text		The column(s) name(s) of the foreign key	`true`
ejb-relationship-role-name2	text		The name of the second (if the relation is bi-directional) relationship role.	false
foreign-key-jdbc-name2	text		The column names of the foreign key	false

40.3.2.2. jonas.cmp-field-jdbc-mapping

The `jonas.cmp-field-jdbc-mapping` element declares the mapping of a container-managed field of an entity to a column of a relational table.

Parameter	Type	Applicability	Description	Mandatory
field-name	text		The field's name.	`true`
jdbc-field-name	text		The column name of the relational table.	`true`
sql-type	text		The sql-type element specifies the SQL type (CMP2 only)	false
key-jdbc-name	text		The column name of the primary key composite of this field	false

Domain Management in JOnAS

41.1. Domain Management Goals

A JOnAS management domain is composed of a set of JOnAS servers that are running under the same management authority and are identified by a *domain name*.

The servers in a domain can be administered by a management application running on one server (or several) in the domain; this is the role of administrator or *master*. The managed servers play the role of *slaves*.

Typically when deploying the JonasAdmin application on a *slave*, the administrator can manage only the server on which the application is running. When deploying JonasAdmin on a *master* server, the administrator can manage all the servers in the domain.

41.2. Mechanisms Used by Domain Management

Basically, domain management in JOnAS relies on the JSR 160 specification. When a JOnAS server is started, it creates at least one JSR 160 connector server as described in Chapter 35 *Howto: JSR 160 Support in JOnAS*.

A connector server makes it possible for a JMX remote API client to access and manage the MBeans exposed through the MBean server running in a remote JVM. In order to establish a connection, the remote client needs to know the address of the connector server. JSR 160 does not provide any specific API that would make it possible for a client to find the address of a connector server. The standard suggests using existing discovery and lookup infrastructures; for instance, JNDI API with a LDAP backend.

Currently a new service added to JOnAS, the *Discovery Service*, allows JOnAS servers running over a LAN within the same management domain to communicate to each other the connector-server addresses they create at start-up. All the servers in the domain having the discovery service in the *services* list, will *publish* their connector-server address at start-up. The goal is to allow *master* servers to discover which servers are running in their domain, and to establish connections allowing the remote management of the *slave* servers by a management application deployed on a *master* server.

In the future, the discovery service will evolve towards a more standard implementation in order to allow remote management of servers running on hosts connected to a WAN.

The current discovery service implementation is based on MBeans (called discovery MBeans), which use:

- IP Multicast protocol to communicate their connector-server addresses and state transitions to each other.
- JMX notifications to update the domain management information within a *master* server.

Domain-management information can be retrieved from the `J2EEDomain` MBean and from the JMX service implementation object. Indeed, both `J2EEDomain` MBean and `JmxServiceImpl` instances listen for JMX notifications sent by discovery MBeans.

41.2.1. J2EEDomain MBean

The `J2EEDomain` MBean has the following JMX ObjectName:

```
domainName:j2eeType=J2EEDomain,name=domainName
```

Where *domainName* is the name of the domain.

A `J2EEDomain` MBean contains the following management attributes and operations:

- If the `J2EEDomain` is registered in a *master* server's MBean server, the *servers* attribute keeps a list of the servers running in the domain. Otherwise, it contains only one element corresponding to itself. The elements in the *servers* list are Strings representing `J2EEServer` ObjectNames associated with the running servers. By iterating over the *servers* list, a management application can determine the name of the servers in the domain.

- The *getConnectorServerURLs* operation returns the connector server addresses for a server in the list.

In order to administer a given server in the domain, any management application has to create an `MBeanServerConnection` object using one of the connector server addresses corresponding to that server (see the `j2eemanagement` sample).

41.2.2. Domain Management Information in JMX Service

The JMX service, as the `J2EEDomain` MBean, implements mechanisms that update information about the servers running in the current domain and creates ready-to-use `MBeanServerConnection` connection objects. The JonasAdmin application, for example, uses these connection objects via the `JonasManagementRepr` helper class.

41.3. How to Start Manageable Servers in a Domain

The servers must obey these rules:

- The servers must be started on hosts connected by a LAN.

- The servers must be configured with the discovery service activated. (see Section 3.7 *Configuring the Discovery Service*).

- There must be at least one *master* server in the domain. The role of a server, *slave* or *master* is determined by the discovery service configuration.

- The servers must have different names, but the same domain name.

Note

Also note the following:

- Servers can be started in any order.

- Servers can join or leave the domain at any time. Domain management information is updated on the *master* servers.

- A *slave* server may become *master* by invoking the `startDiscoveryMaster` management operation exposed by the discovery service. This operation can be invoked using the JonasAdmin management console deployed on a *master* server.

41.3.1. Example:

Consider a scenario in which there are three JOnAS servers named *jonas*, *j1* and *j2*. Assume they have discovery service configured with at least one of the servers playing the role of *master*.

1. Start the server named *jonas* in domain *jonas* (by default, the domain name is given by the server's name):
   ```
   jonas start
   ```

2. Start the server named *j1* in domain *jonas*:
   ```
   jonas start -n j1 -Ddomain.name=jonas
   ```

3. Start the server named *j2* in domain *jonas*:
   ```
   jonas start -n j2 -Ddomain.name=jonas
   ```

    Note

   Note that servers may be started in any order.

Chapter 42.

Clustering with JOnAS

This chapter describes how to configure Apache, Tomcat, and JOnAS to install a cluster.

This configuration uses the Apache/Tomcat plug-in `mod_jk2` to provide load balancing and high availability at the JSP/Servlet level. The `mod_jk2` plug-in enables the use of the Apache HTTP server in front of one or more Tomcat JSP/Servlet engines, and provides the capability of forwarding some of the HTTP requests (typically those concerning dynamic pages—such as JSP and Servlet requests) to Tomcat instances.

The configuration uses the In-Memory-Session-Replication technique based on the group communication protocol JavaGroups to provide failover at the Servlet/JSP level.

For load balancing at the EJB level, a clustered JNDI called CMI is used.

42.1. Cluster Architecture

The architecture with all the clustering functionality available in JOnAS is Apache as the front-end HTTP server, JOnAS/Tomcat as J2EE Container, and a shared database.

At the servlet/JSP level, the `mod_jk2` plug-in provides load balancing/high availability, and the tomcat-replication module provides failover.

At the EJB level, the clustered JNDI CMI provides load balancing/high availability.

The database is shared by the JOnAS servers.

The architecture presented in this document is shown in the following illustration:
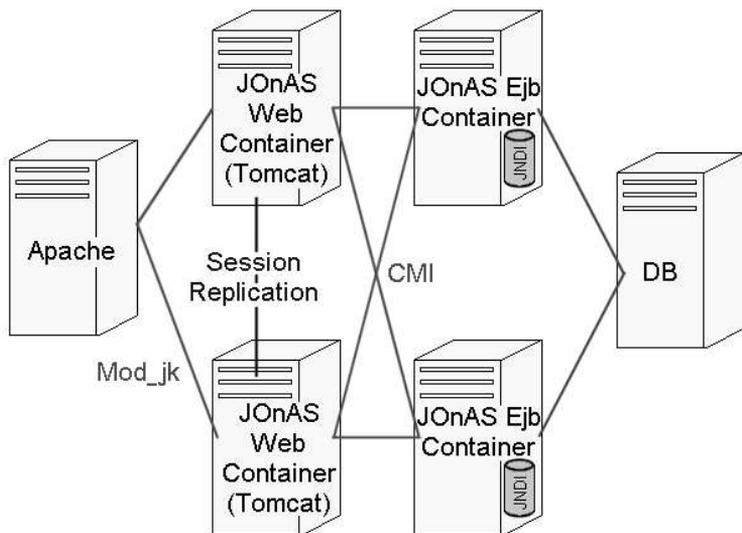
Figure 42-1. Architecture

This architecture provides:

- Load Balancing: Requests can be dispatched over a set of servers to distribute the load. This improves the "scalability" by allowing more requests to be processed concurrently.

- High Availability (HA): having several servers able to fulfill a request makes it possible to ensure that, if a server dies, the request can be sent to an available server (thus the load-balancing algorithm ensures that the server to which the request will be sent is available). Therefore, "Service Availability" is achieved.

- Failover at Servlet/JSP Level: This feature ensures that, if one JSP/Servlet server goes down, another server is able to transparently take over (that is, the request will be switched to another server without service disruption). This means that it will not be necessary to start over, thus achieving continuity.

However, failover at EJB level is not available. This means that no State Replication is provided. The mechanism to provide failover at EJB level is under development and will be available in a coming version of JOnAS.

42.2. Load Balancing at the Web Level with mod_jk2 or Enhydra Director

This section describes how to configure Apache, Tomcat, and JOnAS to run the architecture shown in the following illustration:
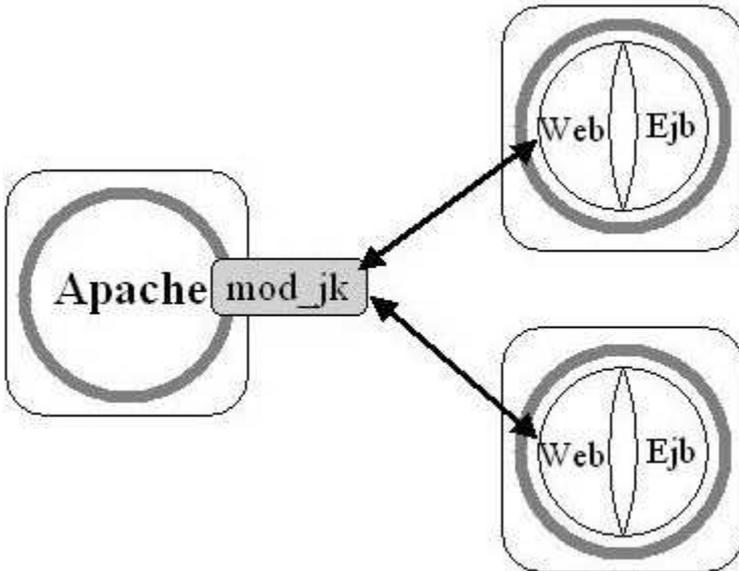
Figure 42-2. Load balancing at the web level with mod_jk2

This example uses mod_jk2 but an alternative configuration using Enhydra Director (http://director.objectweb.org) is also possible. (See Chapter 43 *Load Balancing with Enhydra Director*.)

42.2.1. Configuring the JK Module (mod_jk2)

42.2.1.1. JK module principles

`mod_jk2` is a plug-in that handles the communication between Apache and Tomcat.

`mod_jk2` uses the concept of a *worker*. A worker is a Tomcat instance that is running to perform Servlet requests coming from the web server. Each worker is identified to the web server by the host on which it is located, the port where it listens, and the communication protocol used to exchange messages. In this configuration there is one worker for each Tomcat instance and one worker that will handle the load balancing (this is a specific worker with no host and no port number). All workers are defined in the `workers2.properties` file.

Note:

The JK Module can also be used for site partitioning.

42.2.1.2. Configure Apache

`httpd.conf` should load the following files:

- `AddModule mod_jk2.c`
- `Include $APACHE_HOME/conf/jk/tomcat_jk.conf`
- `LoadModule jk_module libexec/mod_jk2.so`

42.2.1.3. Configure Tomcat

To configure Tomcat, perform the following configuration steps for each Tomcat server.

1. Configure Tomcat for the connector AJP13. In the file `installdir/conf/server.xml`, add (if not already there):
   ```
   <!-- Define an AJP 1.3 Connector on port 9009 -->
   <Connector
       port="9009" minProcessors="5"
       maxProcessors="75"
       acceptCount="10" debug="20" protocol="AJP/1.3"/>
   ```

   Explanations:

   port

   The TCP port number on which this Connector will create a server socket and await incoming connections.

minProcessor

> The minimum number of processors to start at intialization time. If not specified, this attribute is set to 5.

maxProcessor

> The maximum number of processors allowed.

acceptCount

> The maximum queue length for incoming connection requests when all possible requests processing threads are in use. Any requests received when the queue is full will be refused.

debug

> The debugging detail level of log messages generated by this component, with higher numbers creating more detailed output.

protocol

> This attribute must be AJP/1.3 to use the AJP handler.

Note

> Consult the AJP Connector documentation http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/ajp.html.

2. Define the jvmRoute.

In the `installdir/conf/server.xml` file, add a unique route to the Catalina engine. Replace the line:

```
<Engine name="Standalone" defaultHost="localhost"
    debug="0">
```

with:

```
<Engine jvmRoute="worker1" name="Standalone"
    defaultHost="localhost" debug="0">
```

Explanations:

name

> Logical name of this Engine, used in log and error messages.

jvmRoute

> Uniquely identifies the Tomcat server to the Apache server. The name has been specified in the `orkers.properties` file.

defaultHost

> Identifies the Host that will process requests directed to host names on this server.

debug

> The level of debugging detail logged by this Engine.

Note

The jvmRoute name should be the same as the name of the associated worker defined in worker.properties. This will ensure the Session affinity. Consult the Engine Container documentation http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/engine.html.

42.2.2. Configuring JOnAS

In the JOnAS-specific deployment descriptor, add the tag `shared` for the entity beans involved and set it to `true`. When this flag is set to true, multiple instances of the same entity bean in different JOnAS servers can access a common database concurrently.

The following is an example of a deployment descriptor with the flag `shared`:

```
<jonas-ejb-jar>
  <jonas-entity>
    <ejb-name>Id_1</ejb-name>
    <jndi-name>clusterId_1</jndi-name>
    <shared>true</shared>
    <jdbc-mapping>

<jndi-name>jdbc_1</jndi-name>

<jdbc-table-name>clusterIdentityEC</jdbc-table-name>
      <cmp-field-jdbc-mapping>

<field-name>name</field-name>

<jdbc-field-name>c_name</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <cmp-field-jdbc-mapping>

<field-name>number</field-name>

<jdbc-field-name>c_number</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <finder-method-jdbc-mapping>
        <jonas-method>

<method-name>findByNumber</method-name>
        </jonas-method>
        <jdbc-where-clause>where
c_number = ?</jdbc-where-clause>
      </finder-method-jdbc-mapping>
      <finder-method-jdbc-mapping>
        <jonas-method>

<method-name>findAll</method-name>
        </jonas-method>

<jdbc-where-clause></jdbc-where-clause>
      </finder-method-jdbc-mapping>
    </jdbc-mapping>
  </jonas-entity>
</jonas-ejb-jar>
```

42.2.3. Running a Web Application

The web application is now ready to run:

1. Start the jonas servers:
   service jonas start

2. Restart Apache:
   /usr/local/apache2/bin/apachectl restart

3. Use a browser to access the welcome page, usually index.html.

◈ **Note**

The order of launching is important, otherwise it will not work correctly.

42.3. Session Replication at the Web Level

This section shows you how to configure Apache, Tomcat, and JOnAS to run the following architecture:
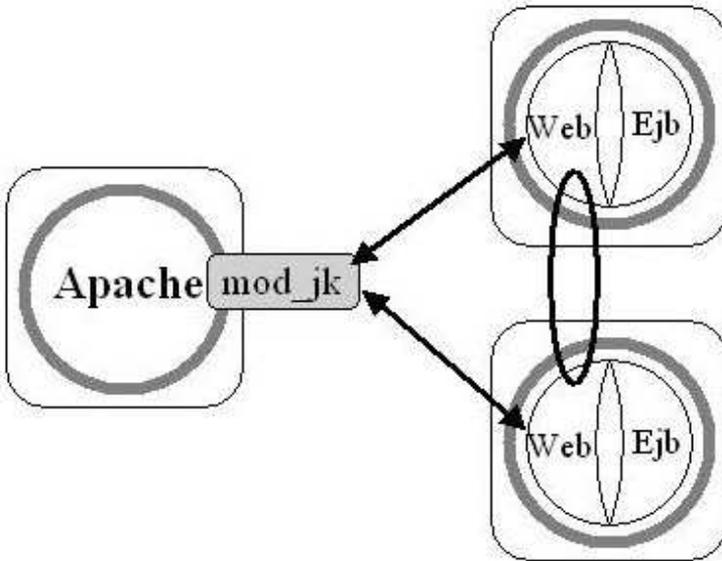
Figure 42-3. Session Replication

The term *session replication* is used when the current service state is being replicated across multiple application instances. Session replication occurs when the information stored in an HttpSession is replicated from, in this example, one Servlet engine instance to another. This could be data such as

items contained in a shopping cart or information being entered on an insurance application. Anything being stored in the session must be replicated for the service to failover without a disruption.

The solution chosen for achieving Session replication is called all-to-all replication. Tomcat uses a proprietary protocol based on TCP for the all-to-all replication.

The follow describes the steps for achieving Session replication with JOnAS.

- The mod_jk2 is used to illustrate the Session Replication. Therefore, first perform the configuration steps presented in the section Section 42.2 *Load Balancing at the Web Level with mod_jk2 or Enhydra Director*.
- On the JOnAS servers, open the `JONAS_BASE/conf/server.xml` file and configure the `<context>` as described:

```
<Cluster className="org.apache.catalina.cluster.tcp.SimpleTcpCluster"
    managerClassName="org.apache.catalina.cluster.session.DeltaManager"
    expireSessionsOnShutdown="false"
    useDirtyFlag="true">

<Membership
    className="org.apache.catalina.cluster.mcast.McastService"
    mcastAddr="228.0.0.4"
    mcastPort="45564"
    mcastFrequency="500"
    mcastDropTime="3000"
    debug = "9"
/>

<Receiver
    className="org.apache.catalina.cluster.tcp.ReplicationListener"
    tcpListenAddress="auto"
    tcpListenPort="4001"
    tcpSelectorTimeout="100"
    tcpThreadCount="6"
/>

<Sender
    className="org.apache.catalina.cluster.tcp.ReplicationTransmitter"
    replicationMode="pooled"
/>

<Valve className="org.apache.catalina.cluster.tcp.ReplicationValve"
    filter=".*\.gif;.*\.js;.*\.jpg;.*\.htm;.*\.html;.*\.txt;"
/>

<Deployer className="org.apache.catalina.cluster.deploy.FarmWarDeployer"
    tempDir="/tmp/war-temp/"
    deployDir="/tmp/war-deploy/"
    watchDir="/tmp/war-listen/"
    watchEnabled="false"
/>
</Cluster>
```

Note

The multicast address and port must be identically configured for all JOnAS/Tomcat instances.

42.3.1. Running your Web Application

The web application is now ready to run in the cluster:

1. Start the JOnAS servers:
   service jonas start

2. Restart Apache: /usr/local/apache2/bin/apachectl restart

3. Use a browser to access the welcome page, usually index.html

42.4. Load Balancing at the EJB Level

This section describes how to configure JOnAS to run the following architecture:
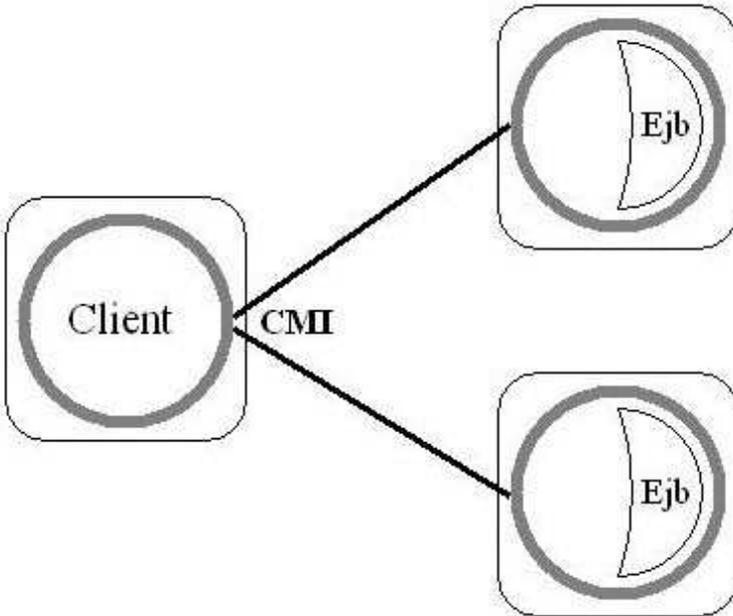
Figure 42-4. Load Balancing

42.4.1. CMI Principles

CMI is a new ORB used by JOnAS to provide clustering for load balancing and high availability. Several instances of JOnAS can be started together in a cluster to share their EJBs. It is possible to start the same EJB on each JOnAS, or to distribute their load. A URL referencing several JOnAS instances can be provided to the clients. At lookup time, a client randomly chooses one of the available servers to request the required bean. Each JOnAS instance has the knowledge (through JavaGroups) of the distribution of the beans in the cluster. An answer to a lookup is a special clustered stub, containing stubs to each instance known in the cluster. Each method call on the home of the bean can be issued

by the stub to a new instance, to balance the load on the cluster. The default algorithm used for load distribution is currently a weighted round-robin.

42.4.2. CMI Configuration

In the case of EJB level clustering (CMI), the client may be either a fat Java client (for example, a Swing application), or a Web application (that is, Servlets/JSPs running within JOnAS). In the second case, the JOnAS server running the Web client should be configured in the same way as the other nodes of the cluster.

- In the `build.properties` of the application, set the protocol name to `cmi` *before* compilation:
  ```
  protocols.names=cmi
  ```
- In the file `carol.properties` of each server (in the directory `$JONAS_BASE/conf`) and of a fat Java client, set the protocol to `cmi`:
  ```
  carol.protocols=cmi
  ```
- In the file `carol.properties` of each server of the cluster, configure the multicast address, the group name, the round-robin weight factor, etc. The following is a configuration example:
  ```
  # java.naming.provider.url property
  carol.cmi.url=cmi://localhost:2002

  # Multicast address used by the registries in the cluster
  carol.cmi.multicast.address=224.0.0.35:35467

  # Groupname for Javagroups
  carol.cmi.multicast.groupname=G1

  # Factor used for this server in weighted round robin algorithms
  carol.cmi.rr.factor=100
  ```
- For a fat Java client, specify the list of registries available in the `carol.properties` file:
  ```
  carol.cmi.url=cmi://server1:port1[,server2:port2...]
  ```

Notes

> The multicast address and group name must be the same for all JOnAS servers in the cluster.

> If Tomcat Replication associated to cmi is used, the multicast addresses of the two configurations must be different.

42.5. Clustering databases with C-JDBC

With C-JDBC it is possible to cluster databases. Refer to Chapter 47 *HowTo Set Up a C-JDBC Database*.

42.6. Preview of a Coming Version

A solution that enables failover at EJB level is currently under development. This signifies state replication for stateful Session Beans and Entity Beans.

This will enable the following architecture:
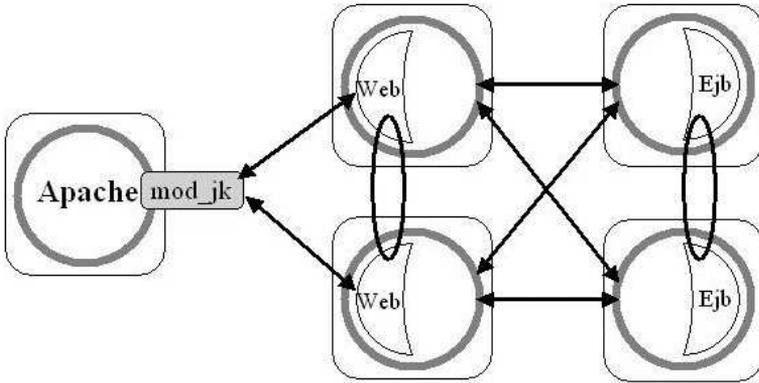
Figure 42-5. Technology Preview

42.7. Used Symbols

	A node (computer) that hosts one or more servers.		
web	A web container.	ejb	An EJB container.
web	A JOnAS instance that hosts a web container.	ejb	A JOnAS instance that hosts an EJB container.
web ejb	A JOnAS instance that hosts a web container and an EJB container.		

	An Apache server with the `mod_jk2` module.		

42.8. References

- Working with mod_jk (http://jakarta.apache.org/tomcat/connectors-doc/index.html)

- Tomcat workers Howto (http://jakarta.apache.org/tomcat/connectors-doc/howto/workers.html)

- Apache JServ Protocol version 1.3 (ajp13) (http://jakarta.apache.org/tomcat/connectors-doc/common/ajpv13a.html)

- Apache - Tomcat HOWTO (http://www.johnturner.com/howto/apache-tomcat-howto.html)

- Tomcat 5 Clustering (http://jakarta.apache.org/tomcat/tomcat-5.0-doc/printer/cluster-howto.html)

Load Balancing with Enhydra Director

1. Download the Tomcat (or Jetty) module from the ObjectWeb site http://director.objectweb.org/index.html.

2. Download the binaries or sources for the Apache module.

> **Note**
>
> You can also download an entire distribution of the Apache webserver in which the module is already included and configured.

3. Configure the Apache director module in the file: `$APACHE_HOME/conf/enhydra_director.conf`

   Example configuration:
   ```xml
   <?xml version="1.0"?>
   <!DOCTYPE EnhydraDirectorConfig SYSTEM "EnhydraDirectorConfig.dtd">
   <EnhydraDirectorConfig>
       <Application prefix="/sampleCluster2/">
           <AppServer host="localhost" port="9999" weight="1" />
           <AppServer host="localhost" port="8888" weight="1" />
       </Application>
       <Status prefix="/status">
           <Restrict server="127.0.0.1" />
           <Restrict client="127.0.0.1" />
       </Status>
   </EnhydraDirectorConfig>
   ```

> **Note**
>
> In this configuration, the connector is configured to forward the requests for one web application (load-balancing and fail-over). The second (sampleCluster2 example) is running on one machine, but on different ports. Furthermore, the `/status` context allows you to enable, disable, and change some parameters of the director configuration. The `/status` context is only available on the machine where director is running in this sample configuration.

4. Copy the tomcat module (`tomcat-director.jar`) to `$CATALINA_HOME/server/lib` or `$JONAS_BASE/lib/catalina/server/lib`.

> **Notes**
>
> • For the JOnAS+Tomcat package, use:`$JONAS_BASE/lib/catalina/server/lib`
>
> • For a standalone Tomcat installation use: `$CATALINA_HOME/server/lib`

5. Configure the connector in the tomcat configuration (`$CATALINE_HOME/conf/server.xml`).

Example configuration:

Add these lines to the configuration where the other connectors (AJP, ...) are defined:

Using TomCat 5.0.x as the servlet container:

```
<!-- Define a Director Connector on port 8888 -->
<Connector
 className =
  "org.enhydra.servlet.connectionMethods.EnhydraDirector. \
   EnhydraDirectorConnectionMethod"
 port = "8888"
 threadTimeout = "300"
 clientTimeout = "30"
 sessionAffinity = "false"
 queueSize = "400"
 numThreads = "200"
 bindAddress = "(All Interfaces)"
 authKey = "(Unauthenticated)"
/>
```

Using TomCat 5.5.x as the servlet container:

```
<!-- Define a Director Connector on port 8888 -->
<Connector
 className =
  "org.enhydra.servlet.connectionMethods.EnhydraDirector. \
   DirectorProtocol"
 port = "8888"
 threadTimeout = "300"
 clientTimeout = "30"
 sessionAffinity = "false"
 queueSize = "400"
 numThreads = "200"
 bindAddress = "(All Interfaces)"
 authKey = "(Unauthenticated)"
/>
```

Note

The session affinity attribute is set to false because the session replication is activated in JOnAS. Otherwise, a new session is created when Director switches between the instances. The Director connector can coexist with the mod_jk connector.

6. Restart Apache.

7. Restart JOnAS.

Setting Up mod_jk with embedded Tomcat

> **Note**
>
> Replace `${JONAS_BASE}` and `${APACHE_HOME}` with appropriate values.

1. Open the file `$JONAS_BASE/conf/server.xml` and uncomment the following line:
   ```
   <!--
   <Connector port="9009" enableLookups="false"
       redirectPort="9043" debug="0"
       protocol="AJP/1.3" />
   -->
   ```

   Edit it to look like the following:
   ```
   <Connector port="9009" enableLookups="false"
       redirectPort="9043" debug="0"
       protocol="AJP/1.3" />
   ```

2. Now you can either let tomcat generate the `mod_jk.conf` file automatically, which will setup forwarding for a limited set of applications, or you can create a custom `mod_jk.conf` file. The automatically generated file cannot be customized since it is re-written every time tomcat is restarted by the user.

   a. To enable automatic generation insert the following into the `server.xml` file nested under the base `<Server>` tag:
      ```
      <Listener className="org.apache.jk.config.ApacheConfig"
          modJk="$APACHE_HOME/modules/mod_jk.so" />
      ```

      and insert the following under the `<Host>` element:
      ```
      <Listener className="org.apache.jk.config.ApacheConfig"
          append="true" forwardAll="false"
          modJk="$APACHE_HOME/modules/mod_jk.so"/>
      ```

      When jonas/tomcat is restarted, there will be a file created under `$JONAS_BASE/conf/auto` called mod_jk.conf.

   b. If you want to create a custom file, the recommendation is to place `mod_jk.conf` under `$JONAS_BASE/conf/jk`. A simple `mod_jk.conf` file that can be used is:
      ```
      # Load the mod_jk module if not loaded.
      <IfModule !mod_jk.c>
      LoadModule jk_module "$APACHE_HOME/modules/mod_jk.so"
      </IfModule>

      # Specify location of worker file and log file.
      # Worker file will follow shortly.
      JkWorkersFile "$JONAS_BASE/conf/jk/workers.properties"
      JkLogFile "$JONAS_BASE/logs/mod_jk.log"

      # When and how much logging.
      JkLogLevel emerg
      ```

```
# This is a little awkward. It seems mod_jk associates
# applications that map to tomcat based on the virtual host.
# If, for instance, I wish to visit the jonasAdmin application
# through http://jonas-server/jonasAdmin from another machine
# and I have the following setting then the application behaves
# perfectly normally; that is, struts kicks in as expected,
# form-based authentication and forwarding is done exactly as
# expected if you were using the application directly using
# the appropriate port (9000). However, if you try using
# http://localhost/jonasAdmin from jonas-server without the
# explicit VirtualHost declaration, only the directory contents
# are mapped. So we need to explicitly mention both virtual hosts.
<VirtualHost jonas-server>
    ServerName jonas-server
    JkMount /olstore ajp13
    JkMount /olstore/* ajp13
    JkMount /jonasAdmin ajp13
    JkMount /jonasAdmin/* ajp13
</VirtualHost>
# ajp13 is in fact the worker name used in workers.properties.
<VirtualHost localhost>
    ServerName localhost
    JkMount /olstore ajp13
    JkMount /olstore/* ajp13
    JkMount /jonasAdmin ajp13
    JkMount /jonasAdmin/* ajp13
</VirtualHost>
```

3. To tell apache to use this file, edit the `$APACHE_HOME/conf/httpd.conf` file and insert:
   ```
   Include ${JONAS_BASE}/conf/jk/mod_jk.conf
   ```

   Note

   Replace ${JONAS_BASE} with appropriate value.

4. Create a workers.properties file. This can be placed in `$JONAS_ROOT/conf/jk/workers.properties`.

   A sample that has been used successfully is:
   ```
   # workers.java_home should point to your Java installation.
   # Normally you should have a /bin and a /lib beneath it.
   #
   workers.java_home=/usr/lib/jvm/java

   #
   # You should configure your environment slash... ps=\ on NT
   # and / on UNIX and possibly something different elsewhere.
   #
   ps=/

   #
   #------ ADVANCED MODE ------------------------------------
   #--------------------------------------------------------
   #


   #
   #------ DEFAULT worker list -----------------------------
   #--------------------------------------------------------
   ```

```
#
#
# The workers that your plugins should create and work with
#
# Add 'inprocess' if you want a JNI connector
worker.list=ajp13
# , inprocess


#
#------ DEFAULT worker1 WORKER DEFINITION ----------------
#--------------------------------------------------------
#


#
#!
Defining a worker named worker1 and of type worker1
# Note that the name and the type do not have to match.
#
worker.ajp13.port=9009
worker.ajp13.host=jonas-server
worker.ajp13.type=ajp13


#--------------------------------------------------------
```

5. Once all this is completed, restart httpd and ensure that jonas/tomcat is up. You can follow that up with a quick test by visiting http://*host_name*/jonasAdmin. You should now be able to use the application as normal.

Setting Up mod_jk2 with Embedded Tomcat

Note

Replace `${JONAS_BASE}` and `${APACHE_HOME}` with appropriate values.

1. Back up `/$APACHE_HOME/conf/httpd.conf`:
   ```
   cp $APACHE_HOME/conf/httpd.conf $APACHE_HOME/conf/httpd.conf.backup
   ```

2. Edit (~line 208) `httpd.conf` to add the loading of mod_jk2.so:
   ```
   # Adding jk2_module reference.
   LoadModule jk2_module modules/mod_jk2.so
   ```

3. Edit `$APACHE_HOME/conf.d/JOnAS.conf` with the following:
   ```
   Alias /docs/jonas "$JONAS_BASE/doc/jonas-4.1.2"
   <Directory "$JONAS_BASE/doc/jonas-4.1.2">
       Options Indexes MultiViews
       AllowOverride None
       Order allow,deny
       Allow from all
   </Directory>

   Alias /webapps/jonas "$JONAS_BASE/webapps/jonas"
   <Directory "$JONAS_BASE/webapps/jonas">
       Options Indexes MultiViews
       AllowOverride None
       Order allow,deny
       Allow from all
   </Directory>

   <IfModule mod_jk2.c>

   <Location /earsample>
       JkUriSet group jonas
   </Location>

   <Location /jonasAdmin>
       JkUriSet group jonas
   </Location>

   <Location /cmp2>
       JkUriSet group jonas
   </Location>

   <Location /alarm>
       JkUriSet group jonas
   </Location>

   </IfModule>
   ```

4. Edit/create `$APACHE_HOME/conf/workers2.properties` with the following:

```
[logger]
level=DEBUG

# Shared memory handling. Needs to be set.
[shm]
info=Scoreboard.
# Required for reconfiguration and status with
# multiprocess servers
file=$APACHE_HOME/logs/jk2.shm
size=1048576
debug=0
disabled=0

# The channel configuration shall be consistent with
# the configuration of JOnAS in server.xml
# port = port of the AJP (jk2) connector
# tomcatId = jvmRoute attribute of the <Engine> element
[channel.socket:toJonas01]
info=channel to a JOnAS instance
# tomcatId shall be identical to jvmRoute in the server.xml
# file of the JOnAS instance
host=localhost
port=9009
group=jonas

[status:status]
info=provides info on the connecteur usage

[uri:$/jkstatus*]
info=get the connector usage info at /jkstatus
group=status:status
```

5. On the JOnAS side, edit `$JONAS_BASE/conf/server.xml` with the following content:

```
...
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="9009" minProcessors="5" maxProcessors="75"
    enableLookups="true" redirectPort="9043"
    acceptCount="10" debug="0" connectionTimeout="0"
    useURIValidationHack="false"
    protocolHandlerClassName="org.apache.jk.server.JkCoyoteHandler"/>
...
<Engine name="Tomcat-JOnAS-Engine" defaultHost="localhost" debug="0"
    jvmRoute="jonas01">
```

6. Comment out all lines in `$JONAS_BASE/conf/jk2.properties`.

7. Restart jonas and httpd services:

```
service jonas restart; service httpd restart
```

8. Deploy `cmp2.ear` and `alarm.ear`.

   Those applications are available at:

   • http://<hostname>/cmp2

   • http://<hostname>/earsample

   • http://<hostname>/alarm

   • http://<hostname>/jonasAdmin

Chapter 46.

Migrating from mod_jk2 to mod_jk with embedded Tomcat

This document shows one of several ways to migrate mod_jk2 settings to work with mod_jk. The mod_jk2 reference setup can be found in Chapter 45 *Setting Up mod_jk2 with Embedded Tomcat*.

Notes

- All references to jonas-server should be replaced by the hostname of the target machine for which you want to configure mod_jk. Replace ${JONAS_BASE} and ${APACHE_HOME} with appropriate values.
- Always back up files such as `server.xml` before editing!

46.1. Performing the Migration

1. Make a backup copy of the `$JONAS_BASE/conf/server.xml` file.

2. Edit the `$JONAS_BASE/conf/server.xml` file; locate and uncomment the following line:

   ```
   <!--
   <Connector port="9009"
       enableLookups="false" redirectPort="9043" debug="0"
       protocol="AJP/1.3" />
   -->
   ```

   Change it to look like this:

   ```
   <Connector port="9009"
       enableLookups="false" redirectPort="9043" debug="0"
       protocol="AJP/1.3" />
   ```

3. Next, comment out the existing mod_jk2-oriented connector and engine elements:

   ```
   ...
   <Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
       port="9009" minProcessors="5" maxProcessors="75"
       enableLookups="true" redirectPort="9043"
       acceptCount="10" debug="0" connectionTimeout="0"
       useURIValidationHack="false"
       protocolHandlerClassName="org.apache.jk.server.JkCoyoteHandler"/>
   ```

   Replace the base Engine tag in which a jvmRoute is used with the standard engine. For example, comment out the line that looks similar to the following:

   ```
   <Engine name="Tomcat-JOnAS-Engine" defaultHost="localhost" debug="0"
       jvmRoute="jonas01">
   ```

   and uncomment the following line:

   ```
   <Engine name="Tomcat-JOnAS-Engine" defaultHost="localhost" debug="0">
   ```

4. Now you can either let tomcat generate the `mod_jk.conf` file automatically, which will set up forwarding for a limited set of applications, or you can create a custom `mod_jk.conf` file where you can specify applications to forward and customize the configuration information.

Note

The automatically generated file cannot be customized because it is re-written every time the user restarts tomcat.

a. To enable automatic generation, insert the following into the `server.xml` file nested under the base `<Server>` tag:

```
<Listener className="org.apache.jk.config.ApacheConfig"
    modJk="$APACHE_HOME/modules/mod_jk.so" />
```

and insert the following under the `<Host>` element:

```
<Listener className="org.apache.jk.config.ApacheConfig"
    append="true" forwardAll="false"
    modJk="$APACHE_HOME/modules/mod_jk.so"/>
```

When jonas/tomcat is restarted, there will be a file created under `$JONAS_BASE/lib/catalina/conf/auto` called `mod_jk.conf`.

b. If you want to create a custom file, place a file named `mod_jk.conf` under `$JONAS_BASE/conf/jk`. This file is the equivalent of the `$APACHE_HOME/conf.d/JOnAS.conf` in the mod_jk2 directions; however, it is not a direct replacement.

A simple `mod_jk.conf` file that can be used is:

```
# Load the mod_jk module if not loaded.
<IfModule !mod_jk.c>
LoadModule jk_module "$APACHE_HOME/modules/mod_jk.so"
</IfModule>

# Specify location of worker file and log file.
# Worker file will follow shortly.
JkWorkersFile "$JONAS_BASE/conf/jk/workers.properties"
JkLogFile "$JONAS_BASE/logs/mod_jk.log"

# When and how much logging.
JkLogLevel emerg

# This is a little awkward. It seems mod_jk associates
# applications it will map to tomcat based on the virtual host.
# If for instance I wish to visit the jonasAdmin application
# through http://jonas-server/jonasAdmin from another machine
# and I have the following setting then the application behaves
# perfectly normally, that is, behaves as one would expect if you
# were using the application directly using the appropriate
# port (9000).
# However, if you try using http://localhost/jonasAdmin from
# jonas-server without the explicit VirtualHost declaration,
# only the directory contents are mapped. No processing on tomcat
# is carried out, so struts-based applications will fail to work.
# That is the reason why we explicitly mention both virtual hosts.
<VirtualHost jonas-server>
    ServerName jonas-server
    JkMount /cmp2 ajp13
    JkMount /cmp2/* ajp13
    JkMount /alarm ajp13
    JkMount /alarm/* ajp1
    JkMount /earsample ajp13
    JkMount /earsample/* ajp13
    JkMount /jonasAdmin ajp13
    JkMount /jonasAdmin/* ajp13
</VirtualHost>
```

```
<VirtualHost localhost>
    ServerName localhost
    JkMount /cmp2 ajp13
    JkMount /cmp2/* ajp13
    JkMount /alarm ajp13
    JkMount /alarm/* ajp13
    JkMount /earsample ajp13
    JkMount /ears!
    mple/* ajp13
    JkMount /jonasAdmin ajp13
    JkMount /jonasAdmin/* ajp13
</VirtualHost>
```

c. Now tell apache to use this file. Back up the `$APACHE_HOME/conf/httpd.conf` file. Edit the `/etc/httpd/conf/httpd.conf` file and insert:
```
Include ${JONAS_BASE}/conf/jk/mod_jk.conf
```

> **Note**
>
> Replace $JONAS_BASE} with the appropriate value.

Locate and comment-out the following line in the same file:
```
LoadModule jk2_module modules/mod_jk2.so
```

d. Now create a `workers.properties` file. This can be placed in `$JONAS_BASE/conf/jk/workers.properties`. A sample that has been used successfully is:
```
# workers.java_home should point to your Java installation.
# Normally you should have a bin and lib directories beneath it.
#
workers.java_home=/usr/lib/jvm/java

#
# You should configure your environment slash... ps=\ on NT
# and / on UNIX and maybe something different elsewhere.
#
ps=/

#
#------ ADVANCED MODE ------------------------------------------
#--------------------------------------------------------------
#

#
#------ DEFAULT worker list -----------------------------------
#--------------------------------------------------------------
#
#
# The workers that your plugins should create and work with
#
# Add 'inprocess' if you want a JNI connector
worker.list=ajp13
# , inprocess


#
#------ DEFAULT worker1 WORKER DEFINITION --------------------
#--------------------------------------------------------------
#
```

```
#
# Defining a worker named worker1 and of type worker1
# Note that the name and the type do not have to match.
#
worker.ajp13.port=9009
worker.ajp13.host=jonas-server
worker.ajp13.type=ajp13


#-------------------------------------------------------
```

e. Once all this is completed, restart jonas, then httpd.

f. Follow that up with a quick test by visiting: http://${HOSTNAME}/jonasAdmin

You should be able to use the application as normal. You can now also use cmp2, earsample, alarm and jonasAdmin using mod_jk instead.

redhat.

HowTo Set Up a C-JDBC Database

> **Note**
>
> This documentation describes the configuration of a clustered database in a RAID1. For further infor-
> mation, refer to the ObjectWeb C-JDBC documentation http://c-jdbc.objectweb.org/doc/index.html.

This configuration starts three HSQL instances, which are formed by C-JDBC as a RAID1. A RAID1
stores the same data in two different databases and uses a third database as a recovery log.

1. Download the C-JDBC binaries from the C-JDBC site (http://c-jdbc.objectweb.org).

2. Extract the C-JDBC archive into a folder:
   ```
   tar xvfz c-jdbc-xx-bin.tar.gz
   ```

3. Set the environment variables:
   ```
   export JAVA_HOME=<Path_to_JDK_HOME>
   export CJDBC_HOME=<Path_to_CJDBC_HOME>
   ```

4. Modify the `$CJDBC_HOME/demo/demo-raidb1.sh` file.

   The ports for the HSQLDB instances have to be modified in order to match the sampleCluster2.
   Change the ports for the three databases to `11001`, `11002`, `11003`.

5. Configure the C-JDBC controller file:

   a. Modify the ports in `$CJDBC_HOME/config/virtualdatabase/hsqldb-raidb1.xml`
      according to values defined in the section above. To avoid problems in the Datasource
      configuration of JOnAS, you must also set a password with which the users can log in.

   b. Back up the original configuration:
      ```
      mv $CJDBC_HOME/config/controller/controller.xml \
          $CJDBC_HOME/config/controller/controller.xml.original
      ```

   c. Create a link for the new configuration:
      ```
      ln -s $CJDBC_HOME/config/controller/controller-raidb1.xml \
          $CJDBC_HOME/config/controller/controller.xml
      ```

   d. Modify the file `$CJDBC_HOME/config/controller/controller.xml`
      to specify the configuration file of the virtual database, which is
      `$CJDBC_HOME/config/virtualdatabase/hsqldb-raidb1.xml`

   e. Start the HSQL instances:
      ```
      $CJDBC_HOME/demo/demo-raidb1.sh
      ```

   f. Start the C-JDBC controller:
      ```
      $CJDBC_HOME/bin/controller.sh
      ```

   g. Connect to the C-JDBC controller using the JDBC driver in `$CJDBC_HOME/drivers` and
      the URL `jdbc:cjdbc://localhost:25322/myDB` with:

      • username: `<user>`

      • password: `<pass>`

Note

No password is specified by default, but you should specify one in the $CJDBC_HOME/config/virtualdatabase/hsqldb-raidb1.xml file.

47.1. Sample Configuration of a C-JDBC Database in JOnAS

```
####################### C-JDBC DataSource configuration example ######
DataSource configuration
#
datasource.name jdbc_1
datasource.url jdbc:cjdbc://localhost:25322/myDB
datasource.classname org.objectweb.cjdbc.driver.Driver
datasource.username user
datasource.password pass
datasource.mapper rdb.hsql
# Note: You must specify the mapper of the databases used as backend
# by C-JDBC.
# A heterogeneous C-JDBC configuration cannot be used here!
```

Distributed Message Beans in JOnAS 4.1

JOnAS release 4.1 dramatically simplifies the use of a distributed JORAM platform from within JOnAS servers. Such a configuration allows, for example, a bean hosted by JOnAS instance "A" to send messages on a JORAM queue, to which a Message-Driven Bean (MDB) hosted by JOnAS instance "B" listens.

The reasons for this progress are:

- The JORAM Resource Adapter allows much finer configuration than the JMS service did.
- JORAM provides a distributed JNDI server that enables JOnAS instances to share information.

> **Note**
>
> Before you proceed with this chapter, you should review Section 3.8.1 *JORAM Resource Adapter*.

48.1. Scenario and General Architecture

The following scenario and general settings are proposed:

- Two instances of JOnAS are run (JOnAS "A" and JOnAS "B"). JOnAS A hosts a simple bean that provides a method for sending a message on a JORAM queue. JOnAS B hosts a Message-Driven Bean that listens on the same JORAM queue.
- Each JOnAS instance has a dedicated, collocated JORAM server: server "s0" for JOnAS A, "s1" for JOnAS B. Those two servers are aware of each other.
- The queue is hosted by JORAM server s1.
- An additional JNDI service is provided by the JORAM servers. This service is used for storing the shared information (basically, the queue's naming reference).

48.2. Common Configuration

The JORAM servers are part of a single JORAM platform, described by the following `a3servers.xml` configuration file:

```
<?xml version="1.0"?>
<config>
   <domain name="D1"/>
   <server id="0" name="S0" hostname="hostA">
      <network domain="D1" port="16301"/>
      <service
        class="org.objectweb.joram.mom.proxies.ConnectionManager"
        args="root root"/>
      <service
        class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
        args="16010"/>
      <service
        class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
```

```
         args="16400 0"/>
   </server>
   <server id="1" name="S1" hostname="hostB">
      <network domain="D1" port="16301"/>
      <service
         class="org.objectweb.joram.mom.proxies.ConnectionManager"
         args="root root"/>
      <service
         class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
         args="16010"/>
      <service
         class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
         args="16400 0 1"/>

   </server>
</config>
```

This configuration describes a platform made of two servers, "s0" and "s1", hosted by machines "hostA" and "hostB", listening on ports 16010, providing a distributed JNDI service (more information on JORAM's JNDI can be found at http://joram.objectweb.org/current/doc/joram4_0_JNDI.pdf).

Each JOnAS server must hold a copy of this file in its `conf/` directory. In their respective `jonas.properties` files, each must declare `joram_for_jonas_ra.rar` as a resource to be deployed and each should remove `jms` from its list of services.

48.3. Specific Configuration

`JOnAS A` embeds JORAM server `s0`. The `jonas-ra.xml` descriptor packaged in the `joram_for_jonas_ra.rar` archive file must provide the following information:

```
<jonas-config-property>
   <jonas-config-property-name>
      HostName
   </jonas-config-property-name>
   <jonas-config-property-value>
      hostA
   </jonas-config-property-value>
</jonas-config-property>
```

The other default settings do not need to be changed.

`JOnAS B` embeds JORAM server `s1`. The `jonas-ra.xml` descriptor packaged in the `joram_for_jonas_ra.rar` archive file must provide the following properties values:

```
<jonas-config-property>
   <jonas-config-property-name>
      ServerId
   </jonas-config-property-name>
   <jonas-config-property-value>
      1
   </jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
   <jonas-config-property-name>
      ServerName
   </jonas-config-property-name>
   <jonas-config-property-value>
      s1
   </jonas-config-property-value>
</jonas-config-property>
```

```
<jonas-config-property>
    <jonas-config-property-name>
        HostName
    </jonas-config-property-name>
    <jonas-config-property-value>
        hostB
    </jonas-config-property-value>
</jonas-config-property>
```

The other default settings do not need to be changed.

The *shared queue* is hosted by JORAM server s1. You then need to declare it in JOnAS B's `joramAdmin.xml` file as follows:

```
<Queue name="sharedQueue">
  <freeReader/>
  <freeWriter/>
  <jndi name="scn:comp/sharedQueue"/>
</Queue>
```

The `scn:comp/` prefix is a standard way to specify which JNDI provider should be used. In this case, the shared queue is bound to JORAM's distributed JNDI server, and may be retrieved from both JOnAS A and JOnAS B. To provide this mechanism, both JOnAS servers must provide access to a standard `jndi.properties` file. For JOnAS A, the file looks as follows, and should be put in its `conf/` directory:

```
java.naming.factory.url.pkgs
   org.objectweb.jonas.naming:fr.dyade.aaa.jndi2
scn.naming.factory.host          hostA
scn.naming.factory.port          16400
```

For JOnAS B, the file looks as follows, and should be put in the right `conf/` directory:

```
java.naming.factory.url.pkgs
   org.objectweb.jonas.naming:fr.dyade.aaa.jndi2
scn.naming.factory.host          hostB
scn.naming.factory.port          16400
```

48.4. The Beans

The *simple bean* on JOnAS A needs to connect to its local JORAM server and access the remote queue. The following is an example of consistent resource definitions in the deployment descriptors:

Standard deployment descriptor:

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

Specific deployment descriptor:

```
<jonas-resource>
  <res-ref-name>jms/factory</res-ref-name>
  <jndi-name>CF</jndi-name>
</jonas-resource>
<jonas-resource-env>
  <resource-env-ref-name>
     jms/sharedQueue
  </resource-env-ref-name>
  <jndi-name>scn:comp/sharedQueue</jndi-name>
</jonas-resource-env>
```

The ConnectionFactory is retrieved from the local JNDI registry of the bean. However, the Queue is retrieved from the distributed JORAM JNDI server because its name starts with the `scn:comp/` prefix. It is the same queue to which the Message-Driven Bean on JOnAS B listens. To do this, its activation properties should be set as follows:

```
<activation-config>
   <activation-config-property>
      <activation-config-property-name>
         destination
      </activation-config-property-name>
      <activation-config-property-value>
         scn:comp/sharedQueue
      </activation-config-property-value>
   </activation-config-property>
   <activation-config-property>
      <activation-config-property-name>
         destinationType
      </activation-config-property-name>
      <activation-config-property-value>
         javax.jms.Queue
      </activation-config-property-value>
   </activation-config-property>
</activation-config>
```

Chapter 49.

How to use Axis in JOnAS

This chapter describes basic Axis use within JOnAS. It assumes that you do not require any explanation about Axis-specific tasks (for example, Axis deployment with WSDD). Before deployment in Axis, you must verify that the deploy.wsdd file *matches the site machine configuration* (the jndiURL parameter in particular):

```
<parameter name="jndiURL" value="rmi://localhost:1099"/>
```

This chapter describes two ways to make an EJB (stateless Session Bean (SB)) available as a Web Service with JOnAS:

- Axis runs in a unique Webapp, the SB is packaged in a separate EJB-JAR (or even EAR). The intent of this approach is to make EJBs from different packages that are *already deployed* accessible as Web Services via a single Axis Webapp deployment. The drawback is that Web Services are *centralized* in one Webapp only and the only way to distinguish between them for access is by the <service-name>, not by the <context-root>/<service-name>. In addition, the EJB-JAR files that contain the Web Services *must be included in the Webapp*.

- The accessed EJB(s) are packaged with the Axis Webapp in an EAR archive. With this approach, *the EJB-JAR files do not have to be included* in the Webapp WEB-INF/lib directory; different Applications that contain Web Services can be hosted, providing the capability of distinguishing between Web Services of different applications.

49.1. Unique Axis Webapp

49.1.1. Constraints

- The EJBs exposed as WebServices must have *remote interfaces*.
- The Axis Webapp must have in its WEB-INF/lib directory all the EJB-JAR files containing Beans exposed as Web Services.

49.1.2. Usage

- Deploy the EJB-JARs or EARs containing Web Services.
- Deploy the Axis Webapp (containing the EJB-JAR files).
- Use the AdminClient tool to deploy the Web Services (with a .wsdd file).

Example:

```
jclient org.apache.axis.client.AdminClient –hjonasServerHostname
    –p9000 deploy.wsdd
```

49.2. Embedded Axis Webapp

49.2.1. Constraints

- The EJBs exposed as Web Services can have either *local or remote interfaces*.
- The EAR must contain a Webapp including a `web.xml` with Axis servlet mapping.

49.2.2. Usage

- Deploy the application archive (EAR):
- Use the AdminClient tool to deploy the webservices (with a `.wsdd` file)

Example:

```
jclient org.apache.axis.client.AdminClient
    -lhttp://localhost:9000/hello/servlet/AxisServlet deploy.wsdd
```

Be careful to use a good URL to reach the Axis Servlet.

Refer to the `embedded_axis` example (in the `$JONAS_ROOT/examples` directory).

49.3. Axis Tests

When everything is deployed and running, use the following URL to view the deployed Web Services:

```
http://yourserver:port/yourwebapp/servlet/AxisServlet
```

This page will display a link for each Web Service with the `WSDL file` (automatically generated by Axis from the Java Interfaces). Use the following URL to access your Web Service (add ?WSDL for the associated WSDL file):

```
http://yourserver:port/yourwebapp/services/<Service-name>
```

A client class can now be run against the Web Service. Note that any language (with Web Services capabilities) can be used for the client (C#, Java, etc.).

49.4. Axis Tools

Use jclient to deploy your Web Services (in the Axis way):

```
jclient org.apache.axis.client.AdminClient [OPTIONS] <WSDD-file>
```

49.4.1. Options

```
-l
```

      `<URL>`:   the   location   of   the   AxisServlet   servlet   (the   default   location   is:
http://localhost:9000/axis/servlet/AxisServlet)

`-p`

    `<port>`: the port of the listening http daemon (default : 9000)

`-h`

    `<host>`: the hostname of the server running the JOnAS server (default : localhost)

Chapter 50.

Using WebSphere MQ JMS

WebSphere MQ (formerly MQSeries) is the messaging platform developed by IBM; it provides Java and JMS interfaces. Section 3.5.10 *Configuring the JMS Service* has information on the JMS service.

This chapter explains how WebSphere MQ can be used as a JMS provider within a JOnAS application server.

Refer to http://www-3.ibm.com/software/integration/mqfamily/library/manualsa/ for WebSphere MQ documentation.

50.1. Architectural Rules

WebSphere MQ, in contrast to JORAM and SwiftMQ, cannot run collocated with JOnAS. WebSphere MQ is an external software package that must be independently administered and configured.

Administering WebSphere MQ consists of the following steps:

- Creating and configuring resources (such as queues) through the WebSphere MQ Explorer tool.
- Creating the corresponding JMS objects (`javax.jms.Queue`, `javax.jms.Topic`, `javax.jms.QueueConnectionFactory`, etc.), and binding them to a registry.

The link between JOnAS and WebSphere MQ is established through the JOnAS registry. WebSphere MQ JMS objects are bound to the JOnAS registry. JMS lookups then return the WebSphere MQ JMS objects, and messaging takes place through these objects.

Given the complex configuration of WebSphere MQ JMS objects, it is not possible to create these objects from JOnAS. Therefore, during the starting phase, a JOnAS server expects WebSphere MQ JMS objects to have already been bound to the registry. It thus becomes necessary to start an independent registry, to which WebSphere MQ may bind its JMS objects, and which may also be used by the starting JOnAS server. The start-up sequence looks as follows:

1. Starting a registry.
2. Creating and binding WebSphere MQ JMS objects.
3. Launching the JOnAS server.

The following architecture is recommended:

- A JOnAS server (for example called "Registry") that provides only a registry.
- A JOnAS server (for example called "EJB") using the registry service of server "Registry".
- Plus, of course, a WebSphere MQ server running locally.

50.2. Setting the JOnAS Environment

The suggested architecture requires two JOnAS server instances. You can do this as follows:

1. Create two base directories. For example `JONAS_REGISTRY` and `JONAS_EJB`.
2. Set the JONAS_BASE environment variable so that it points towards the `JONAS_REGISTRY` directory.

3. In the `$JONAS_ROOT` directory, type: **`ant create_jonasbase`**

4. Set the JONAS_BASE environment variable so that it points towards the `JONAS_EJB` directory.

5. In the `$JONAS_ROOT` directory, type: **`ant create_jonasbase`**

The JOnAS servers may now be configured independently.

50.2.1. Configuring the Registry Server

The "Registry" server is the JOnAS server that hosts the registry service. Its configuration files are in `JONAS_REGISTRY/conf`.

In the `jonas.properties` files, declare only the registry and jmx services:

```
jonas.services     registry,jmx
```

In the carol.properties file, declare the rmi protocol:

```
carol.protocols=rmi
```

You can also configure its port:

```
carol.rmi.url=jrmi://localhost:2000
```

50.2.2. Configuring the EJB Server

The "EJB" server is the JOnAS server that is used as the application server. Its configuration files are in `JONAS_EJB/conf`. Libraries must be added in `JONAS_EJB/lib/ext`.

In the jonas.properties files, set the registry service as remote:

```
jonas.service.registry.mode     remote
```

In the carol.properties file, declare the rmi protocol and set the correct port:

```
carol.protocols=rmi
carol.rmi.url=jrmi://localhost:2000
```

In `lib/ext`, the following libraries must be added:

- `com.ibm.mqjms.jar`, including WebSphere MQ JMS classes.
- `com.ibm.mq.jar`, also a WebSphere MQ library.

50.3. Configuring WebSphere MQ

WebSphere MQ JMS administration is documented in the WebSphere MQ Using Java document.

The configuration file of the JMS administration tool must be edited so that the JOnAS registry is used for binding the JMS objects. This file is the `JMSAdmin.config` file located in WebSphereMQ's `Java/bin` directory. Set the factory and provider URL as follows:

```
INITIAL_CONTEXT_FACTORY=
  org.objectweb.rmi.libs.services.registry.jndi.JRMIInitialContextFactory
PROVIDER_URL=jrmi://localhost:2000
```

You also need to add JOnAS's `client.jar` library to WebSphere MQ's classpath.

When starting, JOnAS expects JMS objects to have been created and bound to the registry. Those objects are connection factories, needed for connecting to WebSphere MQ destinations and other destinations.

JOnAS automatically tries to access the following factories:

- An `XAConnectionFactory`, bound with name "wsmqXACF".
- An `XAQueueConnectionFactory`, bound with name "wsmqXAQCF".
- An `XATopicConnectionFactory`, bound with name "wsmqXATCF".
- A `ConnectionFactory`, bound with name "JCF".
- A `QueueConnectionFactory`, bound with name "JQCF".
- A `TopicConnectionFactory`, bound with name "JTCF".

If one of these objects cannot be found, JOnAS prints a message similar to this:

```
JmsAdminForWSMQ.start : WebSphere MQ XAConnectionFactory
  could not be retrieved from JNDI
```

This does not prevent JOnAS from working. However, if no connection factory is available, no JMS operations are possible from JOnAS.

If destinations have been declared in the `jonas.properties` file, JOnAS also expects to find them. For example, if the following destinations are declared:

```
jonas.service.jms.topics    sampleTopic
jonas.service.jms.queues    sampleQueue
```

The server expects to find the following JMS objects in the registry:

- A `Queue`, bound with name "sampleQueue".
- A `Topic`, bound with name "sampleTopic".

If one of the declared destinations cannot be retrieved, the following message appears and the server stops:

```
JOnAS error: org.objectweb.jonas.service.ServiceException:
 Cannot init/start service jms':
 org.objectweb.jonas.service.ServiceException :
 JMS Service Cannot create administered object: java.lang.Exception:
 WebSphere MQ Queue creation impossible from JOnAS
```

Contrary to connection factories, the JOnAS administration tool enables you to create destinations. As it is not possible to create WebSphere MQ JMS objects from JOnAS, this works only if the destinations are previously created through WebSphere MQ and bound to the registry.

For example, an attempt to create a queue named "myQueue" through the JonasAdmin tool works only if:

- You have created a queue through the WebSphere MQ Explorer tool.
- You have created the corresponding JMS `Queue` and bound it to the registry with the name "myQueue".

To launch WebSphere MQ administration tool, enter:

JMSAdmin

The following prompt appears: `InitCtx>`

To create a `QueueConnectionFactory` and bind it with the name `JQCF`, enter:

`InitCtx> DEF QCF(JQCF)`

You can add more parameters (for example, to specify the queue manager).

To create a `Queue` that represents a WebSphere MQ queue named `myWSMQqueue` and bind it with the name `sampleQueue`, enter:

`InitCtx> DEF Q(sampleQueue) QUEUE(myWSMQqueue)`

You can view objects bound in the registry by entering:

`InitCtx> DIS CTX`

50.4. Starting the Application

Starting the registry server:

1. Clean the local CLASSPATH: `set/export CLASSPATH=""`
2. Set the JONAS_BASE variable so that it points towards `JONAS_REGISTRY`.
3. Start the JOnAS server: `jonas start -n Registry`

Administering WebSphere MQ:

1. In WebSphere MQ's `Java/bin` directory, launch the JMSAdmin tool: **`JMSAdmin`**
2. Create the required JMS objects.

Starting the EJB server:

1. Clean the local CLASSPATH: `set/export CLASSPATH=""`
2. Set the JONAS_BASE variable so that it points towards `JONAS_EJB`.
3. Start the JOnAS server: `jonas start -n EJB`

Starting an EJB client:

1. Add in the jclient classpath the `ibm.com.mq.jar` and `ibm.com.mqjms.jar` libraries.
2. Launch the client: `jclient ...`

50.5. Limitations

Using WebSphere MQ as the JMS transport within JOnAS has some limitations compared to using JORAM or SwiftMQ.

First of all, WebSphere MQ is compliant with the old 1.0.2b JMS specifications. Code written following the JMS 1.1 specification (such as the JMS samples provided with JOnAS) will not work with WebSphere MQ.

Depending on the WebSphere MQ distribution, JMS Publish/Subscribe may not be available. In this case, the Message-Driven Bean samples provided with JOnAS will not work.

Web Service Interoperability between JOnAS and BEA WebLogic

This chapter describes the basic use of web services between JOnAS and WebLogic Server. It assumes that the reader does not require any explanation about Axis-specific tasks (Axis deployment with WSDD, etc.). Before deployment in Axis, verify that the deploy.wsdd file *matches the site machine configuration* (the jndiURL parameter in particular:).

51.1. Libraries

JOnAS incorporates all the necessary libraries, including:

- JAX-R: Reference Implementation from Sun
- JAX-M: Reference Implementation from Sun
- JAX-P: Xerces XML parser (version 2.4.0)
- AXIS: Soap implementation from Apache (with all dependent libs: jaxrpc.jar, etc.)

JAX-M and JAX-R are parts of the Web Services Development Pack from Sun.

WebLogic incorporates all the necessary libraries. The libraries for using the webservice are contained in webserviceclient.jar.

51.2. Accessing a JOnAS Web Service from a WebLogic Server's EJB

51.2.1. Web Service Development on JOnAS

Refer to Chapter 49 *How to use Axis in JOnAS*, which describes how to develop and deploy web services on JOnAS.

51.2.1.1. EJB Creation on JOnAS

To create a web service based on an EJB, first create a stateless EJB. Then, create a web application (.war) or an application (.ear) with this EJB that will define a URL with access to the Web Service.

51.2.1.2. WebService Deployment Descriptor (WSDD)

This section describes the deployment descriptor of the web service. To deploy a web service based on an EJB, specify the various elements in the WSDD. This WSDD enables the web service to be mapped on an EJB, by specifying the different EJB classes used.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<!-- AXIS deployment file for HelloBeanService -->
    <service name="WebServiceName" provider="java:EJB">

<!-- JNDI name specified in jonas-EJB.xml -->
    <parameter name="beanJndiName" value="EJB_JNDI_Name"/>
```

```
<!-- use of remote interfaces to access the EJB is allowed, but this
    example uses local interfaces -->
    <parameter name="homeInterfaceName" value="EJB_Home"/>

<parameter name="remoteInterfaceName" value="EJB_Interface"/>

<!-- JNDI properties: it may be necessary to modify hostname and port
    number, protocol name could be changed in accordance with the ORB
    used -->

<!-- for a RMI ORB -->
    <parameter name="jndiURL" value="rmi://<url>:<port>"/>

<parameter name="jndiContextClass"
    value="com.sun.jndi.rmi.registry.RegistryContextFactory"/>

<!-- Specify here allowed methods for Web Service access (* for all) -->
    <parameter name="allowedMethods" value="*"/>

    </service>
</deployment>
```

The various tags allow mapping of the web service on different java classes. If a web service uses a complex type, this complex type must be mapped with a java class. To do this, two tags can be used:

```
<beanMapping qName="ns:local" xmlns:ns="someNameSpace"
languageSpecificType="java:my.class"/>
```

This maps the `QName [someNameSpace]:[local]` with the class `my.class`.

```
<typeMapping qname="ns:local" wmlns:ns="someNamespace"
languageSpecificType="java:my.class" serializer="my.java.Serializer"
deserializer="my.java.DeserializerFactory"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

where `QName [someNameSpace]:[local]` is mapped with `my.class`. The serializer used is the class `my.java.Serializer` and the deserializer used is `my.java.DeserializerFactory`.

51.2.1.3. Web Service Deployment on JOnAS

First, deploy the web application or the application containing the EJB. Then, deploy the web service using the Axis client tool:

```
jclient org.apache.axis.client.AdminClient
    -hjonasServerHostname -p9000 deploy.wsdd
```

The web service WSDL is accessible from the url:

```
http://host:port/url-servlet/webservicename?wsdl.
```

51.2.2. EJB Proxy Development for WebLogic Server

This EJB provides access to the web service deployed on JOnAS from the WebLogic Server.

51.2.2.1. Generation of Web Service Client Class

To access the web service, generate the client class using the Ant task `clientgen`. For example:

```
<clientgen wsdl="<wsdl_url>" packageName="my.package"
   clientJar="client.jar" generatePublicFields="True"
   keep generated="True"/>
```

This command creates four classes:

- Service implementation
- Java interface
- Stub class
- Service interface to the corresponding web service.

The tool can also generate the Java classes corresponding to future complex types of web services.

51.2.2.2. Build the EJB

Then, call the web service in the EJB proxy code using the generated classes. For example:

```
try {
     WSNAME_Impl tsl=new WSNAME_Impl(); // access web service impl
     EJB_endpoint tsp = tsl.getEJB_endpoint();
        // access WS endpoint interface
     ComplexType tr=tsp.method(param);
   } catch (Exception e) {
     e.printStackTrace(System.err);
   };
```

51.2.2.3. Deploy the EJB on WebLogic Server

Deploy this EJB using the WebLogic administration console.

51.3. Accessing a WebLogic Web Service from a JOnAS EJB

51.3.1. Web Service Development for WebLogic Server

51.3.1.1. Creation of an Application

To create a web service, first develop the corresponding EJB application. Compile the EJB classes and create a JAR file. To create the EJB's container, apply the ant task

```
wlappc
```

to the JAR file. For example:

```
<wlappc debug="${debug}" source="interface_ws_jonas.jar"
    classpath="{java.class.path}:interface_ws_jonas.jar"
```

Then, use the ant task servicegen to create the ear application containing the web service.

```
<servicegen
  destEar="ears/myWebService.ear"
  contextURI="web_services" >
  <service
    ejbJar="jars/myEJB.jar"
    targetNamespace="http://www.bea.com/examples/Trader"
    serviceName="TraderService"
    serviceURI="/TraderService"
    generateTypes="True"
    expandMethods="True" >
  </service>
</servicegen>
```

Use the version of `ant` provided with WebLogic Server.

51.3.1.2. WebService Deployment

Deploy the webservice using the WebLogic administration console, and deploy the corresponding application.

The WSDL is accessible at `http://host:port/webservice/web_services?WSDL`

51.3.2. EJB Proxy Development for JOnAS

This EJB provides access to the web service deployed on WebLogic from JOnAS.

51.3.2.1. Generation of Web Service Client Class

To access a web service, generate a client class using the Axis tool WSDL2Java <webservice-url-wsdl>. This command creates four classes:

- WSNAME_Locator.java: Service implementation
- WSNAME_Port.java: Java Interface
- WSNAME_PortStub.java: Stub class
- WSNAME.java: Service interface to the corresponding web service.

The tool can also generate the Java classes corresponding to future complex types of web services.

51.3.2.2. Build the EJB

Then, use this generated class to call the web service in the EJB proxy code. For example:

```
try {
    WSNAMELocator tsl=new WSNAMELocator();
    WSNAMEPort tsp = tsl.getWSNAMEPort();
    ComplexType tr=tsp.method(param);
    ...
    } catch (Exception e) {
        e.printStackTrace(System.err);
    };
```

51.3.2.3. Deploy the EJB on JOnAS

Deploy the EJB using the JOnAS administration console or command.

RMI-IIOP Interoperability between JOnAS and BEA WebLogic

This chapter describes the basic interoperability between JOnAS and a BEA WebLogic Server using RMI-IIOP.

52.1. Accessing a JOnAS EJB from a WebLogic Server's EJB using RMI-IIOP

52.1.1. JOnAS Configuration

No modification to the EJB code is necessary. However, to deploy it for use with the iiop protocol, add the tag protocols and indicate iiop when creating the `build.xml`. For example:

```
<jonas destdir="${dist.ejbjars.dir}" classpath="${classpath}"
  jonasroot="${jonas.root}" protocols="iiop"/>
```

If GenIC is being used for deployment, the `-protocols` option can be used. Note also that an EJB can be deployed for several protocols. For more details about configuring the communication protocol, refer to the Section 3.3 *Configuring the Communication Protocol and JNDI*.

For the JOnAS server to use RMI-IIOP, the JOnAS configuration requires modification. The iiop protocol must be selected in the file carol.properties. This modification will allow an EJB to be created using the RMI-IIOP protocol.

52.1.2. EJB Proxy on WebLogic

To call an EJB deployed on JOnAS that is accessible through RMI-IIOP, load the class com.sun.jndi.cosnaming.CNCtxFactory as the initial context factory. In addition, specify the JNDI URL of the server name containing the EJB to call: "iiop://<server>:port." For example:

```
try {
  Properties h = new Properties();
  h.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.cosnaming.CNCtxFactory");
  h.put(Context.PROVIDER_URL, "iiop://<server>:<port>");
  ctx=new InitialContext(h);
  }
catch (Exception e) {
  ...
  }
```

Then, the JOnAS EJB is accessed in the standard way.

52.2. Access a WebLogic Server's EJB from a JOnAS EJB using RMI-IIOP

52.2.1. WebLogic Configuration

No modification to the EJB code is necessary. However, to deploy the EJB for use with the iiop protocol, add the element iiop="true" on the wlappc task when creating the `build.xml`. For example:

```
wlappc debug="${debug}" source="ejb.jar" iiop="true"
  classpath="${class.path}"
```

52.2.2. EJB Proxy on JOnAS

To call an EJB deployed on WebLogic Server that is accessible through RMI-IIOP, specify the JNDI URL of the server name containing the EJB to call. This URL is of the iiop://*server*:*port* type. For example:

```
try {
   Properties h = new Properties();
   h.put(Context.PROVIDER_URL, "iiop://server:port");
   ctx=new InitialContext(h);
   }
catch (Exception e) {
    ...
}
```

Then, the EJB deployed on WebLogic Server is accessed in the standard way.

Interoperability between JOnAS and CORBA

This chapter describes the basic interoperability between JOnAS and CORBA using RMI-IIOP.

53.1. Accessing an EJB Deployed on a JOnAS Server by a CORBA Client

53.1.1. JOnAS Configuration

No modification to the EJB code is necessary. However, the EJB should be deployed for the iiop protocol (for example, when the `build.xml` is created, add the tag "protocols" and specify "iiop"). For example:

```
<jonas destdir="${dist.ejbjars.dir}" classpath="${classpath}"
  jonasroot="${jonas.root}" protocols="iiop"/>
```

If GenIC is used for deployment, the `-protocols` option can be used. Note also that an EJB can be deployed for several protocols. Refer to the Chapter 3 *JOnAS Configuration* for more details about configuring the communication protocol.

The JOnAS configuration must be modified for the JOnAS server to use RMI-IIOP. Choose the iiop protocol in the file carol.properties. Refer also to the Section 3.3 *Configuring the Communication Protocol and JNDI* for details about configuring the communication protocol. These modifications will make it possible to create an EJB using the RMI-IIOP protocol.

53.1.2. Using RMIC to Create IDL Files Used by the CORBA Client

To call an EJB deployed on JOnAS that is accessible through RMI-IIOP, use the rmic tool on the EJB Remote interface and EJB Home interface to create the idl files. For example:

```
rmic -classpath $JONAS_ROOT/lib/common/j2ee/ejb.jar -idl package1.Hello
```

This action generates several `idl` files:

```
package1/Hello.idl
package1/HelloHome.idl

java/io/FilterOutputStream.idl
java/io/IOException.idl
java/io/IOEx.idl
java/io/OutputStream.idl
java/io/PrintStream.idl
java/io/Writer.idl
java/io/PrintWriter.idl

java/lang/Exception.idl
java/lang/Ex.idl
java/lang/Object.idl
java/lang/StackTraceElement.idl
java/lang/ThrowableEx.idl
java/lang/Throwable.idl

javax/ejb/EJBHome.idl
```

```
javax/ejb/EJBMetaData.idl
javax/ejb/EJBObject.idl
javax/ejb/Handle.idl
javax/ejb/HomeHandle.idl
javax/ejb/RemoveException.idl
javax/ejb/RemoveEx.idl

org/omg/boxedRMI/seq1_octet.idl
org/omg/boxedRMI/seq1_wchar.idl

org/javax/rmi/CORBA/ClassDesc.idl
org/omg/boxedRMI/java/lang/seq1_StackTraceElement.idl
```

Copy these files to the directory in which CORBA client development is being done.

53.1.3. CORBA Client Development

53.1.3.1. idlj

Once idl files are generated, apply the idlj tool to build Java files corresponding to the idl files (idlj = idl to java). To do this, apply the idlj tool to the Remote interface idl file and the Home interface idl file. For example:

```
idlj -fclient -emitAll package1/Hello.idl
```

The idlj tool also generates bugged classes. Be sure to put the _read and _write method in comments in the classes _Exception.java, CreateException.java, RemoveException.java.

Additionally, the classes OutputStream.java, PrintStream.java, PrintWriter.java, Writer.java, and FilterOuputStream.java must extend Serializable and then replace

```
((org.omg.CORBA_2_3.portable.OutputStream)
  ostream).write_value(value,id());
```

with

```
((org.omg.CORBA_2_3.portable.OutputStream)
  ostream).write_value((Serializable) value,id());
```

in the write method.

53.1.3.2. The CORBA Client

Create the CORBA client:

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class Client {
  public static void main(String args[]) {
    try {
        //Create and initialize the ORB
        ORB orb=ORB.init(args,null);
        //Get the root naming context
        org.omg.CORBA.Object
          objRef=orb.resolve_initial_references("NameService");
        NamingContext ncRef= NamingContextHelper.narrow(objRef);
```

```
        //Resolve the object reference in naming
        //make sure there are no spaces between ""
        NameComponent nc= new NameComponent("HelloHome","");
        NameComponent path[] = {nc};
        HelloHome tradeRef=HelloHomeHelper.narrow(ncRef.resolve(path));

        //Call the Trader EJB and print results
        Hello hello=tradeRef.create();
        String tr=hello.say();
        System.out.println("Result = "+tr);
    }
    catch (Exception e) {
        System.out.println("ERROR / "+e);
        e.printStackTrace(System.out);
    }
  }
}
```

53.1.3.3. Compilation

Compile the generated files.

⚠️**Warning**

Compile the file corresponding to the client parts, the files `Hello.java`, `HelloHome.java`, `_Exception.java`, ..., **and** `_*Stub.java`, `*Helper.java`, `*ValueFactory.java`, `*Operation.java` (* represents the name of the interface).

53.2. Accessing a CORBA Service by an EJB Deployed on JOnAS Server

53.2.1. Setting up the CORBA Service

Create the CORBA service:

1. Create the `idl` file corresponding to this service (for example, the interface name, which is "Hello").

2. Generate the Java file corresponding to the IDL service:
   ```
   idlj -fall Hello.idl
   ```

   (You can find the files used in this example on the Sun tutorial web site at http://java.sun.com/j2se/1.4.2/docs/guide/idl/tutorial/GSIDL.html.

3. Implement the Java interface (in this example, the service will be bound with the name "Hello" in the server implementation).

4. Start the orb.

5. Start the CORBA service.

53.2.2. Setting up the EJB on JOnAS

Set up the EJB on JOnAS as follows:

1. To call the CORBA service, generate the Java file corresponding to the IDL file. To do this, apply the idlj tool on the IDL file corresponding to the CORBA service description:
   ```
   idlj -fclient Hello.idl
   ```

2. Create an EJB.

3. To call the CORBA service, use the InitialContext object (by running JOnAS in IIOP mode).

4. Get the Java object corresponding to the CORBA service with the environment.

5. Call the method on this object.

Example code:

```
// Get the reference on the object
// Here the remote ORBd is running on localhost with the port 1050
// and the remote object is bound with the name "Hello".
org.omg.CORBA.Object o = (org.omg.CORBA.Object)
  initialContext.lookup("corbaname:iiop:1.2@localhost:1050#Hello");

// Do the narrow
Hello hello = (Hello) HelloHelper.narrow(o);

// Call the method
System.out.println(hello.sayHello());
return hello.sayHello();
}
catch (Exception e) {
    ...
}
```

How to Migrate the New World Cruises Application to JOnAS

This section describes the modifications required to migrate the J2EE application "New World Cruise" to a JOnAS server.

"New World Cruise" is a sample application that comes with Sun ONE Application Server. (See http://developers.sun.com/sw/building/tech_articles/jaxrpc/synopsis.html.)

54.1. JOnAS Configuration

The first step is to configure the database used for this application. Copy the file *db*.properties to the directory $JONAS_BASE/conf. Edit this file to complete the database connection.

Then, modify the JOnAS DBM database service configurations in the file $JONAS_BASE/conf/jonas.properties, to specify the file containing the database connection.

54.1.1. New World Cruise Application

54.1.1.1. EJB Modification Code

To be EJB2.0-compliant, add the exceptions RemoveException and CreateException for EJB's methods ejbRemove and ejbCreate.

Additionally, the GlueBean class uses a local object in GlueBean constructor. However, it must use a remote object because it is a class calling an EJB. Therefore, modify the comment in this class with the following:

```
// If using the remote interface, the call would look like this
cruiseManagerHome = (CruiseManagerHome)
  javax.rmi.PortableRemoteObject.narrow(result, CruiseManagerHome.class);
// Using the local interface, the call looks like this
//cruiseManagerHome = (CruiseManagerHome) result;
```

54.1.2. EJB's Deployment Descriptor

There are three EJBs, thus there must be three ejb-jar.xml files that correspond to the EJB's deployment descriptors and three jonas-ejb-jar.xml files that correspond to the JOnAS deployment descriptors.

First, rename the files *ejb_name*.ejbdd with *ejb_name*.xml (these files contain the EJB deployment descriptors).

Create the three jonas-*ejb_name*.xml files corresponding to the EJBs.

For the two entity Beans (Cruise and CruiseManager), describe the mapping between:

- The EJB name and JNDI name (JNDI name=ejb/*ejb_name*)
- The JDBC and the table name
- The EJB field and the table field, (the version of CMP is not specified in ejb-jar and JOnAS by default uses CMP1.1).

For the session Bean, describe the mapping between:

• The EJB name and JNDI name (JNDI name=ejb/*ejb name*)

54.1.3. Web Application

Create the `jonas-web.xml` that corresponds to the deployment descriptor of the New World Cruise application. Package the `jonas-web.xml` and the files under the directory `Cruises/cruise_WebModule` in the WAR file.

54.1.4. Build Application

Build the EAR corresponding to the application.

This EAR contains the three files corresponding to the three EJBs, as well as the web application.

54.2. SUN Web Service

54.2.1. Axis Classes Generation

To call a web service, first generate Axis classes. The generated classes will allow a web service to be called using the static method.

For this step, download the file AirService.wsdl that corresponds to the SUN web service description or use the URL containing this file.

Then use the command:

```
java org.apache.axis.wsdl.WSDL2java <file_name>
```

This command generates four Java files:

AirService.java: the service interface.

AirServiceLocator.java: the service implementation

AirServiceServantInterface: the endpoint interface

AirServiceServantInterfaceBindingStub.java: the stub class

To call the SUN web service, instantiate the service implementation. Then call the method getAirService() to get the end point, and call the appropriate method.

```
AirService airService=new AirServiceLocator();
AirServiceServantInterface interface=airService getAirService();
Object result=interface.<method>;
```

54.2.2. JSP Files

The file Part2_site.zip contains the web application that uses the SUN web service.

It includes several JSP files that must be modified to use the Axis classes.

As an example, make the following replacements in the index.jsp file:

```
// Get our port interface
AirPack.AirClientGenClient.AirService service =
        new AirPack.AirClientGenClient.AirService_Impl();
AirPack.AirClientGenClient.AirServiceServantInterface port =
        service getAirServiceServantInterfacePort();

// Get the stub and set it to save the HTTP log.
AirPack.AirClientGenClient.AirServiceServantInterface_Stub stub =
        (AirPack.AirClientGenClient.AirServiceServantInterface_Stub) port;
java.io.ByteArrayOutputStream httpLog =
        new java.io.ByteArrayOutputStream();
stub._setTransportFactory
    (new com.sun.xml.rpc.client.http.HttpClientTransportFactory(httpLog));

// Get the end point address and save it for the error page.
String endPointAddress = (String)
        stub._getProperty(stub.ENDPOINT_ADDRESS_PROPERTY);
request.setAttribute("ENDPOINT_ADDRESS_PROPERTY", endPointAddress);
```

by

```
// Get our port interface
AirService_pkg.AirService service = new AirService_pkg.AirServiceLocator();
AirService_pkg.AirServiceServantInterface port =
        service getAirServiceServantInterfacePort();
```

Additionally, the exception:

```
throw new com.sun.xml.rpc.client.ClientTransportException(null,
  new Object[] {e});
```

is replaced by:

```
throw new Exception(e);
```

54.2.3. Web Application

Finally, create the web application (jonas-web.xml) and reuse the web.xml that is in Part2_site.zip. Then, build the web application, which contains:

```
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/jonas-web.xml
WEB-INF/lib/
WEB-INF/lib/CruiseManager.jar
WEB-INF/classes/
WEB-INF/classes/AirService_pkg/
WEB-INF/classes/AirService_pkg/AirServiceServantInterface.class
WEB-INF/classes/AirService_pkg/AirServiceServantInterfaceBindingStub.class
WEB-INF/classes/AirService_pkg/AirService.class
WEB-INF/classes/AirService_pkg/AirServiceLocator.class
PalmTree.jpg
aboutus.jsp
air_icon.gif
airbook.jsp
airclient.jsp
airdates.jsp
airdone.jsp
```

```
airlist.jsp
clear.gif
crubook.jsp
crudone.jsp
cruise_icon.gif
cruises.jsp
flights.jsp
index.jsp
nwcl_banner.gif
nwcl_banner_a.gif
nwcl_styles.css
WEB-INF/web.xml
```

54.3. JOnAS Web Service

54.3.1. Deployment

This web service uses the EJB stateless CruiseManager. To deploy this web service, create the web service deployment descriptor:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <!-- AXIS deployment file for EJB Cruise -->
  <service name="AirService" provider="java:EJB">

  <!-- JNDI name specified in jonas-CruiseApp. -->
  <parameter name="beanJndiName"
            value="ejb/CruiseManager"/>

  <!-- you can use remote interfaces to access the EJB -->
    <parameter name="homeInterfaceName"
              value="cruisePack.CruiseManagerHome"/>
    <parameter name="remoteInterfaceName"
              value="cruisePack.CruiseManager"/>


  <!-- Specify allowed methods for Web Service access
      (* for all) -->
  <parameter name="allowedMethods"
            value="createPassenger,getAllDates,getByDepartdate"/>

  <typeMapping
    xmlns:ns="urn:AirService/types"
    qname="ns:ArrayOfString"
    type="java:java.lang.String[]"
    serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
    deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  />
  </service>
</deployment>
```

To deploy this web service, first deploy the web application `axis.war` and the EJB corresponding to the web service (`CruiseManager.jar`).

Then, deploy the web service using the Axis client:

```
jclient org.apache.axis.client.AdminClient
  -lhttp://localhost:port/context-root-axis.war/servlet/AxisServlet
    ws_wsdd
```

54.3.2. Axis Classes Generation

To call a web service, first generate Axis classes. The generated classes will allow a web service to be called using the static method.

For this step, download the file `AirService.wsdl` corresponding to the SUN web service description or use the URL containing this file.

The use of the command is as follows:

```
java org.apache.axis.wsdl.WSDL2java <file_name or url>
```

This command generates four Java files:

CruiseManagerService.java: the service interface

CruiseManagerServiceLocator.java: the service implementation

CruiseManager.java: the endpoint interface

AirServiceSoapBindingStub.java: the stub class.

To call the JOnAS web service, instantiate the service implementation. Then, call the method getAirService() to get the end point interface, and call the appropriate method.

```
AirService_JOnAS.Client.CruiseManagerService cms=
     new AirService_JOnAS.Client.CruiseManagerServiceLocator();

AirService_JOnAS.Client.CruiseManager cmi=cms.getAirService();

Object result=cmi.<method>;
```

54.3.3. JSP Files

To access the JOnAS web service, copy the JSP files contained in the EJB's web application (Cruises/cruise_WebModule).

The JOnAS web service call must replace the call for each EJB.

54.3.4. Web Application

Finally, create the web application `jonas-web.xml`. Then, build the web application, which contains:

```
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/jonas-web.xml
WEB-INF/lib/
WEB-INF/lib/CruiseManager.jar
WEB-INF/classes/
WEB-INF/classes/AirService_pkg/
WEB-INF/classes/AirService_JOnAS/Client/CruiseManagerService.class
WEB-INF/classes/AirService_JOnAS/Client/AirServiceSoapBindingStub.class
WEB-INF/classes/AirService_JOnAS/Client/CruiseManager.class
```

```
WEB-INF/classes/AirService_JOnAS/Client/ \
  CruiseManagerServiceLocator/AirServiceLocator.class
PalmTree.jpg
aboutus.jsp
air_icon.gif
airbook.jsp
airclient.jsp
airdates.jsp
airdone.jsp
airlist.jsp
clear.gif
crubook.jsp
crudone.jsp
cruise_icon.gif
cruises.jsp
flights.jsp
index.jsp
nwcl_banner.gif
nwcl_banner_a.gif
nwcl_styles.css
WEB-INF/web.xml
```

Chapter 55.
Configuring Resource Adapters

55.1. Configuring JDBC Resource Adapters

Instead of using the JOnAS database service for configuring DataSources, it is also possible to use the JOnAS resource service and JDBC connectors compliant with the J2EE Connector Architecture specification. The resulting functionality is the same, and the benefit is the management of pools of JDBC PrepareStatements. This chapter describes how the JDBC Resource Adapters should be configured to connect the application to databases.

55.1.1. Configuring Resource Adapters

For both container-managed or bean-managed persistence, the JDBC Resource Adapter (JDBC RA) makes use of relational storage systems through the JDBC interface. JDBC connections are obtained from a JDBC RA. The JDBC RA implements the J2EE Connector Specification using the *DataSource* interface as defined in the JDBC 2.0 standard extensions. An RA is configured to identify a database and a means to access it via a JDBC driver. Multiple JDBC RAs can be deployed either via the `jonas.properties` file or included in the autoload directory of the resource service. For complete information about RAs in JOnAS, refer to Chapter 56 *Configuring Resource Adapters*. The following section explains how JDBC RARs can be defined and configured in the JOnAS server.

To support distributed transactions, the JDBC RA requires the use of at least a JDBC2-XA-compliant driver. Such drivers implementing the *XADataSource* interface are not always available for all relational databases. The JDBC RA provides a generic *driver-wrapper* that emulates the XADataSource interface on a regular JDBC driver. It is important to note that this driver-wrapper does not ensure a real two-phase commit for distributed database transactions.

The generic JDBC RAs of JOnAS provide implementations of the DriverManager, DataSource, PooledConnection, and XAConnection interfaces (see JDBC 3.0 (http://java.sun.com/products/jdbc/)). They are located in the `$JONAS_ROOT/rars/autoload` directory and thus are deployed automatically. They consist of base (or generic) RAs facilitating the build of the user JDBC RAs. Depending on the relational database management server and the available interface in the used JDBC-compliant driver, the user JDBC RA is linked (through the Section 3.9.2 *RAR Link* feature) to a generic RA (for example, the DriverManager's one). In this case, the user RA contains only a `jonas-ra.xml` file with some specific parameters, such as the connection url, the user/password, or the JDBC-Driver class.

Resource Adapter Provided with JOnAS	Description	JNDI Name
`rars/autoload/JOnAS_j` Generic resource adapter that implements the DataSource interface		JOnASJDBC_DS
`rars/autoload/JOnAS_j` Generic resource adapter that implements the DriverManager interface		JOnASJDBC_DM
`rars/autoload/JOnAS_j` Generic resource adapter that implements the PooledConnection interface		JOnASJDBC_CP

Resource Adapter Provided with JOnAS	Description	JNDI Name
rars/autoload/JOnAS_jdbcxa	Generic resource adapter that implements the XAConnection interface	JOnASJDBC_XA

The remainder of this section, which describes how to define and configure JDBC RAs, is *specific to JOnAS*. However, the way to use these JDBC RAs in the Application Component methods is standard; that is, via the resource manager connection factory references (refer to the example in Section 8.6 *Writing Database Access Operations (Bean-Managed Persistence)* in Chapter 8 *Developing Entity Beans*).

An RAR file must be configured and deployed (for example, `Oracle1.rar` for an Oracle RAR and `MySQL1.rar` for a MySQL RAR, as delivered with the platform).

To define a Resource "Oracle1" in the `jonas.properties` file, add its name "Oracle1" (name of the RAR file) to the line `jonas.service.resource.services` or just include it in the autoload directory. For more information about deploying an RAR, refer to Chapter 56 *Configuring Resource Adapters*.

```
jonas.service.resource.services Oracle1,MySQL1,PostgreSQL1
```

The `jonas-ra.xml` file that defines a DataSource should contain the following information:

jndiname	JNDI name of the RAR
URL	The JDBC database URL: jdbc:*database_vendor_subprotocol*:...
dsClass	Name of the class implementing the JDBC driver
user	Database user name
password	Database user password

An RAR for Oracle configured as `jdbc_1` in JNDI and using the Oracle *thin* DriverManger JDBC driver, should be described in a file called `Oracle1_DM.rar`, with the following properties configured in the `jonas-ra.xml` file:

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.objectweb.org/jonas/ns
  http://www.objectweb.org/jonas/ns/jonas-connector_4_1.xsd" >
   <jndi-name>jdbc_1</jndi-name>
 <rarlink>JOnASJDBC_DM</rarlink>
 .
 .
 <jonas-config-property>
   <jonas-config-property-name>user
     </jonas-config-property-name>
   <jonas-config-property-value>scott
     </jonas-config-property-value>
 </jonas-config-property>
 <jonas-config-property>
   <jonas-config-property-name>password
     </jonas-config-property-name>
   <jonas-config-property-value>tiger
     </jonas-config-property-value>
 </jonas-config-property>
 <jonas-config-property>
   <jonas-config-property-name>loginTimeout
```

```
        </jonas-config-property-name>
    <jonas-config-property-value>
        </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>URL
        </jonas-config-property-name>
    <jonas-config-property-value>jdbc:oracle:thin:@malte:1521:ORA1
        </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>dsClass
        </jonas-config-property-name>
    <jonas-config-property-value>oracle.jdbc.driver.OracleDriver
        </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>mapperName
        </jonas-config-property-name>
    <jonas-config-property-value>rdb.oracle
        </jonas-config-property-value>
  </jonas-config-property>
</jonas-connector>
```

In this example, "malte" is the hostname of the server running the Oracle DBMS, 1521 is the SQL*Net V2 port number on this server, and ORA1 is the ORACLE_SID.

This example makes use of the Oracle "Thin" JDBC driver. For an application server running on the same host as the Oracle DBMS, you can use the Oracle OCI JDBC driver; in this case, the URL to use is jdbc:oracle:oci7: or jdbc:oracle:oci8:, depending on the Oracle release. Oracle JDBC drivers can be downloaded from the Oracle web site http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html.

To create a MySQL RAR configured as `jdbc_2` in JNDI, it should be described in a file called `MySQL2_DM.rar`, with the following properties configured in the `jonas-ra.xml` file:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.objectweb.org/jonas/ns
  http://www.objectweb.org/jonas/ns/jonas-connector_4_1.xsd" >
    <jndi-name>jdbc_2</jndi-name>
  <rarlink>JOnASJDBC_DM</rarlink>
  .
  .
  <jonas-config-property>
    <jonas-config-property-name>user
        </jonas-config-property-name>
    <jonas-config-property-value>
        </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>password
        </jonas-config-property-name>
    <jonas-config-property-value>
        </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>loginTimeout
        </jonas-config-property-name>
    <jonas-config-property-value>
        </jonas-config-property-value>
  </jonas-config-property>
```

```
  <jonas-config-property>
    <jonas-config-property-name>URL
      </jonas-config-property-name>
    <jonas-config-property-value>jdbc:mysql://malte/db_jonas
      </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>dsClass
      </jonas-config-property-name>
    <jonas-config-property-value>org.gjt.mm.mysql.Driver
      </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>mapperName
      </jonas-config-property-name>
    <jonas-config-property-value>rdb.mysql
      </jonas-config-property-value>
  </jonas-config-property>
</jonas-connector>
```

To create a PostgreSQL RAR configured as `jdbc_3` in JNDI, it should be described in a file called `PostGreSQL3_DM.rar`, with the following properties configured in the `jonas-ra.xml` file:

```
<?xml version = "1.0" encoding = "UTF-8"?>
  <jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.objectweb.org/jonas/ns
   http://www.objectweb.org/jonas/ns/jonas-connector_4_1.xsd" >
    <jndi-name>jdbc_3</jndi-name>
  <rarlink>JOnASJDBC_DM</rarlink>
  .
  .
  <jonas-config-property>
    <jonas-config-property-name>user</jonas-config-property-name>
    <jonas-config-property-value>jonas</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>password</jonas-config-property-name>
    <jonas-config-property-value>jonas</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>loginTimeout
      </jonas-config-property-name>
    <jonas-config-property-value></jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>URL</jonas-config-property-name>
    <jonas-config-property-value>jdbc:postgresql:/malte:5432/db_jonas
      </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>dsClass</jonas-config-property-name>
    <jonas-config-property-value>org.postgresql.Driver
      </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>mapperName
      </jonas-config-property-name>
    <jonas-config-property-value>rdb.mpostgres
      </jonas-config-property-value>
  </jonas-config-property>
</jonas-connector>
```

The database user and password can be handled in one of two ways:

- Put it in the `jonas-ra.xml` file in the RAR file and have the Application Components use the `getConnection()` method.
- Not have it in the RAR file and have the Application Component use the `getConnection(String username, String password)` method.

55.1.2. Using CMP2.0/JORM

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework (see http://www.objectweb.org/jorm/index.html). JORM must adapt its object-relational mapping to the underlying database, and makes use of adapters called "mappers" for this purpose. Thus, for each type of database (and more precisely for each JDBC driver), the corresponding mapper must be specified in the `jonas-ra.xml` file of the deployed RAR. The `mapperName` element is provided for this purpose.

The JORM database mapper that has the property name `mapperName` can have the following values:

- rdb: generic mapper (JDBC standard driver ...)
- rdb.postgres: mapper for PostgreSQL
- rdb.oracle8: mapper for Oracle 8 and lesser versions
- rdb.oracle: mapper for Oracle 9
- rdb.mckoi: mapper for McKoi Db
- rdb.mysql: mapper for MySQL

Refer to the JORM documentation for a complete updated list.

55.1.3. ConnectionManager Configuration

Each RAR uses a connection manager that can be configured through the `jonas-ra.xml` file. The pool parameters are described in Section 3.9.1 *ConnectionManager Pool Configuration*. The following table described the JDBC parameters. These settings all have default values and they are not required.

Property name	Description	Default Value
jdbc-check-level	JDBC connection checking level: 0 : no check 1 : check connection still open 2 : call the test statement before reusing a connection from the pool	0 (no check)
jdbc-test-statement	jdbc-test-statement is not used when jdbc-check-level is equal to 0 or 1	select 1

Table 55-1. jdbc-conn-params elements

55.1.4. Tracing SQL Requests through P6Spy

The P6Spy tool is integrated into JOnAS and it provides an easy way to trace the SQL requests sent to the database.

To enable this tracing feature, perform the following configuration steps:

1. Update the appropriate RAR file's `jonas-ra.xml` file by setting the `dsClass` property to `com.p6spy.engine.spy.P6SpyDriver`.

2. Set the `realdriver` property in the `spy.properties` file (located in `$JONAS_BASE/conf`) to the JDBC driver of your actual database.

3. Verify that `logger.org.objectweb.jonas.jdbc.sql.level` is set to `DEBUG` in `$JONAS_BASE/conf/trace.properties`.

Example:

jonas-ra.xml file content:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jonas-connector xmlns="http://www.objectweb.org/jonas/ns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.objectweb.org/jonas/ns
 http://www.objectweb.org/jonas/ns/jonas-connector_4_1.xsd" >
  <jndi-name>jdbc_3</jndi-name>
  <rarlink>JOnASJDBC_DM</rarlink>
  <native-lib></native-lib>
  <log-enabled>true</log-enabled>
  <log-topic>org.objectweb.jonas.jdbc.DMPostgres</log-topic>
  <pool-params>
    <pool-init>0</pool-init>
    <pool-min>0</pool-min>
    <pool-max>100</pool-max>
    <pool-max-age>0</pool-max-age>
    <pstmt-max>10</pstmt-max>
  </pool-params>
  <jdbc-conn-params>
    <jdbc-check-level>0</jdbc-check-level>
    <jdbc-test-statement></jdbc-test-statement>
  </jdbc-conn-params>
  <jonas-config-property>
    <jonas-config-property-name>user</jonas-config-property-name>
    <jonas-config-property-value>jonas</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>password</jonas-config-property-name>
    <jonas-config-property-value>jonas</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>loginTimeout</jonas-config-property-name>
    <jonas-config-property-value></jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>URL</jonas-config-property-name>
    <jonas-config-property-value>
       jdbc:postgresql://your_host:port/your_db
          </jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>dsClass</jonas-config-property-name>
    <jonas-config-property-value>
       com.p6spy.engine.spy.P6SpyDriver</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>mapperName</jonas-config-property-name>
    <jonas-config-property-value>rdb.postgres
       </jonas-config-property-value>
```

```
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>logTopic</jonas-config-property-name>
    <jonas-config-property-value>
       org.objectweb.jonas.jdbc.DMPostgres</jonas-config-property-value>
  </jonas-config-property>
</jonas-connector>
```

In the `$JONAS_BASE/conf/spy.properties` file:

```
realdriver=org.postgresql.Driver
```

In `$JONAS_BASE/conf/trace.properties`:

```
logger.org.objectweb.jonas.jdbc.sql.level   DEBUG
```

55.1.5. Migration from dbm Service to the JDBC RA

The migration of a `Database.properties` file to a similar Resource Adapter can be accomplished through the execution of the following RAConfig tool command. Refer to Section 6.7 *RAConfig* for a complete description.

The pool parameters are described in Section 3.9.1 *ConnectionManager Pool Configuration*.

```
RAConfig -dm -p MySQL $JONAS_ROOT/rars/autoload/JOnAS_jdbcDM MySQL
```

This command will create a `MySQL.rar` file based on the `MySQL.properties` file, as specified by the `-p` parameter. It will also include the `<rarlink>` to the `JOnAS_jdbcDM.rar`, as specified by the `-dm` parameter.

The `jonas-ra.xml` created by the previous command can be updated further, if desired. Once the additional properties have been configured, update the `MySQL.rar` file using the following command:

```
RAConfig -u jonas-ra.xml MySQL
```

Configuring Resource Adapters

This chapter describes how to use resource adapters with JOnAS.

56.1. Principles

Resource Adapters are packaged for deployment in a standard Java programming language Archive file called a RAR file (Resource ARchive), which is described in the J2EE Connector Architecture specification.

The standard method of creating the `jonas-ra.xml` file is by using the RAConfig tool, for a complete description see Section 6.7 *RAConfig*.

56.2. Description and Examples

The `jonas-ra.xml` contains JOnAS-specific information describing deployment information, logging, pooling, JDBC connections, and RAR configuration property values.

- *Deployment Tags*:
  - jndiname: (Required) Name the RAR will be registered as. This value will be used in the resource-ref section of an EJB.
  - rarlink: The JNDI Name of a base RAR file. Useful for deploying multiple connection factories without having to deploy the complete RAR file again. When this is used, the only entry in the RAR is `META-INF/jonas-ra.xml`.
  - native-lib: Directory where additional files in the RAR should be deployed.

- *Logging Tags*:
  - log-enabled: Determines if logging should be enabled for the RAR.
  - log-topic: Log topic to use for the PrintWriter logger, which allows a separate handler for each deployed RAR.

- *Pooling Tags*:
  - pool-init: Initial size of the managed connection pool.
  - pool-min: Minimum size of the managed connection pool.
  - pool-max: Maximum size of the managed connection pool. Value of -1 is unlimited.
  - pool-max-age: Maximum number of milliseconds to keep the managed connection in the pool. Value of 0 is an unlimited amount of time.
  - pstmt-max: Maximum number of PreparedStatements per managed connection in the pool. Only required with JDBC resource adapters. Value of 0 is unlimited; -1 disables the cache.

- *JDBC Connection Tags*: Only valid with a Connection implementation of java.sql.Connection.

- jdbc-check-level: Level of checking that will be done for the JDBC connection. Values are 0 for no checking, 1 to validate that the connection is not closed before returning it, and greater than 1 to send the jdbc-test-statement.

- jdbc-test-statement: Test SQL statement sent on the connection if the jdbc-check-level is set accordingly.

- *Config Property Value Tags*:
  - Each entry must correspond to the config-property specified in the `ra.xml` of the RAR file.

56.2.1. Examples

The following portion of a `jonas-ra.xml` file illustrates linking to a base RAR file named `BaseRar`. All properties from the base RAR will be inherited and any values given in this `jonas-ra.xml` will override the other values.

```
<jonas-resource>
  <jndiname>rar1</jndiname>
  <rarlink>BaseRar</rarlink>
  <native-lib>nativelib</native-lib>
  <log-enabled>false</log-enabled>
  <log-topic>com.xxx.rar1</log-topic>
  <jonas-config-property>
    <jonas-config-property-name>ip</jonas-config-property-name>
    <jonas-config-property-value>www.xxx.com</jonas-config-property-value>
  </jonas-config-property>
  .
  .
</jonas-resource>
```

The following portion of a `jonas-ra.xml` file shows the configuration of a JDBC RAR file.

```
<jonas-resource>
  <jndiname>jdbc1</jndiname>
  <rarlink></rarlink>
  <native-lib>nativelib</native-lib>
  <log-enabled>false</log-enabled>
  <log-topic>com.xxx.jdbc1</log-topic>
  <pool-params>
    <pool-init>0</pool-init>
    <pool-min>0</pool-min>
    <pool-max>100</pool-max>
    <pool-max-age>0</pool-max-age>
    <pstmt-max>20</pstmt-max>
  </pool-params>
  <jdbc-conn-params>
    <jdbc_check-level>2</jdbc_check-level>
    <jdbc-test-statement>select 1</jdbc-test-statement>
  </jdbc-conn-params>
  <jonas-config-property>
    <jonas-config-property-name>url</jonas-config-property-name>
    <jonas-config-property-value>jdbc:oracle:thin:@test:1521:DB1
      </jonas-config-property-value>
  </jonas-config-property>
  .
  .
</jonas-resource>
```

Index

Symbols

A

B

C