JOnAS Platform Documentation

Table of Contents

JOnAS Documentation......1

Java Open Application Server (JOnAS): a J2EETM Platform......4
- Introduction......4
- JOnAS Features......7
- JOnAS Architecture......9
- JOnAS Development and Deployment Environment......17
- Clustering and Performance......18
- Perspectives......20

Getting started with JOnAS......22
- JOnAS Installation......22
- Running a First EJB Application......23
- More Complex Examples......25

Configuration Guide......30
- JOnAS Configuration Rules......31
- Configuring JOnAS Environment......32
- Configuring the Communication Protocol and JNDI......33
- Configuring the Logging System (monolog)......35
- Configuring JOnAS Services......37
- Configuring Registry Service......38
- Configuring EJB Container Service......39
- Configuring WEB Container Service......39
- Configuring WebServices Service......40
- Configuring EAR Service......41
- Configuring Transaction Service......42
- Configuring Database Service......42
- Configuring Security Service......45
- Configuring JMS Service (not compliant with 2.1 MDBs)......45
- Configuring Resource Service......46
- Configuring JMX Service......46
- Configuring Mail Service......47
- Configuring DB Service (hsql)......49
- Configuring Security......49
- Configuring JDBC DataSources......61
- Configuring JDBC Resource Adapters......66
- Configuring JMS Resource Adapters......72

J2EE Application Programmer's Guide......78
- Target Audience and Content......78
- Principles......78

Table of Contents

J2EE Application Programmer's Guide
JOnAS class loader hierarchy....80

EJB Programmer's Guide: Developing Session Beans....84
Target Audience and Content....84
Introduction....84
The Home Interface....85
The Component Interface....85
The Enterprise Bean Class....86
Tuning Stateless Session Bean Pool....88

EJB Programmer's Guide: Developing Entity Beans....90
Target Audience and Content....90
Introduction....90
The Home Interface....91
The Component Interface....93
The Primary Key Class....94
The Enterprise Bean Class....96
Writing Database Access Operations (bean−managed persistence)....102
Configuring Database Access for Container−managed Persistence....104
Tuning Container for Entity Bean Optimizations....108
Using CMP2.0 persistence....111

EJB Programmer's Guide: Message−driven Beans....133
Description of a Message−driven Bean....133
Developing a Message−driven Bean....133
Administration aspects....136
Running a Message−driven Bean....137
Transactional aspects....139
Example....139
Tuning Message−driven Bean Pool....142

EJB Programmer's Guide: Transactional Behaviour....143
Target Audience and Content....143
Declarative Transaction Management....143
Bean−managed Transaction....145
Distributed Transaction Management....145

EJB Programmer's Guide: Enterprise Bean Environment....148
Target Audience and Content....148
Introduction....148
Environment Entries....148

Table of Contents

EJB Programmer's Guide: Enterprise Bean Environment
- Resource References....149
- Resource Environment References....150
- EJB References....150
- Deprecated EJBContext.getEnvironment() method....152

EJB Programmer's Guide: Security Management....153
- Target Audience and Content....153
- Introduction....153
- Declarative Security Management....153
- Programmatic Security Management....154

EJB Programmer's Guide: Defining the Deployment Descriptor....157
- Target Audience and Content....157
- Principles....157
- Example of Session Descriptors....158
- Example of Container−managed Persistence Entity Descriptors (CMP 1.1)....160
- Tips....162

EJB Programmer's Guide: EJB Packaging....163
- Target Audience and Content....163
- Principles....163

Web Application Programmer's Guide....164
- Target Audience and Content....164
- Developing Web Components....164
- Defining the Web Deployment Descriptor....170
- WAR Packaging....174

J2EE Connector Programmer's Guide....175
- Target Audience and Content....175
- Principles....175
- Defining the JOnAS Connector Deployment Descriptor....175
- Resource Adapter (RAR) Packaging....177
- Use and Deployment of a Resource Adapter....177
- JDBC Resource Adapters....179
- Appendix: Connector Architecture Principles....180

J2EE Client Application Programmer's Guide....182
- Target Audience and Content....182
- Launching J2EE Client Applications....182
- Defining the Client Deployment Descriptor....184

Table of Contents

J2EE Client Application Programmer's Guide
- Client Packaging 186

J2EE Application Assembler's Guide 188
- Target Audience and Content 188
- Defining the Ear Deployment Descriptor 188
- EAR Packaging 190

Deployment and Installation Guide 192
- Target audience 192
- Deployment and installation process principles 192
- Example of deploying and installing an EJB using an ejb–jar file 193
- Deploying and installing a Web application 194
- Deploying and installing a J2EE application 194

Administration Guide 196
- jonas admin 196
- JonasAdmin 196

JOnAS Commands Reference Guide 202
- jonas 202
- jclient 205
- newbean 206
- registry 208
- GenIC 208
- JmsServer 210
- RAConfig 210

Creating a New JOnAS Service 213
- Target Audience and Rationale 213
- Introducing a new Service 213
- Advanced Understanding 215

JMS User's Guide 218
- JMS installation and configuration aspects 218
- Writing JMS operations within an application component 219
- Some programming rules and restrictions when using JMS within EJB 222
- JMS administration 225
- Running an EJB performing JMS operations 227
- A JMS EJB example 229

Table of Contents

Ant EJB Tasks User Manual......234
New JOnAS (Java Open Application Server) element for the current JOnAS version......234

Login Modules in a Java Client Guide......237
Configuring an environment to use login modules with java clients......237
Example of a client......237

Web Services with JOnAS......239
1. Web Services......239
2. Exposing a J2EE Component as a Web Service......241
3. The client of a Web Service......244
4. WsGen......247
6. Limitations......249

Working with Management Beans......250
Target Audience and Rationale......250
About JOnAS MBeans and their use in JonaSAdmin......250
Using JOnAS MBeans in a Management Application......250
Registering User MBeans......251

Howto: JOnAS Versions Migration Guide......252
JOnAS 4.1 to JOnAS 4.3.x......252
JOnAS 3.3.x to JOnAS 4.1......253
JOnAS 3.1 to JOnAS 3.1.4......254
JOnAS 3.0 to JOnAS 3.1......257
JOnAS 2.6.4 to JOnAS 3.0......258
JOnAS 2.6 to JOnAS 2.6.1......258
JOnAS 2.5 to JOnAS 2.6......259
JOnAS 2.4.4 to JOnAS 2.5......260
JOnAS 2.4.3 to JOnAS 2.4.4......260
JOnAS 2.3 to JOnAS 2.4......261

Howto: Installing JOnAS from scratch......263
JDK 1.4 installation......263
Ant 1.6 installation......263
Tomcat 5.0.x installation......264
Jetty 5.0.x installation......264
JOnAS installation......264
Setup......265

Table of Contents

Howto: Installing the packaging JOnAS with a web container (JOnAS/Tomcat or JOnAS/Jetty) from scratch....266
JDK 1.4 installation....266
ANT 1.6 installation....266
JOnAS/Web Container installation....267
Setup....267
Starting JOnAS and running some examples....267

Howto: How to compile JOnAS....269
Target Audience and Rationale....269
Getting the JOnAS Source....269
Recompiling JOnAS from the Source....269
Recompiling the package JOnAS/Jetty/Axis from the Source....270
Recompiling the package JOnAS/Tomcat/Axis from the Source....270

Howto: Clustering with JOnAS....272
Architecture....272
Products Installation....274
Load balancing at web level with mod_jk....274
Session Replication at web level....278
Load Balancing at EJB level....280
Preview of a coming version....282
Used symbols....283
References....283

Howto: Usage of AXIS in JOnAS....284
Libraries....284
1. Unique Axis Webapp....284
2. Embedded Axis Webapp....285
3. Tests....285
Tools:....285

Howto: Using WebSphere MQ JMS guide....287
Architectural rules....287
Setting the JOnAS Environment....288
Configuring WebSphere MQ....289
Starting the application....291
Limitations....291

Howto: Web Service Interoperability between JOnAS and Weblogic....292
Libraries....292
Access a web service deployed on JOnAS from an EJB deployed on Weblogic server....292

Table of Contents

Howto: Web Service Interoperability between JOnAS and Weblogic
Access a web service deployed on Weblogic server from an EJB deployed on JOnAS 295

Howto: RMI–IIOP interoperability between JOnAS and Weblogic 297
Accessing an EJB deployed on JOnAS from an EJB deployed on Weblogic server using RMI–IIOP 297
Access an EJB deployed on Weblogic Server by an EJB deployed on JOnAS using RMI–IIOP 298

Howto: Interoperability between JOnAS and CORBA 299
Accessing an EJB deployed a on JOnAS server by a CORBA client 299
Accessing a CORBA service by an EJB deployed on JOnAS server 301

Howto: Migrate the New World Cruises application to JOnAS 304
JOnAS configuration 304
New World Cruise Application 304
SUN web service 305
JOnAS web service 307

Howto: Execute JOnAS as a WIN32 Service 311
Instructions 311
Files managed by create_win32service 312
Modify JOnAS Configuration 313
Testing configuration 313

Howto: Getting Started with WebServices and JOnAS 3.X 314
WebServices and J2EE 314
Early Integration of Axis in JOnAS 3.X series 314
How to use WebServices 314

Howto: Distributed Message Beans in JOnAS 4.1 323
Scenario and general architecture 323
Common configuration 323
Specific configuration 324
And now, the beans! 325

Howto: install jUDDI server on JOnAS 327
I. UDDI Server 327
II. jUDDI Overview 327
III. How to Find the Latest Version 327
IV. Install Steps 328
V. Links 331

Table of Contents

Howto: JOnAS and JORAM: Distributed Message Beans......332
A How−To Document for JOnAS version 3.3......332
JOnAS and JORAM: Configuration Basics......332
JORAM Topics and JOnAS Administration......333
The Solution......333
The Full Configuration......334
The JoramDistributionService......337
Maintaining the configuration......340
Conclusion......341

Howto: JSR 160 support in JOnAS......342
Target Audience and Rationale......342
What is JSR 160 ?......342
Connector servers created by JOnAS......342

Howto: Using the MC4J JMX Console......344
Connecting to the JMX server......344

Howto: JOnAS and JMX, registering and manipulating MBeans......345
Introduction......345
ServletContextListener......345
Configuration......349
Library Dependences......350
HibernateService Extension......350

Howto: Using JOnAS through a Firewall......353
Target Audience and Rationale......353
RMI/IIOP through a Firewall......353

How to configure and use xdoclet for JOnAS......354
Downloading and installing xdoclet......354
Ejbdoclet Ant target......354
Xdoclet tags......355

JOnAS Documentation

1. White Paper
2. Getting Started
    - JOnAS Installation
    - Running a First EJB Application
    - More Complex Examples
3. Configuration Guide
    - JOnAS Configuration Rules
    - Configuring JOnAS Environment
    - Configuring the Communication Protocol and JNDI
        - Choosing the Protocol
        - Security and Transaction Context Propagation
        - Multi−protocol Deployment (GenIC)
    - Configuring the Logging System (monolog)
    - Configuring JOnAS Services
        - Configuring Registry Service
        - Configuring EJB Container Service
        - Configuring WEB Container Service
        - Configuring WebServices Service
        - Configuring Ear Service
        - Configuring Transaction Service
        - Configuring DataBase Service
        - Configuring Security Service
        - Configuring JMS Service
        - Configuring Resource Service
        - Configuring JMX Service
        - Configuring Mail Service
        - Configuring DB Service (Hsql)
    - Configuring Security
    - Configuring JDBC DataSources
    - Configuring JDBC Resource Adapters
    - Configuring JMS Resource Adapters
4. J2EE Application Programmer's Guide
    - Principles
    - JOnAS class loader hierarchy
5. Enterprise Beans Programmer's Guide
    - Developing Session Beans
    - Developing Entity Beans
    - Developing Message−driven Beans
    - Transactional Behaviour
    - Enterprise Bean Environment
    - Security Management
    - Defining the Deployment Descriptor

- EJB Packaging

6. Web Application Programmer's Guide
    - Developing Web Components
    - Defining the Web Deployment Descriptor
    - WAR Packaging
7. J2EE Connector Programmer's Guide
    - Principles
    - Defining the JOnAS Connector Deployment Descriptor
    - Resource Adapter (RAR) Packaging
    - Use and Deployment of a Resource Adapter
    - Appendix: Connector Architecture Principles
8. J2EE Client Application Programmer's Guide
    - Launching J2EE Client Applications
    - Defining the Client Deployment Descriptor
    - Client Packaging
9. J2EE Application Assembler's Guide
    - Defining the Ear Deployment Descriptor
    - EAR Packaging
10. Deployment and Installation Guide
11. Administration Guide
12. JOnAS Commands Reference Guide
13. Advanced topics
    - Creating a New JOnAS Service
    - Using JMS in Application Components
    - Ant EJB Tasks User Manual
    - Using Login Modules in Java Clients
    - Web Services with JOnAS
    - Working with Management Beans
14. Howto Documents
    - JOnAS Versions Migration Guide
    - Installing JOnAS from Scratch
    - Installing JOnAS−Tomcat or JOnAS−Jetty from Scratch
    - How to Compile JOnAS
    - JOnAS Clustering
    - How to develop distributed message beans
    - How to develop distributed message beans with JOnAS 4.1
    - How to Use Axis
    - How to Use WebSphere MQ JMS with JOnAS
    - Web Service Interoperability between JOnAS and Weblogic
    - RMI−IIOP Interoperability between JOnAS and Weblogic
    - Interoperability between JOnAS and CORBA
    - How to Migrate the New World Cruises Application to JOnAS
    - Execute JOnAS as WIN32 Service
    - WebServices with JOnAS 3.X series
    - Install jUDDI on JOnAS

- JSR 160 Support in JOnAS
- Using the MC4J JMX Console
- JOnAS and JMX: registering and manipulating MBeans
- Using JOnAS through a Firewall
- How to configure and use xdoclet for JOnAS

Thanks to Bruno Bellamy for the JOnAS logo.

Java Open Application Server (JOnAS): a J2EE^{TM} Platform

Last modified at 2004−06−04, JOnAS 4.1

This document provides an overview of the JOnAS platform. The content of this document is the following:

- Introduction
    - J2EE
    - ObjectWeb
- JOnAS Features
    - System Requirements
    - Java Standard Conformance
    - Key Features
    - JOnAS Packages
- JOnAS Architecture
    - Communication and Naming service
    - EJB Container service
    - WEB Container service
    - Ear service
    - Transaction service
    - Database service
    - Security service
    - Messaging service
    - JCA Resources service
    - Management service
    - Mail service
    - WebServices service
- JOnAS Development and Deployment Environment
    - JOnAS Configuration and Deployment Facilities
    - JOnAS Development Environments
- Clustering and Performance
- Perspectives

Introduction

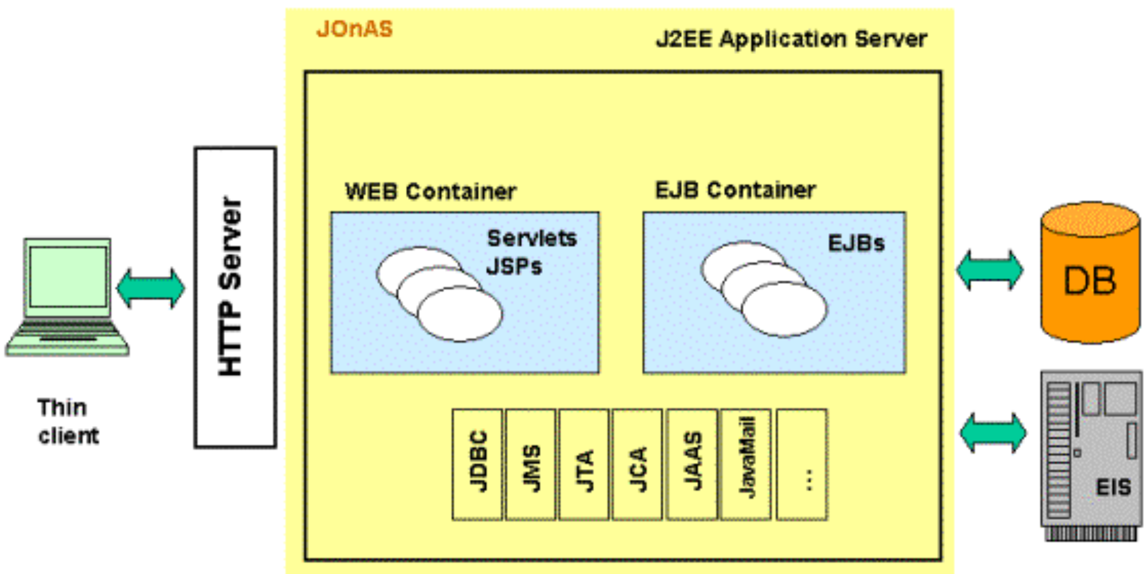
J2EE

The Sun™ J2EE specification, together with its related specifications (EJB™, JMS™,...), defines an architecture and interfaces for developing and deploying distributed Internet Java™ server applications based on a multi–tier architecture. This specification intends to facilitate and standardize the development, deployment, and assembling of application components; such components will be deployable on J2EE platforms. The resulting applications are typically web–based, transactional, database–oriented, multi–user, secured, scalable, and portable. More precisely, this specification describes two kinds of information:

- The first is the runtime environment, called a J2EE server, which 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 explaining how an application component should be developed, deployed, and used.

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

A typical J2EE application is composed of 1) presentation components, also called "web components" (Servlets and JSPs™), which define the application Web interface, and 2) enterprise components, the "Enterprise JavaBeans" (EJB), which define the application business logic and application data. The J2EE server provides containers for hosting web and enterprise components. The container provides the component life–cycle management and interfaces the components with the services provided by the J2EE server. There are two types of containers; the web container handles Servlet and JSP components, while the EJB container handles the Enterprise JavaBeans components. A J2EE server can also provide an environment for deploying Java clients (accessing EJBs); it is called client container.



ObjectWeb

JOnAS is an open source application server, developed within the ObjectWeb consortium. ObjectWeb is an open source initiative which can be compared to Apache or Linux, but in the area of ***middleware***. The aim of ObjectWeb is to develop and promote open source middleware software.

ObjectWeb is an International Consortium hosted by INRIA, officially founded in February 2002 by Bull, France Telecom, and INRIA. All software is available with the LGPL license.

The technical objective of this consortium is to develop a distributed component–based, middleware technology, in line with CORBA, Java, and W3C standards. The intent is to apply the component model, as already used at the application level in J2EE and in the CORBA Component Model, at the middleware level itself. The functional

coverage of ObjectWeb projects addresses naming, trading, communication (events, messages), availability and safety (transactions, persistence, replication, fault tolerance), load balancing, and security. Some related topics are also addressed, such as robustness, optimization, code quality, as well as benchmarks, tests, evaluations, demonstrators, and development tools.

Thus, the global ObjectWeb architectural model goes top down from applications (as benchmarks such as Rubis from the JMOB project) running on middleware platforms (such as JOnAS, OpenCCM or ProActive). The platforms are based on technical components such as Message−Oriented Middleware (JORAM which implements JMS), a communication framework (CAROL), a persistence framework (JORM), a database query framework (MEDOR), a transactional monitor (JOTM), an Object Request Broker (Jonathan). A technical component such as C−JDBC allows any platform to benefit from database clusters. JOnAS makes use of all these ObjectWeb components (JORAM, CAROL, Jonathan, JORM, MEDOR, C−JDBC, and soon JOTM), but also uses open source components from other communities, such as Tomcat or Jetty being used as a Web container, or for AXIS being used to provide the "Web Services" environment.

ObjectWeb already has a significant number of members: corporations, universities, individual members (individual membership is free). ObjectWeb members contribute to ObjectWeb orientations and participate in all ObjectWeb working groups, meetings, workshops, and conferences. The community of developers and users working with ObjectWeb components and platforms is constantly growing.

JOnAS Features

JOnAS is a pure Java, open source, application server. Its high modularity allows to it to be used as

- a J2EE server, for deploying and running EAR applications (i.e. applications composed of both web and ejb components),
- an EJB container, for deploying and running EJB components (e.g. 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 (e.g. for applications without EJB components).

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.).

Java Standard Conformance

JOnAS supports the deployment of applications conforming to J2EE 1.4 specification. Its current integration of Tomcat or Jetty 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, JavaMail™ 1.3.

Key Features

JOnAS provides the following important advanced features:

- ***Management:*** JOnAS server management uses JMX and provides a JSP/Struts–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 (e.g. for open source contributors). It also provides a way to start only the services needed by a particular application, thus saving valuable system resources. JOnAS services are manageable 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, JMS, J2EE CA), 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 the CAROL (Common Architecture for RMI ObjectWeb Layer) ObjectWeb project, which allows simultaneous support of several communication protocols:
    - *RMI* using the Sun proprietary protocol JRMP
    - *RMI* on IIOP
    - *CMI*, the "Cluster aware" distribution protocol of JOnAS
    - *Jeremie*, the RMI personality of an Object Request Broker called Jonathan, from Objectweb.

  Used with Jeremie or JRMP, JOnAS benefits from transparent local RMI call optimization.
- ***Support of "Web Services:"*** Due to the integration of AXIS, JOnAS allows J2EE components to access "Web services" (i.e., 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, SPEEDO, 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:

- ***J2EECA:*** Enterprise Information Systems (EIS) can be easily accessed from JOnAS applications. 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. For example, Bull GCOS mainframes can be accessed from JOnAS using their associated HooX connectors. Moreover, resource adapters will become the "standard" way to plug JDBC drivers (and JMS implementation, with J2EE 1.4) to J2EE platforms. A JDBC Resource Adapter is available with JOnAS, which provides JDBC PreparedStatement pooling and can be used in place of the JOnAS DBM service. A JORAM JMS Resource adapter is also available.
- ***JMS:*** JMS implementations can be easily plugged into JOnAS. They run as a JOnAS service 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, SwiftMQ, and Websphere MQ. J2EE CA Resource Adapters are also available, providing a more standard way to plug JORAM or SwiftMQ to JOnAS.

- ***JTA:*** 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 Transaction Service (OTS).

JOnAS Packages

JOnAS is available for download with three different packagings:

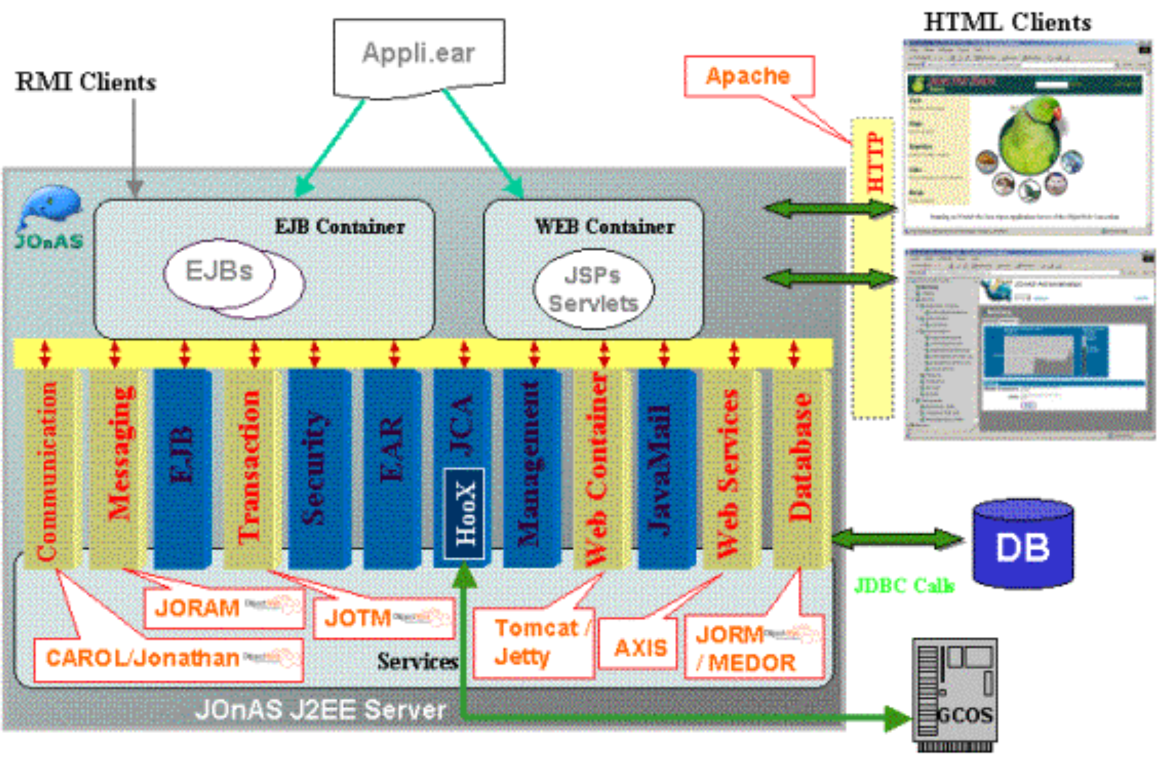
- A simple packaging contains the JOnAS application server only, without the associated Web Container implementation. To use it for J2EE Web applications, Tomcat or Jetty must be installed (with an adequate version) and it must be configured to work with JOnAS.
- A JOnAS−Tomcat package contains both JOnAS and Tomcat, pre−configured, and with compiled examples. This is a ready−to−run J2EE application server.
- A JOnAS−Jetty package contains both JOnAS and Jetty, pre−configured, and with compiled examples. This is a ready−to−run J2EE application server.

These packages also contain AXIS, thus providing pre−configured "Web Services" support.

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).



Communication and Naming Service

This service (also called "Registry") is used for launching the RMI registry, the CosNaming, the CMI registry, and/or the Jeremie registry, depending on the JOnAS configuration (CAROL configuration, which specifies which communication protocols are to be used). There are different registry launching modes: in the same JVM or not, automatically if not already running. CAROL enables multi–protocol runtime support and deployment, which avoids having to redeploy components when changing the communication protocol.

This service provides the JNDI API to application components and to other services in order to bind and lookup remote objects (e.g. EJB Homes) and resource references (JDBC DataSource, Mail and JMS connection factories, etc.).

EJB Container Service

This 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, short–lived (one method call or a client session), that 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:
    - An enterprise bean with ***Container–Managed Persistence***. In this case, 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.
    - An enterprise bean with ***Bean–Managed Persistence***. In this case, 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.

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) and MEDOR (Middleware Enabling Distributed Object Requests) 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.

WEB Container Service

This 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 or Jetty. Servlet/JSP engines are integrated within JOnAS as "web containers," i.e. such containers 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 this service be launched during JOnAS initialization. Additionally, JOnAS configuration provides a means for specifying a set of war files to be loaded. War files may also be deployed at server runtime using the JOnAS administration tools. User management for Tomcat/Jetty and JOnAS has been unified. The class−loading delegation policy (priority to the Webapp classloader or to the parent classloader) can be configured.

Servlet and JSP[TM] are technologies for developing dynamic web pages. The Servlet approach allows the development of Java classes (HTTP Servlets) that can be invoked through HTTP requests and that generate HTML pages. Typically, Servlets access the information system using Java APIs (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 J2EE components, the actual assignment of these resources is performed at component deployment time and is specified in the deployment descriptor of each component, since the component code uses logical resource names.

Ear Service

This service is used for deploying complete J2EE applications, i.e. 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, 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.

Transaction Service

This service encapsulate a Java Transaction Monitor called JOTM (a project from ObjectWeb). It is a mandatory service which 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 may 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 only allowed from Web components, session beans, or message−driven beans (only these two types of beans, which 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 behaviour 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, 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, else, 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:`*** With this attribute the client is required to call the method without a transaction context, else 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 been enhanced to provide advanced transaction features, such as nested transactions and "Web Services" transactions (an implementation of DBTP is available).

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.

Security Service

This 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, the user *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. Thus, 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. 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 login modules for Java clients. These Realms use authentication resources provided by JOnAS, which rely either 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 CRLLoginModule login module for certificate revocation.

JOnAS also implements the Java Authorization Contract for Containers (JACC 1.0) specification, allowing authorizations to be managed as java security permissions, and providing the ability to plug any security policy provider.

Messaging Service

Asynchronous EJB method invocation is possible on Message−driven Beans components. A Message−driven Bean is an EJB component that can be considered as a JMS (Java Message Service) MessageListener, i.e. which processes JMS messages asynchronously. It is associated with a JMS destination and 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.

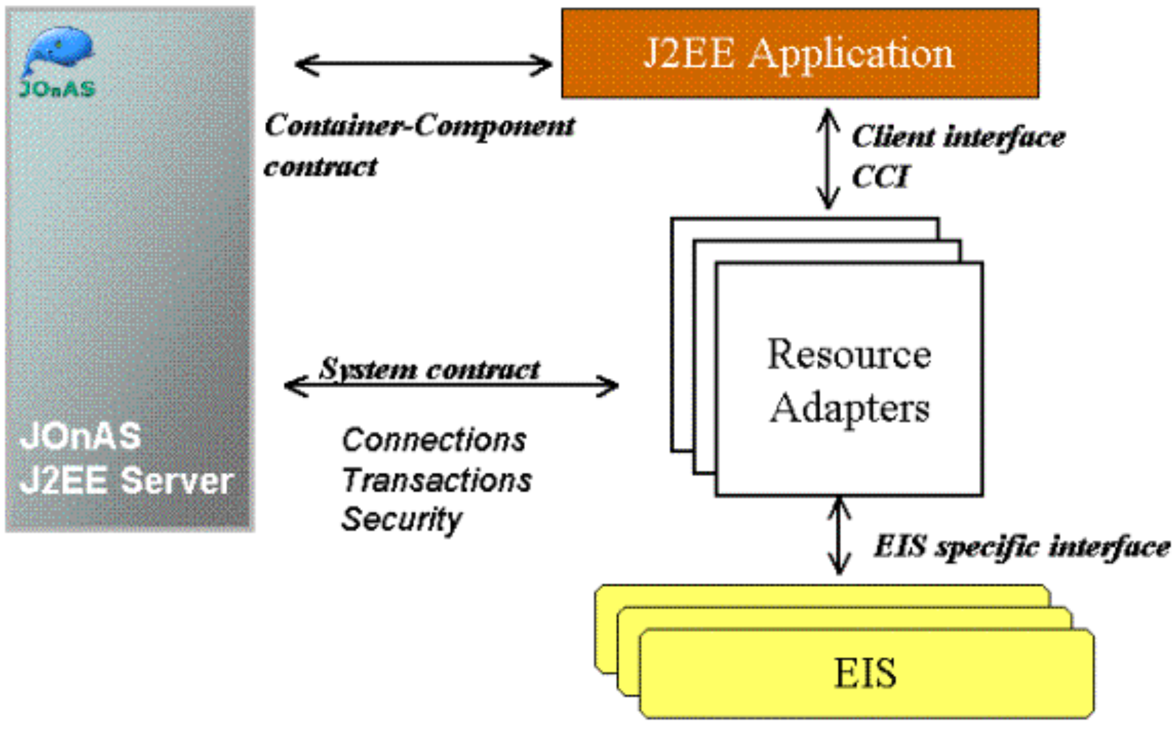
For supporting Message−driven Beans and JMS operations coded within application components, the JOnAS application server relies on a JMS implementation. JOnAS makes use of a third−party JMS implementation; currently the JORAM open source software is integrated and delivered with JOnAS, the SwiftMQ product can also be used, and other JMS provider implementations can easily be integrated. JORAM provides several noteworthy features: in particular, reliability (with a persistent mode), distribution (transparently to the JMS client, it can run as several servers, thus allowing load balancing), and the choice of TCP or SOAP as communication protocol for transmitting messages.

The JMS service is in charge of launching (or establishing 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 may 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.

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), an architecture component comparable to a software driver, which 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.



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 JORAM.

Management Service

The Management service is needed to administrate a JOnAS server from the JOnAS administration console. Each server running this service is visible from the administration console. This service is based on JMX. Standard MBeans are defined within the JOnAS application server; they expose the management methods of the instrumented JOnAS server objects such as services, containers, the server itself. These MBeans implements 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), 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, such as used memory, used threads, number of EJB instances, which component currently uses which resources, etc.. 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.

Mail Service

A J2EE application component can send e−mail messages using JavaMail™. The Mail service of the JOnAS application server provides the necessary resources to such 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*.

WebServices Service

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

JOnAS Development and Deployment Environment

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 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. A deployment tool developed by the ObjectWeb JOnAS community (earsetup) will also be available for working with the JSR88−compliant (J2EE 1.4) deployment APIs provided by the ObjectWeb Ishmael project.
- Some basic tools for the deployment itself are the JOnAS GenIC command line tool and the corresponding ANT ejbjar 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.

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) and Eclipse (JOPE and Lomboz) provide the means to develop, deploy, and debug J2EE components on JOnAS. The Xdoclet code−generation engine 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 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.

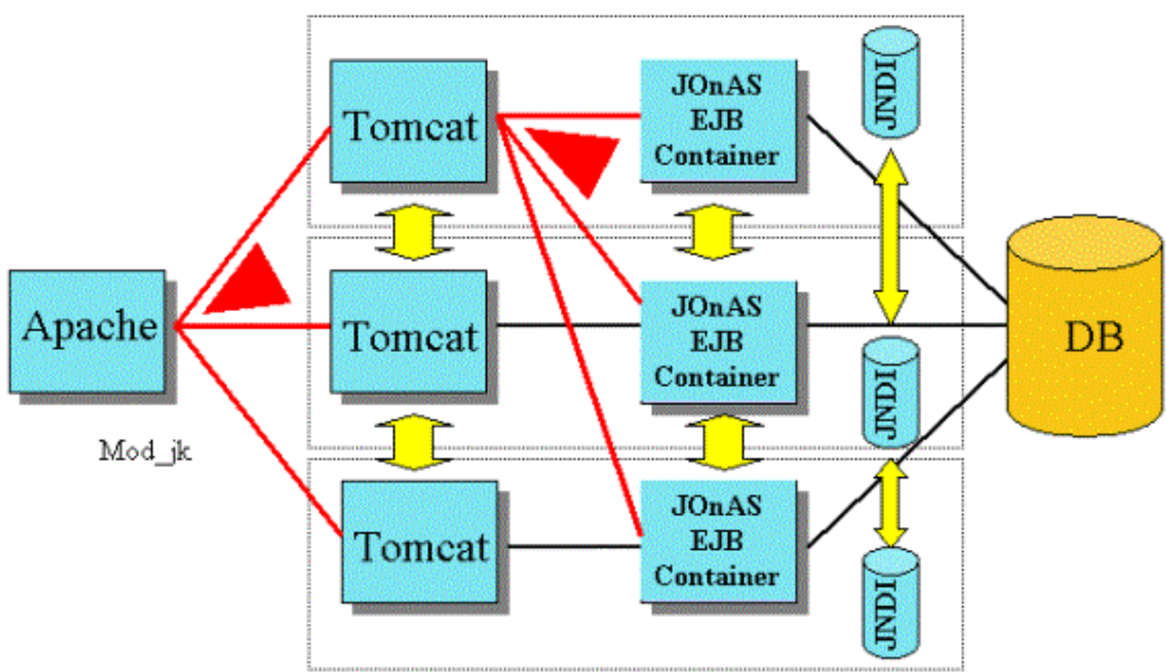
Clustering and Performance

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, HA, 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.



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 is 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. The use of C−JDBC is transparent to the application (in our case to JOnAS), since 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 mechanisms, intrinsic to the JOnAS server, for being highly scalable and efficient. This includes the following:

- 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 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.
- An optimization called "Data Prefetching" provides JDBC resultsets re−use by the EJB container (this optimization reduces the number of SQL statements executed by the EJB container).

Some benchmarks and JOnAS Use cases have already proven that JOnAS is highly scalable (e.g., refer to the Rubis results or the OpenUSS Use case). Rubis is a benchmark for e−commerce J2EE applications, which now belongs to the ObjectWeb JMOB (Java Middleware Open Benchmarking) project. OpenUSS is an operational University portal with approximately 20,000 users.

Perspectives

As an open source implementation of an application server, JOnAS is constantly evolving to satisfy user requirements and to follow the related standards. The main JOnAS evolutions currently planned are the following:

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

Sun, Java, and all Java−based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the

U.S. and other countries.

Getting started with JOnAS

1. JOnAS installation
    - Locating JOnAS
    - Downloading JOnAS
    - Setting up the JOnAS environment
2. Running a first EJB application
    - JOnAS examples
    - Building the sb example
    - Running the sb example
    - Understanding why this works
3. More complex examples
    - Other JOnAS examples
    - An example with DataBase access
        - Configuring Database access
        - Creating the table in the database
        - Configuring the classpath
        - Building the eb example
        - Running the eb example

The purpose of this tutorial is to guide the user through installing JOnAS and running a first example EJB. Guidance is also provided for running a more complex example in which an EJB has access to a database.

Additional information about JOnAS configuration may be found here.

JOnAS Installation

Locating JOnAS

The latest stable binary version of JOnAS is located at the following site :

- Objectweb

The binary version and sources are available at this site. The current (or any previous) version of JOnAS sources can be obtained via CVS.

Downloading JOnAS

The JOnAS download page allows you to choose between the JOnAS latest binary version (standalone or packaged with Tomcat or Jetty) or the latest JOnAS source code.
All of these configurations are available as *.tgz* files or *.exe* auto−installable files for windows.

Setting up the JOnAS Environment

Select a location for JOnAS installation, for example `your_install_dir`.
Be aware that if a previous version of JOnAS is already installed in this location, the new installation will overwrite previous files and configuration files that have been customized may be lost. In this case, it is recommended that these files be saved before re−starting the installation process. Note that it is also recommended to use a different location through the JONAS_BASE environment variable for customizing configuration files as explained in the Configuration Guide.

JOnAS installation consists of unzipping the downloaded files. To begin the installation process, go to the directory under which JOnAS is to be installed (e.g., `your_install_dir`) and unzip the downloaded files. You can use `gunzip` and `tar -xvf` on unix−based systems, or the *winzip* utility on Windows systems.

Once the JOnAS product is installed, the following two variables must be set in your environment to complete the installation:

- JONAS_ROOT = `your_install_dir`/JONAS where `your_install_dir` is the directory in which JOnAS is installed,
- PATH = $JONAS_ROOT/bin/unix on Unix or with %JONAS_ROOT%\bin\nt on Windows.

When using standalone packages (without Tomcat or Jetty), you **must** run '`ant install`' in the JOnAS_ROOT directory to rebuild global libraries based on your environment. This must also be done again if your web environment changes (e.g., switching from CATALINA to JETTY). Note that this step is not necessary for JOnAS−Tomcat and JOnAS−Jetty packages.

Additionally, if RMI/IIOP will be used, global libraries must be rebuilt by running '`ant installiiop`' in the JOnAS_ROOT directory. This is valid for all JOnAS packages. Also refer to the Configuration Guide about the communication protocol choice.

JOnAS is now ready to run the first EJB application.

Running a First EJB Application

JOnAS examples

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

This basic example will verify that JOnAS is correctly installed.

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.

Building the sb example

Examples should be built using the Ant build tool. Note that some Ant tasks need an additional `bcel.jar` that can be found on the BCEL site (go to the download area). If you have the *Ant build tool* installed, you can build the JOnAS examples with the *build.xml* file found in the *$JONAS_ROOT/examples* or *$JONAS_ROOT/examples/src* directories.

Running the sb example

This a distributed example in which two processes are involved:

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

To run this example:

- Run the JOnAS server:
  `jonas start`
  the following message is displayed on the standard output:
  `The JOnAS Server 'jonas' version-number is ready`
- 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`
- Run the java client in another Unix *xterm*:
  `jclient sb.ClientOp`
- Or run the java client in a Windows console:
  `jclient sb.ClientOp`
- The following output should display:
  ```
  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.
  ```
  If this output displayed, the first EJB application with JOnAS has run successfully.
- Before ending, be sure to stop the JOnAS server:
  `jonas stop`
- These instructions are also located in the **README** file in the working directory.

Understanding why this works

- To begin with, the `CLASSPATH` does not need to be set, because, when the JOnAS server is launched, the `jonas` script calls the JOnAS `bootstrap` class, which is in charge of loading all the required classes for JOnAS. Moreover, it is recommended that the CLASSPATH contain the least number of elements possible, to avoid potential conflicts.
  Also, 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. For this, server and client are using **JNDI**. The `carol.properties` file (which contains 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.

NOTE

jclient is a script that is used to make it easier the running of a pure java client in a development environment. It is not the J2EE compliant way to run a java client which is to use a J2EE container client. Examples of container client may be found in `earsample` and `jaasclient` examples of the JOnAS distribution.

More Complex Examples

Other JOnAS examples

Following are brief descriptions of other examples that are located under `$JONAS_ROOT/example/src`:

1. **The eb** example uses **Entity beans**.
   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.
2. **The lb** example uses **Entity bean 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.
3. **The jms** directory contains a **Stateful Session bean** with methods performing JMS operations and a pure JMS client message receptor.
   A complete description of this example is here in the JMS User's Guide.
4. **The mailsb** directory contains a **SessionMailer** and **MimePartDSMailer Stateful Session beans** with methods providing a way for building and sending mail.
5. **mdb/samplemdb** contains a **Message Driven bean** that listens to a topic and a 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.

6. **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.
7. **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 or Jetty (for JSP pages and servlets),
   - security.
8. **earsample** contains a complete J2EE application. This sample is a simple **stateful Session bean**, with 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 J2EE uniform naming works in the servlets.
9. **cmp2** contains an example illustrating most of the concepts of CMP 2.0.
10. **jaasclient**< contains an example of a JAAS login module that illustrates the different methods for authentication. This applies to a pure Java client.
    There are different callback handlers that demonstrate how to enter identification at a command line prompt, with a dialog box, or without any prompt (where the client uses its own login/password in the code).
11. **`j2eemanagement`** contains an example of how the J2EE Management component (MEJB) can be used to access the manageable object within the JOnAS platform. The MEJB component provides access to the J2EE Management Model as defined in the J2EE Management Specification.

The corresponding directories contain a **README** that explains how to build and run each example.

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 the README.

Before running this example:

- configure Database access,

- configure your classpath environment,
- create the table in the database used for this example (BMP and CMP 1.1 only).

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 Firebird, InstantDB, InterBase, McKoi, MySQL, Oracle or PostgreSQL databases:

- `FirebirdSQL.properties`
- `InstantDB1.properties`
- `InterBase1.properties`
- `McKoi1.properties`
- `MySQL.properties`
- `Oracle1.properties`
- `PostgreSQL1.properties`

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

For example:

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

for the `Oracle1.properties` file.

The section "Configuring Database service" in the Configuration Guide provides more details about DataSource objects and their configuration.

Creating the table in the database

`$JONAS_ROOT/examples/src/eb` contains an SQL script for Oracle, `Account.sql`, and another for InstantDB, `Account.idb`. If your Oracle server is running, or if InstantDB is properly installed and configured, you can create the table used by the example (do not create it if you are using CMP 2.0).

Example for Oracle:

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

Configuring the classpath

The JDBC driver classes must be accessible from the classpath. For that, update the `config_env` or `config_env.bat` file. In this file, set one of the following variables: `IDB_CLASSES`, `ORACLE_CLASSES`,

`POSTGRE_CLASSES` or `INTERBASE_CLASSES` with the appropriate value for your database installation.

You may also add the JDBC driver classes in your CLASSPATH, or copy the classes into `JONAS_ROOT/lib/ext`.

Building the eb example

The *Ant build tool* is used to 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 on Unix, or the `build.bat` script on Windows.

Running the eb example

Here, again, two processes are involved:

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

To run this example:

- Run the JOnAS server to make beans available to clients:
  ```
  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
  ```
- Then, run the java clients in another Unix *xterm*:
  ```
  jclient eb.ClientAccount AccountImplHome
  jclient eb.ClientAccount AccountExplHome
  ```
- Or run the java clients in Windows console:
  ```
  jclient eb.ClientAccount AccountImplHome
  jclient eb.ClientAccount AccountExplHome
  ```
- 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
```

the example *eb*. has run successfully.

- Before ending, be sure to stop the JOnAS server:

```
jonas stop
```

Configuration Guide

The content of this Configuration Guide is the following:

1. JOnAS Configuration Rules
2. Configuring JOnAS Environment
3. Configuring the Communication Protocol and JNDI
    - Choosing the Protocol
    - Security and Transaction Context Propagation
    - Multi−protocol Deployment (GenIC)
4. Configuring The Logging System (monolog)
5. Configuring JOnAS services
    - Configuring Registry service
    - Configuring EJB Container service
    - Configuring WEB Container service
    - Configuring WebServices service
    - Configuring Ear service
    - Configuring Transaction service
    - Configuring Database service
    - Configuring Security service
    - Configuring JMS service
    - Configuring Resource service
    - Configuring JMX service
    - Configuring Mail service
    - Configuring DB service (hsql)
6. Configuring Security
    - Using web container Tomcat 5.0.x interceptors for authentication
    - Using web container Jetty 5.0.x interceptors for authentication
    - Configuring mapping principal/roles
    - Configuring LDAP resource in the `jonas-realm.xml` file
    - Configuring client authentication based on clientcertificate in the web container
7. Configuring JDBC DataSources
    - Configuring DataSources
    - CMP2.0/JORM
    - JDBC Connection Pool Configuration
    - Tracing SQL Requests through P6Spy
8. Configuring JDBC Resource Adapters
    - Configuring Resource Adapters
    - CMP2.0/JORM
    - ConnectionManager Configuration
    - Tracing SQL Requests through P6Spy
    - Migration from dbm service to the JDBC RA
9. Configuring JMS Resource Adapters
    - JORAM Resource Adapter

JOnAS Configuration Rules

As described in the Getting Started chapter, 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 *JEREMIE* or *RMI/IIOP* for remote access or to work with additional java classes (for example, JDBC driver classes), additional configuration tasks must be performed, such as setting a specific port number for the registry.

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`***.

JONAS_BASE

The `JONAS_BASE` environment variable has been introduced in JOnAS version 3.1. Starting with this version, the previous configuration rule is replaced with a very basic one:

> **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 `JONAS_BASE`:

1. Perform the following actions:
   - Create a new directory and initialize `JONAS_BASE` with the path to this directory.
   - Create the following sub–directories in `$JONAS_BASE`:
     - conf
     - ejbjars
     - apps
     - webapps
     - rars
     - logs
   - Copy the configuration files located in `$JONAS_ROOT/conf` into `$JONAS_BASE/conf`. Then, modify the configuration files according to the requirements of your application, as explained in the following sections.
2. Perform the following actions:
   - Initialize `$JONAS_BASE` with a path.
   - In `$JONAS_ROOT` directory, enter: `ant create_jonasbase`. This will copy all the required files and create all the directories.

Note that 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`.

Configuring JOnAS Environment

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.

Default configuration is provided in `$JONAS_ROOT/conf/jonas.properties`. This file holds all possible properties with their default values. This configuration file is mandatory. The JOnAS server looks for this file at start time in the `$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 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.

Note that starting with the JOnAS 3.1. version, the `jonas.name` property no longer exists. The name of a server can be specified on the ***jonas*** command line using a specific option (***-n name***). If the name is not specified, the default value is **jonas**.

When several JOnAS servers must run simultaneously, it is beneficial to set a different name for each JOnAS server in order to administrate these servers via the JOnAS administration tools.

Also note that it is possible to define configuration properties on the command line: (`java -D...`).

Use the ***jonas check*** command to review the JOnAS configuration state.

Configuration scripts

The JOnAS distribution contains two configuration scripts:

- `$JONAS_ROOT/bin/unix/setenv` on Unix and `%JONAS_ROOT%\bin\nt\setenv.bat` on Windows

  This configuration script sets useful environment variables for JAVA setup (`$JAVA` and `$JAVAC`). It adds `$JONAS_BASE/conf` to the `$CLASSPATH` if `$JONAS_BASE` is set, otherwise it adds `$JONAS_ROOT/conf`. This script is called by almost all other scripts (`jclient`, `jonas`,`newbean`,`registry`,`GenIC`).
- `$JONAS_ROOT/bin/unix/config_env` on Unix and `%JONAS_ROOT%\bin\nt\config_env.bat` on Windows

  This script is used to add some required `.jar` files to the `$CLASSPATH`. This script is called by the `jonas` script.

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`, `POSTGRE_CLASSES`,`INTERBASE_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 Bootstrap class loader.

Configuring the Communication Protocol and JNDI

Choosing the Protocol

Typically, access to JNDI 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. This currently provides support for RMI/JRMP, RMI/IIOP, JEREMIE, and CMI (clustered protocol) by changing the configuration. Other protocols may be supported in the future. In versions prior to JOnAS 3.1.2, it was necessary to rebuild JOnAS for switching (e.g., from RMI/JRMP to JEREMIE), and to change the value of the OBJECTWEB_ORB environment variable; OBJECTWEB_ORB is no longer used. 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.

Supported communication protocols are the following:

- ***RMI/JRMP*** is the JRE implementation of RMI on the JRMP protocol. This is the default communication protocol.
- ***JEREMIE*** is an implementation of RMI provided by the Jonathan Objectweb project. It provides local call (RMI calls within a same JVM) optimization. *JEREMIE* has a specific configuration file ***jonathan.xml*** delivered in ***$JONAS_ROOT/conf***. Typically, no modifications are necessary; just make sure it is copied under ***$JONAS_BASE/conf*** if `$JONAS_BASE` is defined.
- ***RMI/IIOP*** is the JacORB implementation of RMI over the IIOP protocol. The configuration file of JacORB is the `$JONAS_BASE/conf/jacorb.properties` file. Since version 4.3.1, JOnAS no longer uses the JRE ORB implementation; it uses the JacORB implementation. The default iiop model now used is the POA model. 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 contents of the ***carol.properties*** file is:

```
# jonas rmi activation (jrmp, iiop, jeremie, or cmi)
carol.protocols=jrmp
```

```
#carol.protocols=cmi
#carol.protocols=iiop
#carol.protocols=jeremie
# RMI JRMP URL
carol.jrmp.url=rmi://localhost:1099

# RMI JEREMIE URL
carol.jeremie.url=jrmi://localhost:12340

# 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 1) choose the protocol through the `carol.protocols` property, 2) change `localhost` to the hostname where registry will be run, 3) 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.

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 jrmp, cmi, jeremie, iiop)
carol.protocols=jrmp,jeremie
```

Note: Before version 4.1, JOnAS could not support `jeremie` and `iiop` at the same time. Due to an incompatibility between Jeremie and rmi/iiop, there was no configuration option providing a choice between these two protocols when building a JOnAS server. By default, JOnAS only allowed a choice between `rmi/jrmp`, `Jeremie` or `cmi`. To use `rmi/iiop`, it was necessary to run an "`ant installiiop`" command under the $JONAS_ROOT directory. This resulted in a choice between `rmi/jrmp`, `Jeremie`, `rmi/iiop` or `cmi`.

This ***restriction*** does not exist starting with 4.1 release. The command "`ant installiiop`" is no longer needed. The choice can be made between `rmi/jrmp`, `Jeremie`, `rmi/iiop` or `cmi`.

Security and Transaction Context Propagation

For implementing EJB security and transactions, JOnAS typically relies on the communication layer for propagating the security and transaction contexts across the 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
```

Note that the `secpropag` attribute of the JOnAS ejbjar ANT task and the `-secpropag` option of GenIC are no longer used. They were used in JOnAS versions prior to 3.1.2 to specify that the security context should be propagated.

Multi–protocol Deployment (GenIC)

The JOnAS deployment tool (GenIC) must be notified for which protocols 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` and `Jeremie`. To change this configuration, call GenIC with the `-protocols` option specifying a comma–separated list of protocols (chosen within jeremie, jrmp, iiop, cmi), e.g.:

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

This list of protocols can also be specified for the JOnAS ejbjar ANT task:

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

Since 4.3.1 version, the default iiop model used is the POA model. Thus, GenIC should be relaunched on all previously generated beans.

Configuring the Logging System (monolog)

Starting with JOnAS 2.5, the logging system is based on Monolog, the new standard API for Objectweb projects. 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
- Using the jonas admin command or the JonasAdmin administration tool 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 the "Configuring JDBC Datasources" section.

trace.properties syntax

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

The the monolog documentation 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:

- **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, i.e. 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 it can use the **tty** handler.

Example of setting DEBUG level for the logger used in the jonas_ejb module:

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

Example for setting the output of JOnAS traces to both the console and a file `/tmp/jonas/log`:

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

Example for setting 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
```

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 the JOnAS service chapter.

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

- **registry**: this service is used for binding remote objects and resources that will later be accessed via JNDI. It is automatically launched before all the other services when starting JOnAS.
- **ejb**: the *EJB Container service* is the service that provides support for EJB application components.
- **web**: the *WEB Container service* is the service that provides support for web components (as Servlets and JSP). At this time JOnAS provides an implementation of this service for Tomcat and Jetty.
- ws: the WebServices service is the service that provides support for WebServices (WSDL publication).
- **ear**: the *EAR service* is the service that provides support for J2EE applications.
- **jmx**: this service is needed in order to administrate the JOnAS servers and the JOnAS services via a JMX−based administration console. JOnAS is using MX4J.
- **security**: this service is needed for enforcing security at runtime.
- **jtm**: the *Transaction manager service* is used for support of distributed transactions. This is the only mandatory service for JOnAS.
- **dbm**: the *Database service* is needed by application components that require access to one or several relational databases.
- **resource**: this service is needed for access to *Resource Adapters* conformant to the *J2EE Connector Architecture Specification*.
- **jms**: this service was needed by application components that used the *Java Message Service* standard API for messaging, and was mandatory for using message−driven beans until JOnAS release 4.1. Since this release, a JMS provider can be integrated through the deployment of a resource adapter.
- **mail**: the *Mail service* is required by applications that need to send e−mail messages.

These services can be managed using the `JonasAdmin` administration console, which is a Servlet−based application using the JMX technology. Note that JonasAdmin can only manage available services (currently, it is not possible to launch a service after the server start−up).

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 will be 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 following services: **dbm**, **resource**, and **jms**.
- **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, adhering to a strict naming convention: a service **XX** will be configured via a set of properties:

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

Configuring Registry Service

This service is used for accessing the RMI registry, Jeremie registry, CMI registry, or the CosNaming (iiop), depending on the configuration of communication protocols specified in ***`carol.properties`***, refer to the section "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 the `jonas.service.registry.mode` property are:

- `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,
- `automatic`: the Registry is launched in the same JVM as JOnAS Server, if not already started. This is the default value.

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

Configuring EJB Container Service

This is the primary JOnAS service, providing EJB containers for EJB components.

An EJB container can be created from an `ejb-jar` file using one of the following possibilities:

- 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 name, 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* will create a container from the `ejb-jar` file named `Bank.jar`. This file will be searched for in the **`$JONAS_BASE/ejbjars/`** directory.
- Another way to automatically create an EJB container at server start–up time is to 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.
- EJB containers may be dynamically created from `ejb-jar` files using the `JonasAdmin` tool.

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 (see Bootstrap class loader).

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/false`.
Important : This functionnality works only in local mode (server + client must be in the same filesystem)
Note that clients must have a SecurityManager set to download Stubs (if you use jclient see `jclient` script options).

When using Client Container :
Example : java –Djava.security.manager –Djava.security.policy=/url/to/java.policy –jar /path/to/client.jar app–client.jar

Configuring WEB Container Service

This service provides WEB containers for the WEB components used in the J2EE applications.
JOnAS provides two implementations of this service: one for Jetty 5.0.x, one for Tomcat 5.0.x.
It is necessary to run this service to use the `JonasAdmin` tool.

If you are not using a JOnAS/WebContainer package, 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* will create a container from the `war` file named `Bank.war`. It will search for this file in the **`$JONAS_BASE/webapps/`** directory.

Using webapp directories instead of packaging a new war file each time can improve the development process. You can replace the classes with the new compiled classes and 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 web container (Jetty or Tomcat) for the production time.
Example of using the **`jonasAdmin/ webapp directory`** instead of **`jonasAdmin.war`**

- Go to the `JONAS_BASE/webapps/autoload` directory
- Create a directory (for example, jonasAdmin) : `JONAS_BASE/webapps/autoload/jonasAdmin`
- Move the `jonasAdmin.war` file to this directory
- Unpack the war file to the current directory, then remove the war file
- At the next JOnAS startup, the webapp directory will be used instead of the of the war file. Change the jsp and see the changes at the same time.

WEB containers can be also dynamically created from `war` files using the `JonasAdmin` tool.

Configuring WebServices Service

A. Choose a Web Service Engine

At this time, only 1 implementation for WebServices is available : the Axis implementation. But in the future, a Glue implementation can be made easily.

jonas.properties :

```
#...

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

#...
```

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, the only thing to do is to define them in the jonas.properties.

Example : if you want to publish a WSDL into the local FileSystem, use the FileWSDLHandler

jonas.properties :

```
#...

# a list of comma separated WSDL Handlers
jonas.service.ws.wsdlhandlers file1
# Configuration of the file WSDL Handler
jonas.service.ws.file1.class org.objectweb.jonas.ws.handler.FileWSDLHandler
# Specify String params for WSDLHandler creation jonas.service.ws.file1.params
location
# Make sure that user that run JOnAS have rad/write access in this directory
jonas.service.ws.file1.location /path/to/directory/where/store/wsdls

#...
```

Configuring 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.

The `JonasAdmin` tool can be used to dynamically deploy the J2EE application components from an `ear` file.

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.

Configuring 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
```

Configuring Database Service

The description of the new JDBC Resource Adapters as a replacement for the Database service is located in Configuring JDBC Resource Adapters.

To allow access to one or more relational databases (e.g. Oracle, InstantDB, PostgreSQL, ...), 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 the "Configuring JDBC DataSources" section.

The following subsections briefly explain how to configure a DataSource object for your database, to be able to run

the Entity Bean example delivered with your specific platform.

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 the "Configuring JDBC Datasources" section.

Configuring Oracle for the supplied example

A template `Oracle1.properties` file is supplied in the installation directory. This file is used to define a *DataSource* object, 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 Configuring JDBC DataSources): `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, its name (Oracle1, not the JNDI 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, InstantDB1
```

To create the table used in the example with the SQL script provided in `examples/src/eb/Account.sql`, the Oracle server must be running with a JDBC driver installed (Oracle JDBC drivers may be downloaded at their Web site).
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` on Unix, or `%JONAS_ROOT%\bin\nt\config_env.bat` on Windows).

Configuring InstantDB for the supplied example

A template `InstantDB1.properties` file is supplied in the installation directory. This file is used to define a Datasource object named InstantDB1. 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 InstantDB it is `jdbc:idb=db1.prp.`
datasource.classname	Name of the class implementing the JDBC driver: `org.enhydra.instantdb.jdbc.idbDriver`

To have the EJB platform create the corresponding DataSource object, its name (InstantDB1, not the JNDI name) must be in the ***`jonas.properties`*** file, on the jonas.service.dbm.datasources line.

```
jonas.service.dbm.datasources            InstantDB1
```

InstantDB must have been properly installed and configured, using a version higher than 3.14. (examples have been tested with 3.25).

The JDBC driver classes must be accessible from the classpath. To do this, update the *config_env* file; (`$JONAS_ROOT/bin/unix/config_env` on Unix, or `%JONAS_ROOT%\bin\nt\config_env.bat` on Windows).

Create the Account database using the utility provided with InstantDB:

```
cd examples/src/eb
. $JONAS_ROOT/bin/unix/config_env
java org.enhydra.instantdb.ScriptTool Account.idb
```

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 (e.g. Informix, Sybase, SQL Server), not all JDBC drivers have been tested against JOnAS.

Configuring Security Service

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 propagation across method calls. Refer also to the "Security and Transaction Context Propagation" section. All other security configuration related to JOnAS is done in the file jonas–realm.xml and security configuration related to web containers, certificate, etc., is done in the appropriate files. Refer to the subsection "Configuring Security" for a discussion of security configuration.

Configuring JMS Service (not compliant with 2.1 MDBs)

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` config 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. This process is described in the Configuring JMS resource adapters section.

JOnAS integrates a third–party JMS implementation (JORAM) which is the default JMS service. Other JMS providers, such as SwiftMQ and WebSphere MQ, may easily be integrated as JMS services.

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

jms service is an optional service that must be started before the *ejb container service*.

Following are the properties that can be set in `jonas.properties` file for *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.mdbthreadpoolsize` is used for setting the default thread pool used Message Driven Beans (10 is the default value).
- `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. For the SwiftMQ product, obtain a *com.swiftmq.appserver.jonas.JmsAdminForSwiftMQ* class from the SwiftMQ site. For WebSphere MQ, the class to use is *org.objectweb.jonas_jms.JmsAdminForWSMQ*.

Some additional information about JMS configuration (in particular, several JORAM advanced configuration aspects) is provided in the "JMS Administration" and "Running an EJB performing JMS operations" sections of the Using JMS in application components chapter.

Information related to using WebSphere MQ is provided in the Using WebSphere MQ JMS howto.

Configuring 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 *Resource Adapter*.

The role of the *Resource service* is to deploy the *Resource Adapters* in the JOnAS server, *i.e* configure it in the operational environment and register in JNDI name space 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 **`$JONAS_BASE/rars/`** directory. This property is a comma–separated list of *resource adapter* file names ('.rar' suffix is optional).
- Another way to automatically deploy *resource adapter* files at the server start–up is to place the `RAR` files in an *autoload* directory. The name of this directory is specified using the `jonas.service.resource.autoloaddir` property in `jonas.properties` file. This directory is relative to the **`$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 Defining the JOnAS Connector Deployment Descriptor for additional information.

Configuring JMX Service

The *JMX service* must be started in order to administrate or instrument the JOnAS server via `JonasAdmin`. This 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.

Configuring Mail Service

The *Mail service* is an optional service that may be used to send email.
It is based on JavaMail™ and on JavaBeans™ Activation Framework (JAF) API.

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.

Configuring Session mail factory

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

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.
The section "Configuring a mail factory" contains a list of the available properties.

The following subsection provides information about configuring JOnAS in order to create the required mail factory objects.

Configuring JOnAS

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
```

Configuring a mail factory

The fields are 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`.
Optional properties	
Authentication properties	
mail.authentication.username	Set the username for the authentication.
mail.authentication.password	Set the password for the authentication.
javax.mail.Session.properties (refer to JavaMail documentation for more information)	
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 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.
MimePartDataSource properties (**Only used if mail.factory.type is javax.mail.internet.MimePartDataSource**)	
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.

Configuring 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
```

There is a file `HSQL1.properties` in `$JONAS_ROOT/conf` that can be used with these default values. The users are declared in the following way:

```
jonas.service.db.user   login:password
```

ie: jonas.service.db.user1 jonas:jonas A user with name jonas and password jonas has access to this database. This login and this password (jonas/jonas) are used in the `HSQL1.properties` file.

Configuring Security

Configure JAAS security for certificate authentication

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* exploits *security roles* and *method permissions* located in the EJB deployment descriptor.

Note that:

- JOnAS relies on Tomcat or Jetty for the identification of the web clients. The java clients use the JAAS login

modules for the identification. JOnAS performs the user authentication.
JOnAS provides three types of Realm which are defined in the file `$JONAS_ROOT/conf/jonas-realm.xml`:

- Memory realm: users, groups, and roles are written in the file in the section `<jonas-memoryrealm>`.
- Datasource realm: users, groups, and roles information is stored in a database; the configuration for accessing the corresponding datasource is described in the section `<jonas-dsrealm>` of the `$JONAS_ROOT/conf/jonas-realm.xml` file.
  The configuration requires the name of the datasource, the tables used, and the names of the columns.
- LDAP realm: users, groups, and roles information is stored in a LDAP directory. This is described in the section `<jonas-ldaprealm>` of the `$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.
  ie: 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.catalina50.JACC`
For Jetty, use the realm `org.objectweb.jonas.security.realm.web.jetty50.Standard`
These realms require as 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 (e.g., groups on Unix).

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 propagation across method calls. Refer also to the "Security and Transaction Context Propagation" section.

Using web container Tomcat 5.0.x interceptors for authentication

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

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                 debug="0" resourceName="UserDatabase"/>
```

By the line:

```
<Realm className="org.objectweb.jonas.security.realm.web.catalina50.JACC"
                      debug="99" 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.

Using web container Jetty 5.0.x interceptors for authentication

To use Jetty interceptors in a web application, provide a `web-jetty.xml` file under the `WEB-INF` directory in the `.war` file, in which it is specified that the security interceptor `org.objectweb.jonas.security.realm.web.jetty50.Standard` for JOnAS be used instead of the default one. Such as, for the `earsample` example:

```
<Call name="setRealmName">
  <Arg>Example Basic Authentication Area</Arg>
</Call>
<Call name="setRealm">
  <Arg>
    <New class="org.objectweb.jonas.security.realm.web.jetty50.Standard">
      <Arg>Example Basic Authentication Area</Arg>
      <Arg>memrlm_1</Arg>
    </New>
  </Arg>
</Call>
```

Several web–jetty.xml examples are located in the `earsample` example and `alarm` demo.

Configuring mapping principal/roles

JOnAS relies on the `jonas-realm.xml` file for access control to the methods of EJB components.
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.

Configuring Memory resource in `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.
Example : A developer makes two applications, one which need the role role1 and the second application which need the role role2. Instead of declaring two users, user1 with role1 and user2 with role2, a group can be declared. This group will have two roles, role1 and role2.

```
<groups>
  <group name="myApplications" roles="role1,role2" description="Roles for my applications" />
</groups>
```

Then a user named 'john' can be added with the group 'myApplications'. This user must be add in the section <users>...</users> of section <memoryrealm name="...">...</memoryrealm> of the `jonas-realm.xml` file.

```
<user name="john" password="john_password" groups="myApplications" />
```

This user will have two roles, role1 and role2 as these roles are in the group called 'myApplications'.
Another user 'joe' can be declared with the same group.

```
<user name="joe" password="joe_password" groups="myApplications" />
```

Of course, joe user can be in another group, for example group 'otherApplications'.

```
<user name="joe" password="joe_password" groups="myApplications,otherApplications" />
```

If joe must be able to authenticate on jonasAdmin web application, he need the role 'jonas'

```
<user name="joe" password="joe_password" groups="myApplications,otherApplications" roles="jonas" />
```

So, a user can have many groups and many roles defined.

Now, if the developer add a third application which need the role 'role3', the developer just have to add this role in the group named 'myApplications' and all the users from the group 'myApplications' will have this new role.

```
<groups>
  <group name="myApplications" roles="role1,role2,role3" description="Roles for my applications" />
</groups>
```

Add the memory resource in the `jonas-realm.xml` file:
Note : [...] means that it can be existing memory realms as several memoryrealm can be declared in the section <jonas–memoryrealm>
One memory–realm can be used for Tomcat, another for Jetty, or different realms can be assigned for specific contexts (in server.xml for Tomcat or web–jetty.xml, jetty.xml for Jetty).

```
  <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>
```

Note that it's better to use jonasAdmin web application to add/delete user, group, role than editing the jonas–realm.xml by the hand. It can avoid some errors of configuration.

Configuring 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`

user_name	user_pass
...	...
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`

user_name	role_name
...	...
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>
```

Configuring 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, the unique name will be: `DN uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org` for the user `'jonas_user'`.
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 methods for the authentication: the bind or the compare method.

- The bind method: 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 that this method requires the admin roles in the configuration in order to read the user passwords.*

By default, the bind method is used.
All the elements of the configuration for the ldap resource can be found on the `jonas-realm_1_0.dtd` DTD file.

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>
```

Configuring client authentication based on client certificate in the web container

1. Introduction

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

- Configure the Realm the Web container will have to use.
- Configure an SSL listener on the Web container.
- Configure the Web application to make it ask a client certificate.
- Configure the JAAS LoginModules.
- 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.

2. Configure the Realm the Web container will have to use

If you use Tomcat

If the JOnAS−Tomcat package is installed, skip this paragraph.

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

```
<Realm className="org.objectweb.jonas.security.realm.web.catalina50.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.

If you use Jetty

Edit the `web−jetty.xml` file under `WEB−INF` directory in the `.war` file to declare a Realm name and a Realm:

```
<Configure class="org.mortbay.jetty.servlet.WebApplicationContext">
...
!-- Set the same realm name as the one specified in <realm-name> in <login-config>
    in the web.xml file of your web application -->
<Call name="setRealmName">
  <Arg>Example Authentication Area</Arg>
</Call>
<!-- Set the class Jetty has to use to authenticate the user and a title name for
     the pop-up window -->
<Call name="setRealm">
  <Arg>
    <New class="org.objectweb.jonas.security.realm.web.jetty50.JAAS">
      <Arg>JAAS on Jetty</Arg>
    </New>
  </Arg>
</Call>
...
</Configure>
```

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. Configure an SSL listener on the Web container

For Tomcat

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.

For Jetty

In the global deployment descriptor of Jetty (the `jetty.xml` file), that is in the `$JETTY_HOME/conf` or `$JONAS_BASE/conf` directory (depending on whether using Jetty is being used in standalone or not), set:

```
<Configure class="org.mortbay.jetty.Server">
...
<Call name="addListener">
<Arg>
        <New class="org.mortbay.http.SunJsseListener">
                <Set name="Port">9043</Set>
                <Set name="Keystore">
                        <SystemProperty name="jonas.base" default="."/>/conf/keystore</Set>
                <Set name="Password">changeit</Set>
                <Set name="KeyPassword">jettykeypassword</Set>
                <Set name="NeedClientAuth">true</Set>
        </New>
</Arg>
</Call>
...
</configure>
```

The Jetty certificate is stored in a repository called the keystore. This is the certificate that will present Jetty when a connection is about to be established.

For more information, refer to http://jetty.mortbay.org/jetty/doc/SslListener.html.

4. Configure your Web application to make it 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:

- For both Tomcat and Jetty, ensure that the restricted Web application area is configured in the `web.xml` file in the `.war` file with a `security-constraint` declaration.
- For both Tomcat and Jetty, 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.
- For both Tomcat and Jetty, when authentication is enabled on client certificate, the user's Web browser receives the list of the trusted (by your Web application) Certification Authorities. A connection will be established with the client only if it has a certificate issued by a trusted Certification Authorities, otherwise its certificate will be rejected.
    - In Jetty, the certificates of all the trusted Certification Authorities must be imported inside the keystore specified in the SSL listener.
    - In Tomcat, 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).
- For Jetty, if the parameter `NeedClientAuth` in the `jetty.xml` file is set to true, all Web applications accessed with SSL will ask the user to present a certificate. Use different SSL listeners on different ports to have both Web applications that require client certificate and those that do not.
- For Jetty, remember to set the same `realm-name` as the one in the `web-jetty.xml` file.

For Jetty, refer to http://jetty.mortbay.org/jetty/doc/ClientCert.htmlfor more information.

5. 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.

Choose the LoginModules

org.objectweb.jonas.security.auth.spi.JResourceLoginModule

This 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.
It requires only one parameter :

- **`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.
- **`certCallback`**: Specify this optional parameter if you want this login module to ask for a certificate callback. By default it is set to false. It should be set to true when using JAAS realms with certificates.

org.objectweb.jonas.security.auth.spi.CRLLoginModule

This LoginModule 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.

It 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; another parameter pointing to that directory must be specified:
        - ◊ **`CRLsDirectoryName`**: the directory containing the CRL files (the extension for these files must be .crl).
    - ♦ `"LDAP"` : if your CRL files are stored in a LDAP repository; **this functionality is experimental**; 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.

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 `$JONAS_BASE/conf` and named `jaas.config`. To change its location and name, edit the `$JONAS_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 follows that 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 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. There are two requirements: the entry dedicated to Tomcat must be named `tomcat`, and the entry for Jetty, `jetty`. Note that everything in this file is case–sensitive.

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

- `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.

6. 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 his 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 looks like the following:

```
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`, a memory realm might contain:

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

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

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

Configuring JDBC DataSources

This section describes how to configure the Datasources for connecting the application to databases.

Configuring DataSources

For both container–managed or 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. A DataSource object identifies a database and a means to access it via JDBC (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. 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 database management server products, such as Oracle, PostGres, or InstantDB.

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 the section "Writing database access operations" of the Developing Entity Bean Guide).

A DataSource object should be defined in a file called <DataSource name>.properties (for example Oracle1.properties for an Oracle DataSource and InstantDB1.properties for an InstantDB DataSource, as delivered with the platform).

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

```
jonas.service.dbm.datasources Oracle1,InstantDB1,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 may be downloaded at the Oracle Web site.

To create an InstantDB DataSource object (for example, InstantDB1) named *jdbc_2* in JNDI, describe it as follows (in a file `InstantDB1.properties`):

```
datasource.name       jdbc_2
datasource.url        jdbc:idb=db1.prp
datasource.classname  jdbc.idbDriver
datasource.username   useless
datasource.password   useless
```

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.

The database user and password can be placed in the DataSource description (<DataSource name>.properties file) and have the Application Components use the getConnection() method, or not placed in the DataSource description and have the Application Components use the getConnection(String username, String password) method. In the resource reference of the associated datasource in the standard deployment descriptor, the <res−auth> element should have the corresponding value, i.e. Container or Application.

CMP2.0/JORM

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework. 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. This is the purpose of the ***datasource.mapper*** property.

property name	description	possible values
datasource.mapper	JORM database mapper	• 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

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. All these attributes can be reconfigured when jonas is running, with the jonas console (**`JonasAdmin`**).

property name	description	default value
jdbc.connchecklevel	JDBC connection checking level: • 0 : no check • 1 : check connection still open • 2 : call the test statement before reusing a connection from the pool	1
jdbc.connteststmt	test statement in case jdbc.connchecklevel = 2.	select 1
jdbc.connmaxage	nb of minutes a connection can be kept in the pool. After this time, the connection will be closed, if minconpool limit has not been reached.	1440 mn (= 1 day)
jdbc.maxopentime	Maximum time (in mn) 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 ensures that the pool size will not go below this limit during the datasource lifetime.	0
jdbc.maxconpool	Maximum number of connections in the pool. Limiting the max pool size avoids errors from the database.	no limit.
jdbc.samplingperiod	Sampling period for JDBC monitoring. nb of seconds between 2 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		1000

	Maximum of concurrent waiters for a JDBC Connection. This is valid only if maxconpool has been set.	

When a user requests a jdbc connection, the dbm connection manager first checks to see if a connection is already open for its transaction. If not, it 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 its connection, it is put back in the free list. Many statistics are computed (every *jdbc.samplingperiod* seconds) and can be viewed by **`JonasAdmin`**. This is useful for tuning these parameters and for seeing the server load at any time.

When a connection has been open for too long a time (*jdbc.connmaxage*), the pool will try to release it from the freelist. However, the dbm connection manager always tries to keep open at least the number of connections specified in *jdbc.minconpool*.

When the user has forgotten to close a jdbc connection, the system can automatically close it, after *jdbc.maxopentime* minutes. Note that if the user tries to use this connection later, thinking it is still open, it will return an exception (socket closed).

When a connection is reused from the freelist, it is possible to verify that it is still valid. This is configured in *jdbc.connchecklevel*. The maximum level is to try a dummy statement on the connection before returning it to the caller. This statement is configured in *jdbc.connteststmt*.

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.

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

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

Example:

Datasource properties file content:

```
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 JONAS_BASE/conf/spy.properties file:

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

Within JONAS_BASE/conf/trace.properties:

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

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 that are compliant to the J2EE Connector Architecture specification. The resulting functionality is the same, and the benefit is the management of pools of JDBC PrepareStatements. This section describes how the JDBC Resource Adapters should be configured to connect the application to databases.

Configuring Resource Adapters

For both container−managed or bean−managed persistence, the JDBC Resource Adapter(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 J2EE Connector Programmer's Guide. 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. These can be configured using a JDBC−compliant driver for some relational database management server products, such as Oracle, PostGres, or MySQL.

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, i.e. via the resource manager connection factory references (refer to the example in the section "Writing Database Access Operations" of the Developing Entity Bean Guide).

An RAR file must be configured and deployed (e.g., 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 J2EE Connector Programmer's Guide.

```
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 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_2.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.

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_2.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_2.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: 1) put it in the jonas–ra.xml file in the RAR file and have the Application Components use the getConnection() method, or 2) not have it in the RAR file and have the Application Component use the getConnection(String username, String password) method.

CMP2.0/JORM

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework. 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.

property name	description	possible values
mapperName	JORM database mapper	• 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 • rdb.sqlserver: mapper for MS SQL Server • Refer to JORM documentation for a complete updated list.

ConnectionManager Configuration

Each RAR uses a connection manager that can be configured via the additional properties described in the following table. The Postgres1.jonas–ra.xml file provides an example of the settings. These settings all have default values and they are not required.

pool–params elements

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 milliseconds to keep the connection	0 (unlimited)
pstmt–max	Maximum number of PreparedStatements cached per connection	10

jdbc–conn–params elements

property name	description	default value
jdbc–check–level	JDBC connection checking level	0 (no check)
jdbc–test–statement	test statement	

jdbc–test–statement is not used when jdbc–check–level is equal to 0 or 1.

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:

- Update the appropriate RAR file's jonas–ra.xml file by setting the `dsClass` property to `com.p6spy.engine.spy.P6SpyDriver`.
- Set the `realdriver` property in the `spy.properties` file (located in $JONAS_BASE/conf) to the jdbc driver of your actual database.
- 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_2.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-valu
  </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 $JONAS_BASE/conf/spy.properties file:

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

In $JONAS_BASE/conf/trace.properties:

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

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 RAConfig description for a complete description.

```
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 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 to 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 RA, which is configured to identify and access a JMS server. 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 J2EE Connector Programmer's Guide.

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

JORAM Resource Adapter

The JORAM resource adapter archive (`joram_for_jonas_ra.rar`) is provided with the JOnAS distribution. It is located in the `$JONAS_ROOT/rars` directory. To deploy it, the archive file can be declared in the `jonas.properties` file, as follows:

```
jonas.service.resource.resources        joram_for_jonas_ra
```

jms must be removed from the list of services:

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

The archive can also be deployed through the JonasAdmin tool, or placed in the JOnAS' `rars/autoload` directory.

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 which:

- 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 an `a3servers.xml` file located in the `$JONAS_ROOT/conf` directory.
- Creates managed JMS `ConnectionFactory` instances and binds them with the names ***CF***, ***QCF***, and ***TCF***.
- Creates administered objects for this server (JMS *destinations* and non−managed *factories*) as described by the ***joram−admin.cfg*** file, located in the `$JONAS_ROOT/conf` directory; those objects are bound with the names ***sampleQueue***, ***sampleTopic***, ***JCF***, ***JQCF***, and ***JTCF***.

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

The default configuration may, of course, be modified.

Configuring the JORAM adapter

`jonas-ra.xml` is the JOnAS−specific deployment descriptor that configures the JORAM adapter. Changing the configuration of the RA requires extracting and editing the deployment descriptor, and updating the archive file. The ***`RAConfig`*** utility is provided for doing this (refer to RAConfig description for a complete description). To extract `jonas-ra.xml`, do the following:

```
RAConfig joram_for_jonas_ra.rar
```

Then, to update the archive, do the following:

```
RAConfig -u jonas-ra.xml joram_for_jonas_ra.rar
```

The `jonas-ra.xml` file sets the central configuration of the adapter, defines and sets managed connection factories for outbound communication, and defines a listener for inbound communication.

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 `joram-admin.cfg` files are located.	• Any String describing an absolute path (ex: /myHome/myJonasRoot/conf). • Empty String, files will be searched in `$JONAS_ROOT/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).
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).
AdminFile	Name of the file describing the administration tasks to perform; if the file does not exist, or is not found, no administration task is performed.	• Name of the file (ex: myAdminFile.cfg). • Nothing (default joram−admin.cfg 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).

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 (ex: myQueueConnectionFactory).
UserName	Default user name that will be used for opening JMS connections.	• Any name (ex: myName). • Nothing (default anonymous name will be set).
UserPassword	Default user password that will be used for opening JMS connections.	• Any name (ex: 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).

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 (ex: joramActivationSpec).

Configuring a collocated JORAM server

The ***`a3servers.xml`*** file describes a JORAM platform configuration and is used by a starting JORAM server (thus, it is never used if JORAM is in non−collocated mode).

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.

Specifying administration tasks

The ***`joram-admin.cf`***file describes administered objects to be (optionally) created when deploying the adapter.

The default file provided with JOnAS creates a queue bound with name ***sampleQueue***, a topic bound with name ***sampleTopic***, sets the anonymous user, and creates and binds non−managed connection factories named ***JCF***, ***JQCF*** and ***JTCF***. It also defines a host name and server port, which have the same meanings as the parameters set in the `jonas-ra.xml` file. Their goal is to allow host and port values to be easily changed without having to edit the deployment descriptor.

For requesting the creation of a queue with name "myQueueName", simply add the following line:

```
Queue        myQueueName
```

For requesting the creation of a topic with name "myTopicName", simply add the following line:

```
Topic        myTopicName
```

For requesting the creation of the user "myName" − "myPass", simply add the following line:

```
User          myName myPass
```

For requesting the creation of a non−managed ConnectionFactory, to be bound with name "myCF", simply add the following line:

```
CF          myCF
```

For requesting the creation of a non−managed QueueConnectionFactory, to be bound with name "myQCF", simply add the following line:

```
QCF          myQCF
```

For requesting the creation of a non−managed TopicConnectionFactory, to be bound with name "myTCF", simply add the following line:

```
TCF          myTCF
```

To be noted:

- All administration tasks are performed locally, meaning 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.

Undeploying and deploying again 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 the server will be retrieved in its pre−undeployment 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 the intent is to start a brand new JORAM server, its persistence directory should be removed. This directory is located in JOnAS' running directory and has the same name as the JORAM server (for example, `s0/` for server "s0").

J2EE Application Programmer's Guide

Target Audience and Content

The target audience for this guide is the application component provider, i.e. the person in charge of developing the software components on the server side (the business tier).

The content of this guide is the following:

1. Target Audience and Content
2. Principles
    - Enterprise Bean Creation
    - Web Components Creation
    - J2EE Application Assembler
    - Application Deployer and Administrator
3. JOnAS class loader hierarchy
    - Understanding class loader hierarchy
    - Commons class loader
    - Application class loader
    - Tools class loader
    - Tomcat class loader
    - JOnAS class loaders
    - Conclusion

Principles

JOnAS supports two types of J2EE application components: **Enterprise Beans** and **Web components**. In addition to providing guides for construction of application components, guides are supplied for application assembly, deployment, and administration.

Enterprise Bean Creation

The individual in charge of developing Enterprise Beans should consult the Enterprise Beans Programmer's Guide for instructions on how to perform the following tasks:

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

This JOnAS documentation provides guides for developing the three types of enterprise bean:

- Session beans
- Entity beans

- Message driven beans

Deployment descriptor specification is presented in the Defining the Deployment Descriptor chapter.

More specific issues related to transaction behavior, the Enterprise Bean environment, or security service, are presented in the corresponding chapters: Transactional Behaviour, Enterprise Bean Environment, Security Management.

Principles and tools for providing EJB JAR files are presented in the chapters EJB Packaging and Deployment and Installation Guide.

Web Components Creation

Web designers in charge of JSP pages and software developers providing servlets can consult the Web Application Programmer's Guide.

The 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 the Defining the Web Deployment Descriptor chapter.

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 WAR Packaging and the Deployment and Installation Guide.

J2EE Application Assembler

The application assembler in charge of assembling the application components already bundled in EJB JAR files and WAR files into a J2EE EAR file, can obtain useful information from the J2EE Application Assembler's Guide chapter.

Application Deployer and Administrator

JOnAS provides tools for the deployment and administration of Enterprise Beans (EJB JARs), Web applications (WARs), and J2EE applications (EARs).

The Deployment and Installation Guide covers issues related to the deployment of application components.

The Administration Guide presents information about how to manage the JOnAS server and the JOnAS services that allow deployment of the different types of application components: EJB Container service, Web Container service, and EAR service.

JOnAS class loader hierarchy

This section describes a new and important key feature of the J2EE integration: the class loader hierarchy in JOnAS.

Understanding 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 (this concerns local references in the same JVM).

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 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.

Commons class loader

The `commons` class loader is a JOnAS−specific class loader that will load all classes required to start the JOnAS server. 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 apps, 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` directory. You can extend this class loader by adding your own JARs inside this directory. If you are using a Jetty packaging, `JONAS_ROOT/lib/jetty` will be loaded too.

Note that the *lib/ext* extension mechanism is now deprecated. You should place additional JARs directly in the classloader directory (`commons`, `apps`, `tools`).

To have a library available for each component running inside JOnAS, add the required JAR files in the `JONAS_ROOT/lib/ext` directory or in the `JONAS_BASE/lib/ext`. The jars in `JONAS_BASE/lib/ext` are loaded first, followed by the jars in `JONAS_ROOT/lib/ext`. All jars in subordinate directories will also be loaded. If a specific jar is needed only for a web application (i.e., need to use a version of a jar file which is different than a version loaded by JOnAS), change the compliance of the web application classloader to the java 2 delegation model. Refer to the following: WEB class loader.
It is also possible to use the extension mechanism, which is described in the section dependencies of the J2EE specification (section 8.1.1.28)

Application class loader

The application class loader is a JOnAS−specific class loader that will load all application classes required by the user applications. This implies that this loader will load all single RAR files. Thus, all applications have the visibility of the resource adapters classes. This class loader has the commons class loader as parent class loader.

The JARs loaded by the application class loader are located under the JONAS_ROOT/lib/apps directory and under JONAS_ROOT/lib/catalina/common/lib directory. (CATALINA_HOME/common/lib if you are not using the Tomcat package). You can extend this class loader by adding your own JARs inside this directory.

Tools class loader

The tools class loader is a JOnAS−specific class loader that will load all classes for which applications do not require visibility. (User applications will not have the ability 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 parent class loader.

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

Tomcat class loader

The tomcat class loader is a class loader that will load all classes of the tomcat server. (CATALINA_HOME/server/lib directory). The classes of the common directory of tomcat (CATALINA_HOME/common/lib directory) are loaded by the application classloader and not by this tomcat classloader. Applications have the visibility of the classes and not the server classes. To have the visibility of the server classes, the context must have the privileged attribute set to true. This class loader has the application class loader as 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.

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:

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 JOnAS classes visible to it.

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.

WEB class loader

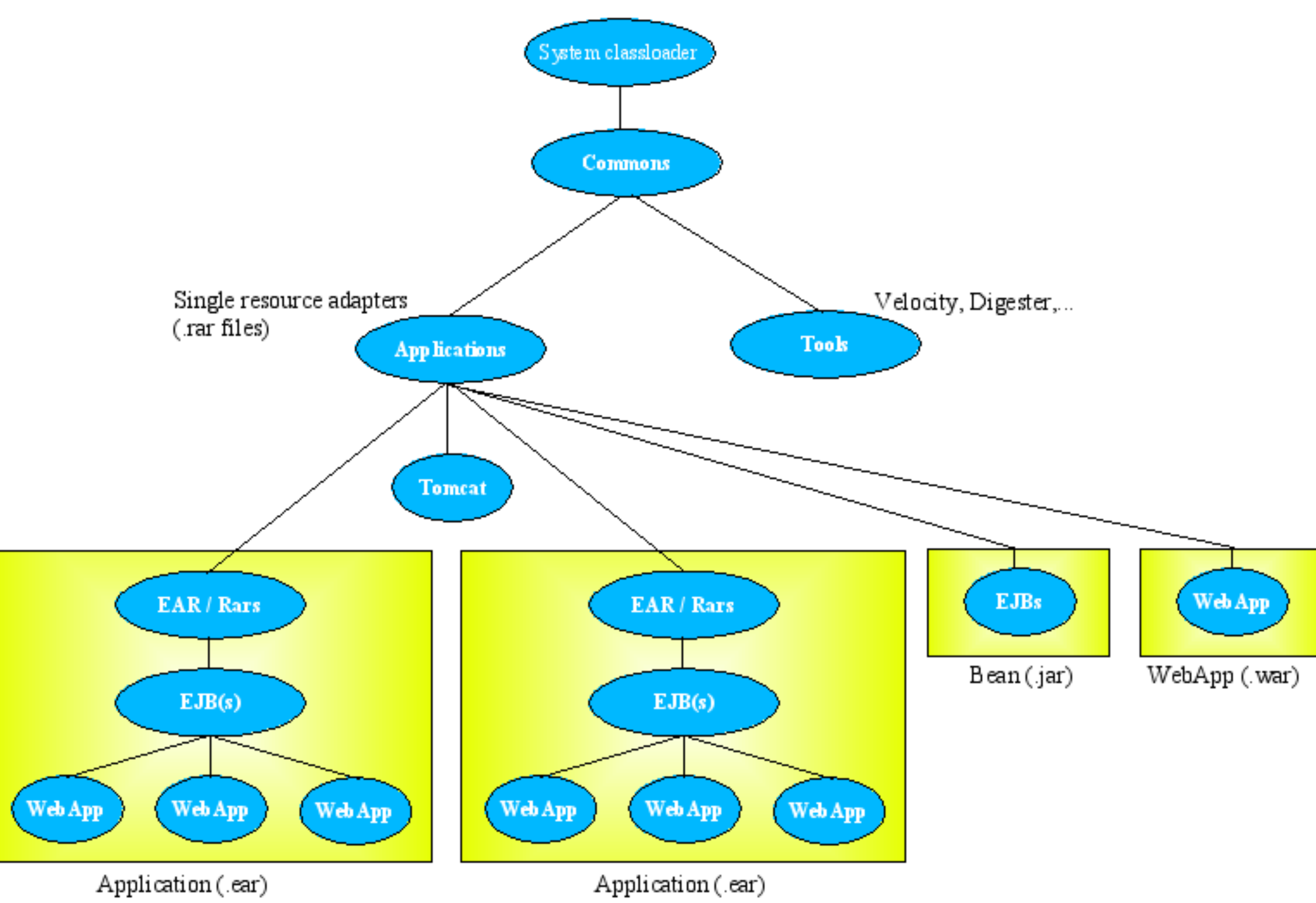
The WEB class loader is responsible for loading the Web components. There is one WEB class loader per WAR file, and this class loader is the child of the EJB class loader. Using this class loader hierarchy (the EJB class loader is the parent of the WEB class loader) eliminates the problem of visibility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

Conclusion

The resulting JOnAS class loader hierarchy is as follows:


System classloader
Commons
Single resource adapters
(.rar files)
Applications
Velocity, Digester,...
Tools
Tomcat
EAR / Rars
EJB(s)
Web App
Web App
Web App
Application (.ear)
EAR / Rars
EJB(s)
Web App
Web App
Web App
Application (.ear)
EJBs
Bean (.jar)
Web App
WebApp (.war)

EJB Programmer's Guide: Developing Session Beans

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side and, more specifically, the Session Beans.

The content of this guide is the following:

1. Target Audience and Content
2. Introduction
3. The Home Interface
4. The Component Interface
5. The Enterprise Bean Class
6. Tuning Stateless Session Bean Pool

Introduction

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 following sections.

Note: in this documentation, the term "Bean" always means "Enterprise Bean."

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.

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.

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;
}
```

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. The interface must extend the `javax.ejb.EJBObject` interface if it is remote, or the `javax.ejb.EJBLocalObject` if it is local. 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).

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 ();
}
```

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.

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;

// 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() {
        // Nothing to do for this simple example

        // 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;
    }
}
```

Tuning 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. The purpose of this value is to keep JOnAS scalable. The policy is the following:
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.

example

```
<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>
```

EJB Programmer's Guide: Developing Entity Beans

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side, and more specifically the Entity Beans.

The content of this guide is the following:

1. Target Audience and content
2. Introduction
3. The Home Interface
4. The Component Interface
5. The Primary Key Class
6. The Enterprise Bean Class
7. Writing Database Access Operations (bean−managed persistence)
8. Configuring Database Access for Container−managed Persistence
9. Tuning Container for Entity Bean Optimizations
10. Using CMP2.0 Persistence
    - Standard CMP2.0 Aspects
        - Entity Bean Implementation Class
        - Standard Deployment Descriptor
    - JOnAS Database Mappers
    - JOnAS Database Mapping (Specific Deployment Descriptor)
        - Specifying and Initializing the Database
        - CMP fields Mapping
        - CMR fields Mapping to primary−key−fields (simple pk)
        - CMR fields Mapping to composite primary−keys

Introduction

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 which 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 that, according to the EJB 2.0 specification, the couple "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.

The description of these elements is provided in the following sections.

Note that in this documentation, the term "Bean" always means "Enterprise Bean."

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. 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 the section "Configuring JDBC DataSources."

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 only needs to provide the desired interface; the container will automatically provide the implementation. The interface must extend the `javax.ejb.EJBHome` interface if it is remote, or the `javax.ejb.EJBLocalHome` interface if it is local. 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 (same number and types of arguments, but different return types).

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, only for a remote interface).

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 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.

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 the section "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>*" (e.g. `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.

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.

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;
}
```

The Component Interface

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).

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;
}
```

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 must be followed:

- 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.

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) {

            ...

      }
```

}

Special case: Automatic generation of primary keys field

There are two ways to manage the automatic primary key with JOnAS. The first method is closer to what is described in the EJB specification, i.e. an automatic PK is a hidden field, the type of which is not known even by the application. The second method is to declare a usual PK CMP field of type java.lang.Integer as automatic. The two cases are described below.

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 even by the application. All that is necessary is to stipulate in the standard deployment descriptor that this EJB has an automatic PK, by specifying java.lang.Object as primkey−class. The primary key will be completely hidden to 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.

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 will generate a database column with 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, using the specific <automatic−pk−field−name> element, as follows (this 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-
<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>
```

2) CMP field as automatic primary key (was already in JOnAS 3.3.x)

The idea is to declare a usual PK CMP field of type java.lang.Integer as automatic, then it no longer appears in create methods and its value is automatically generated by the container at EJB instance creation time. But it is still a cmp field, with getter/setter methods, accessible from the application. Example:

In the standard DD, there is a usual 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>
```

and in the JOnAS−specific Deployment Descriptor, it should be specified 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 are not sequential.

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 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>*** *(...); // 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 the section "Configuring database access for container−managed persistence." The EJB 2.0 specification defines a standard way to describe these finder methods, i.e. in the standard deployment descriptor, as an EJB−QL query. Also refer to the section "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.

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.

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();
    }
}
```

CMP 2.0

```
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();
}
```

```
}
```

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 the section "JDBC DataSources configuration").

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.
- Lookup the datasource in the enterprise bean environment using the JNDI interface (refer to the section "Enterprise Bean's 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.

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 bloc of the try statement.

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>
</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 (for an explicit mapping definition, refer to the "JOnAS Database Mapping" section of the Using CMP2.0 persistence chapter):

```
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
  <cleanup>create</cleanup>
</jdbc-mapping>
```

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 behaviour of the implementation of a *find<method>* method (i.e. 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 &gt; ?</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 &gt; 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 behaviour of the implementation of a *find<method>* method is located in **the standard deployment descriptor**, as an EJB−QL query (i.e. 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 &gt; ?1</ejb-ql>
    </query>
```

The datatypes supported for container–managed fields in CMP 1.1 are the following:

Java Type	JDBC Type	JDBC driver Access methods
boolean	BIT	getBoolean(), setBoolean()
byte	TINYINT	getByte(), setByte()
short	SMALLINT	getShort(), setShort()
int	INTEGER	getInt(), setInt()
long	BIGINT	getLong(), setLong()
float	FLOAT	getFloat(), setFloat()
double	DOUBLE	getDouble(), setDouble
byte[]	VARBINARY or LONGVARBINARY (1)	getBytes(), setBytes()
java.lang.String	VARCHAR or LONGVARCHAR (1)	getString(), setString()
java.lang.Boolean	BIT	getBoolean(), setObject()
java.lang.Integer	INTEGER	getInt(), setObject()
java.lang.Short	SMALLINT	getShort(), setObject()
java.lang.Long	BIGINT	getLong(), setObject()
java.lang.Float	REAL	getFloat(), setObject()
java.lang.Double	DOUBLE	getDouble(), setObject()
java.math.BigDecimal	NUMERIC	getBigDecimal(), setObject()
java.math.BigInteger	NUMERIC	getBigDecimal(), setObject()
java.sql.Date	DATE	getDate(), setDate()
java.sql.Time	TIME	getTime(), setTime()
java.sql.Timestamp	TIMESTAMP	getTimestamp(), setTimestamp()
any serializable class	VARBINARY or LONGVARBINARY (1)	getBytes(), setBytes()

(1) The mapping for String will normally be VARCHAR, but will turn into LONGVARCHAR if the given value exceeds the driver's limit on VARCHAR values. The case is similar for byte[] and VARBINARY and LONGVARBINARY values.

For CMP 2.0, the supported datatypes depend on the JORM mapper used.

Tuning Container for Entity Bean Optimizations

JOnAS must make a compromise between scalability and performance. Towards this end, we have introduced several tags in the JOnAS−specific deployment descriptor. For most applications, there is no need to change the default values for all these tags. See `$JONAS_ROOT/xml/jonas-ejb-jar_4_0.xsd` for a complete description of the JOnAS−specific deployment descriptor.

Note that if several of these elements are used, they should appear in the following order within the <jonas−entity> element:

1. is−modified−method−name
2. passivation−timeout
3. inactivity−timeout
4. shared
5. prefetch
6. max−cache−size
7. min−pool−size
8. lock−policy

lock−policy

The JOnAS ejb container is able to manage 5 different lock−policies :

- `container-serialized` (default): The container ensures the transaction serialization. This policy is suitable for most entity beans, particularly if the bean is accessed only from this container (shared = false).
- `container-read-committed`: This policy is similar to container−serialized, except that accesses outside transaction do not interfere with transactional accesses. This can avoid deadlocks in certain cases, when accessing a bean concurrently with and without a transactional context. The only drawback of this policy is that it consumes more memory (2 instances instead of 1).
- `container-read-uncommitted`: all methods share the same instance (like container−serialized), but there is no synchronization. For example, this policy is of interest for read−only entity beans, or if the bean instances are very rarely modified. It will fail if 2 or more threads try to modify the same instance concurrently.
- `database`: Let the database deal with transaction isolation. With this policy, you can choose the transaction isolation in your database. This may be of interest for applications that heavily use transactional read−only operations, or when the flag shared is needed. It does not work with all databases and is not memory efficient.
- `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.

Important: 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 can use container–read–uncommitted.

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 lock–policy is `container-serialized`, and `true` in the other cases.

prefetch

This is a CMP2–specific option. The default is false. This can be set to true if it is desirable to have a cache managed after finder methods execution, in order to optimize further accesses inside the same transaction.

Important note :

- The prefetch will not be used for methods that have no transactional context.
- It is impossible to set the prefetch option if the lock policy is container–read–uncommitted.

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". To save memory, this value should be set very low if you know that instances will not be reused.

min–pool–size

This optional integer value represents the number of 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.

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 that this is useless with CMP2 entity beans, since this will be done automatically by the container.

Example

The bean implementation manages a boolean `isDirty` and implements a method that returns the value of this 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.

passivation−timeout

Entity bean instances are passivated at the end of each transaction and reactivated at the beginning of the next transaction, when the `shared` flag has been set to true. In the case where `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:

1. When the bean is unloaded from the server, at least when the server is stopped.
2. When a transaction is started on this instance.
3. 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 non−mandatory tag <passivation−timeout>. Example:

```
<jonas-entity>
  <ejb-name>Item</ejb-name>
  <passivation-timeout>5</passivation-timeout>
  .....
</jonas-entity>
```

This entity bean will be passivated every five second, if not accessed within transactions. This flag is also used to set the timeout to passivate beans modified inside transactions when max−cache−size has been reached.

inactivity−timeout

Bean passivation sends the state of the bean to persistent storage and removes from memory only the bean instance objects that are holding this state. All container objects handling bean access (remote object, interceptors, ...) 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; this can be achieved through the <inactivity−timeout> element. This element is used to save memory when a bean has not been used for a long period of 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 it, then the client will receive an exception.

Using CMP2.0 persistence

This section highlights 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.

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.

Entity Bean Implementation Class

The EJB implementation class 1) implements the bean's business methods of the component interface, 2) implements the methods dedicated to the EJB environment (the interface of which is explicitly defined in the EJB specification), and 3) 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 the section "Developing Entity Beans" of the JOnAS documentation.

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>
```

Note that for running CMP1.1−defined entity beans on an EJB2.0 platform, such as JOnAS 3.x, ***you must introduce this*** <cmp−version> ***element in your deployment descriptors, since the default cmp−version value (if not specified) is 2.x***.

Note that for CMP 2.0, the information defining the behaviour 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.

Finder method example 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 &gt; ?1</ejb-ql>
</query>
```

JOnAS Database mappers

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework. 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.

property name	description	possible values
datasource.mapper	JORM database mapper	• `rdb`: generic mapper (JDBC standard driver ...)

		• `rdb.firebird`: Firebird • `rdb.mckoi`: McKoi Db • `rdb.mysql`: MySQL • `rdb.oracle8`: Oracle 8 and lesser versions • `rdb.oracle`: Oracle 9 • `rdb.postgres`: PostgreSQL (>= 7.2) • `rdb.sapdb`: Sap DB • `rdb.sqlserver`: MS Sql Server • `rdb.sybase`: Sybase

Contact the JOnAS team 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 (i.e. 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.

JOnAS Database Mapping (Specific Deployment Descriptor)

The mapping to the database of entity beans and their relationships may be specified 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 (i.e. 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.

Specifying and Initializing the Database

For specifying the database within which a CMP 2.0 entity bean is stored, the `jndi-name` element of the `jdbc-mapping` is necessary. 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, 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.

It may be useful for testing purposes to delete the database data each time a bean is loaded. For this purpose, 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>
```

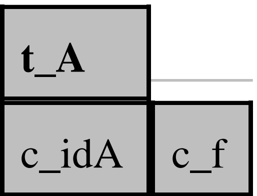
CMP fields Mapping

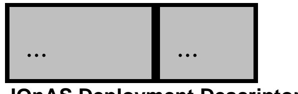
Mapping CMP fields in CMP2.0 is similar to that of CMP 1.1, but in CMP2.0 it is also possible to specify the SQL type of a column. Usually this SQL type is used if JOnAS creates the table (`create` value of the `cleanup` element), and if the JORM default chosen SQL type is not appropriate.

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>
.....
```

Database Mapping





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>
.....
```

Defaults values:

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.

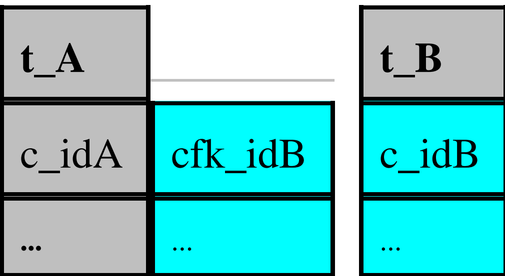
CMR fields Mapping to primary–key–fields (simple pk)

1–1 unidirectional relationships

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>
.....
```

Database Mapping



There is a foreign key in the table of the bean that owns the CMR field.

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.

Default values:

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.

1−1 bidirectional relationships

Compared to 1−1 unidirectional relationships, there is a CMR field in both of the beans, thus making two types of mapping possible.

Standard Deployment Descriptor

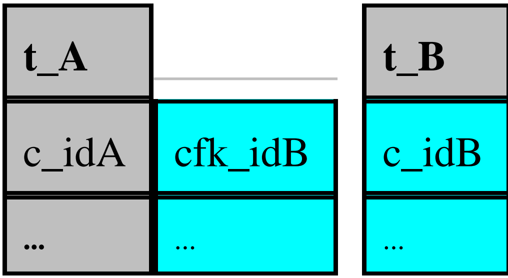
```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</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>
  .....
```

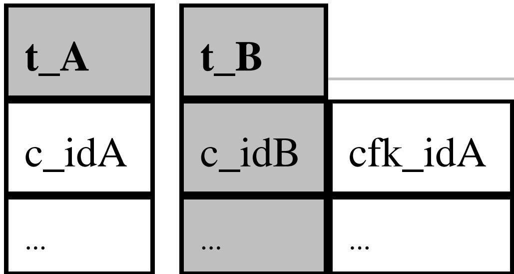
Database Mapping

Two mappings are possible. One of the tables may hold a foreign key.

Case 1:



Case 2:



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.

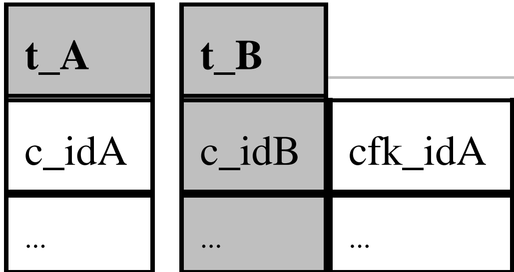
1–N unidirectional relationships

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>
.....
```

Database Mapping



In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in the table of the source bean of the relationship role with multiplicity many), t_B.

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>
.....
```

Default values:

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 (i.e. the source bean of the relationship role with multiplicity one), suffixed by _, and by its primary–key–field. **A_ida** in the example.

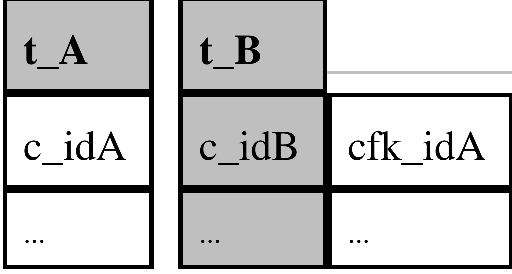
1–N bidirectional relationships

Similar to 1–N unidirectional relationships, but with a CMR field in each bean.

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>
.....
```

Database mapping



In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in the table of the source bean of the relationship role with multiplicity many), t_B.

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>
.....
```

Default values:

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 (i.e. the source bean of the relationship role with multiplicity one), suffixed by _, and by its primary–key–field. **A_ida** in the example.

N–1 unidirectional relationships

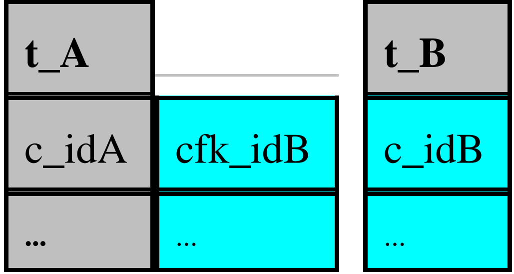
Similar to 1–N unidirectional relationships, but the CMR field is defined on the "many" side of the relationship, i.e. on the (source bean of the) relationship role with multiplicity "many."

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>
  .....
```

Database mapping



In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in table of the source bean of the relationship role with multiplicity many), t_A.

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>
  .....
```

Default values:

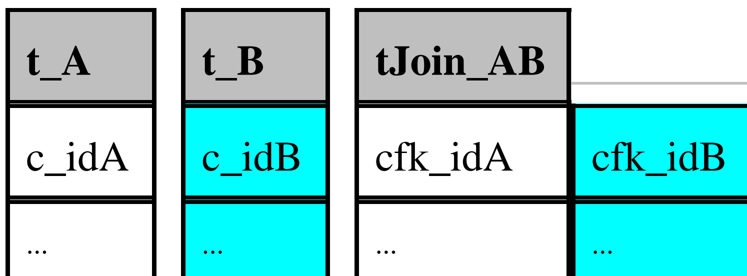
jonas–ejb–relation	Optional
foreign–key–jdbc–name	Optional. Default value is the abstract–schema–name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _, and by its primary–key–field. **B_idb** in the example.

N–M unidirectional relationships

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>
.....
```

Database mapping



In this case, there is a join table composed of the foreign keys of each entity bean table.

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>
  .....
```

Default values

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.

N–M bidirectional relationships

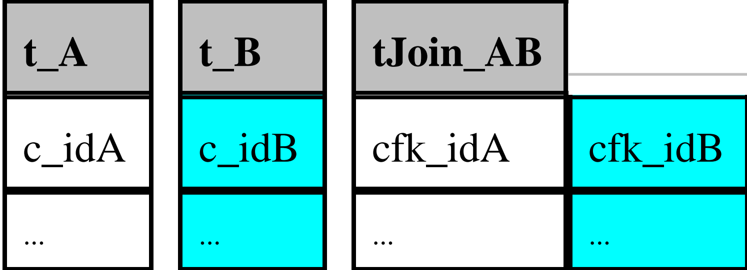
Similar to N–M unidirectional relationships, but a CMR field is defined for each bean.

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>
  .....
```

Database mapping



In this case, there is a join table composed of the foreign keys of each entity bean table.

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>
  .....
```

Default values:

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.

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.

1−1 bidirectional relationships

Standard Deployment Descriptor

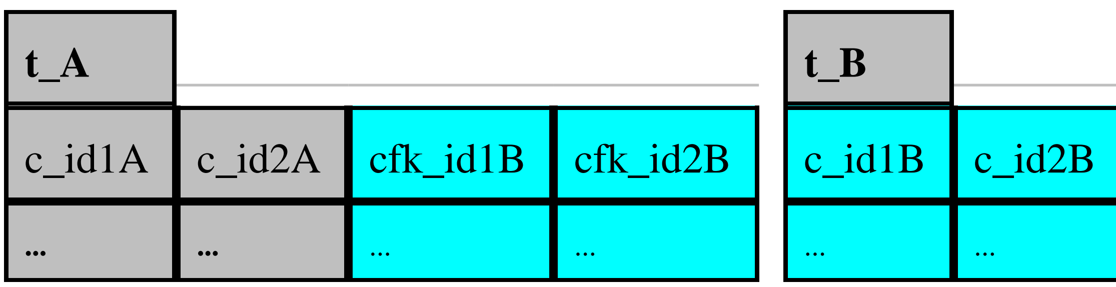
```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <prim-key-class>p.PkA</prim-key-class>
  .....
  <cmp-field>
    <field-name>id1A</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2A</field-name>
  </cmp-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <prim-key-class>p.PkB</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>
.....
```

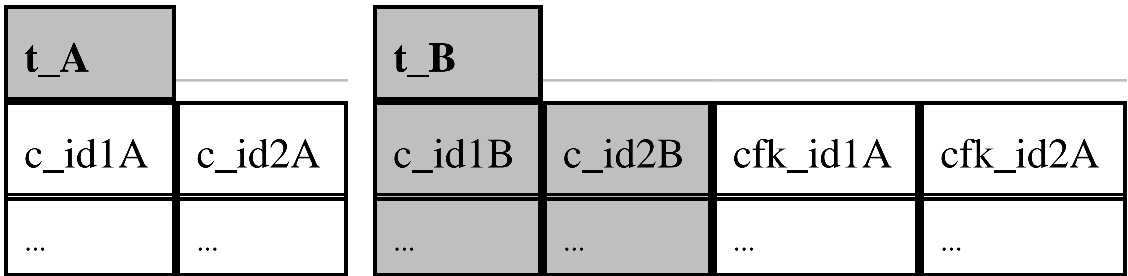
Database mapping

Two mappings are possible, one or another of the tables may hold the foreign key.

Case 1:



Case 2:



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.

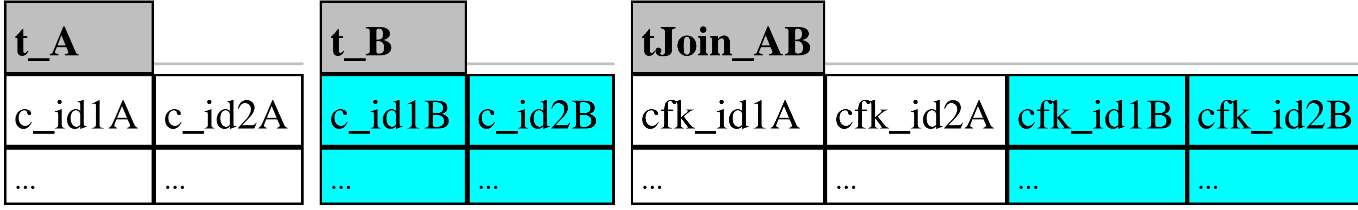
N–M unidirectional relationships

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>
  .....
```

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.

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>
  .....
```

EJB Programmer's Guide: Message−driven Beans

The content of this guide is the following:

1. Description of a Message−driven Bean
2. Developing a Message−driven Bean
3. Administration aspects
4. Running a Message−driven Bean
5. Transactional aspects
6. Example
7. Tuning Message−driven Bean Pool

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, which 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.

Description of a Message−driven Bean

A Message−driven Bean is an EJB component that can be considered as a JMS MessageListener, i.e. 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, i.e. 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 the "Transactional aspects" section 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), i.e. 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.

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, e.g. 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 to an MDB is specified in the deployment descriptor of the bean. A destination is a JMS−administered object, accessible via JNDI. The description of an MDB in the EJB 2.0 deployment descriptor contains the following elements, which are specific to MDBs:

- the JMS acknowledgment 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 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 assigned in the JMS client program sending the messages, as follows:

```
      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 the JMS User's Guide (Writing JMS operations within an application component / Accessing the destination object section).

Administration aspects

It is assumed at this point that the JOnAS server will make use of an existing JMS implementation, e.g. Joram, 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, refer to the 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 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 may execute concurrently on behalf of the same MDB. The default size of the pool of thread is 10, and it may be customized via the jonas property ***jonas.service.ejb.mdbthreadpoolsize***, which is specified in the ***jonas.properties*** file as in the following example:

```
jonas.service.ejb.mdbthreadpoolsize   50
```

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). Refer to the section "Launching the Message−Oriented Middleware."
- 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 (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.

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 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 localhost) 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 Joram, 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
```

- ***Change Joram default configuration***

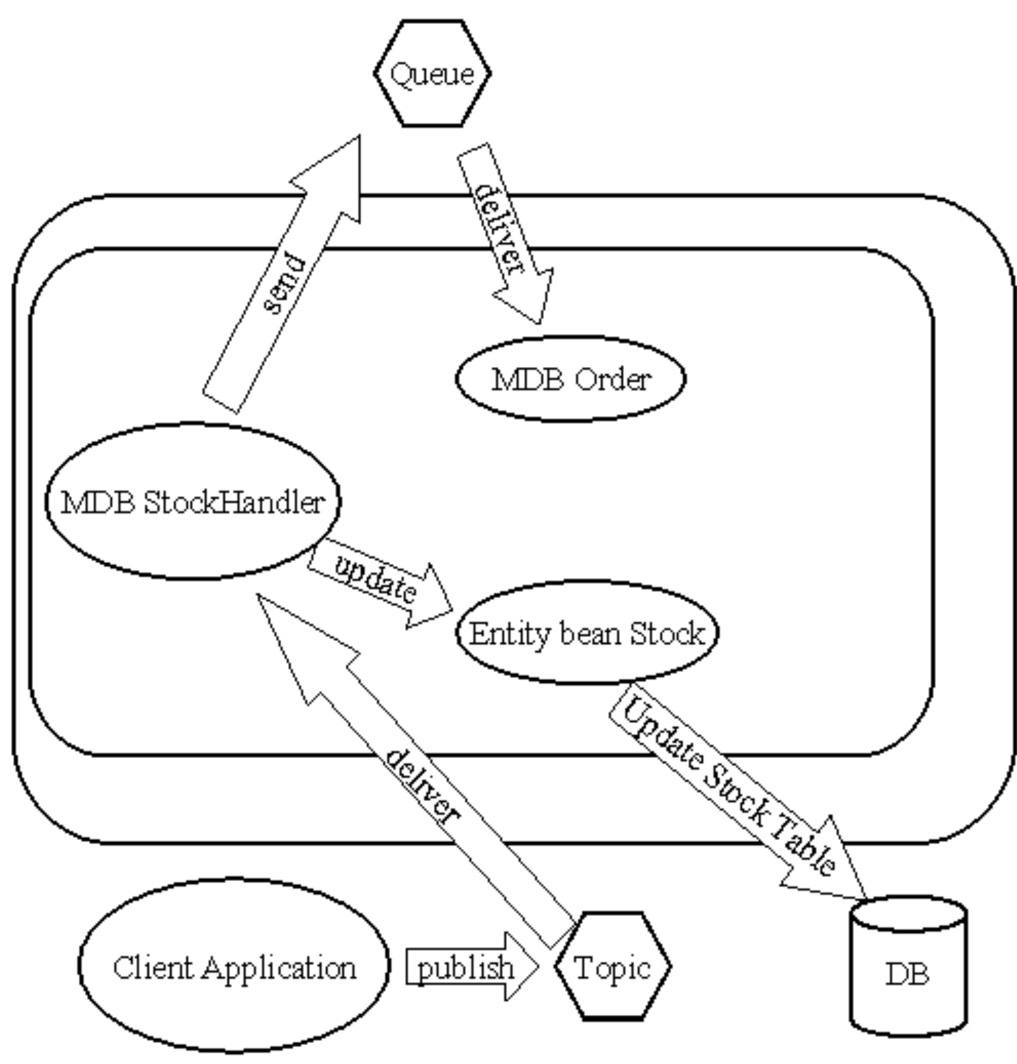
  As mentioned previously, the default host or default connection port number may need to be changed. This requires modifying the ***a3servers.xml*** configuration file provided by the JOnAS delivery in JONAS_ROOT/conf directory. For this, JOnAS must be configured with the property ***jonas.service.jms.collocated*** 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 (e.g., distribution, multi−servers), refer to the Joram documentation on http://joram.objectweb.org.

Transactional aspects

Because a transactional context cannot be carried by a message (according to the EJB 2.0 specification), 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 transaction. In the first case, container−managed transaction, the container will start a new transaction before de−queueing 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 accessed 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).

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.


Queue
send
deliver
MDB Order
MDB StockHandler
update
Entity bean Stock
deliver
Update Stock Table
Client Application
publish
Topic
DB

There are two Message−driven beans in this example:

- ***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.
- ***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 ***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 client application ***SampleAppliClient*** is a JMS Client that sends several messages on the topic *StockHandlerTopic*. It uses Map messages with three fields: "CustomerId," "ProductId," "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. Eleven messages are sent, the corresponding transactions are committed, and the last message sent causes the transaction to be rolled back.

Compiling this example

To compile `examples/src/mdb/sampleappli`, use *Ant* with the `$JONAS_ROOT/examples/src/build.xml` file.

Running this example

The default configuration of the JMS service in `jonas.properties` is the following:

```
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 not already existing.

- Run the JOnAS Server.

  ```
  jonas start
  ```
- Deploy the sampleappli container.

```
jonas admin -a sampleappli.jar
```

- Run the EJB client.

```
jclient sampleappli.SampleAppliClient
```

- Stop the server.

```
jonas stop
```

Tuning 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 of *ServerSession* that may be created in memory. The purpose of this value is to keep JOnAS scalable. The policy is the following:
When the *ConnectionConsumer* ask for a *ServerSession* instance (in order to deliver a new message) JOnAS try to give a instance from the *ServerSessionPool*. If the pool is empty, a new instance is created only if the number of yet created instances is smaller than max−cache−size parameter. When the max−cache−size is reached the *ConnectionConsumer* is blocked and it cannot deliver new messages until a *ServerSession* is eventually returned in the pool. A *ServerSession* is pushed into the pool at the end of *onMessage* method.
The default value is no limit (this means that a new instance of *ServerSession* is always created when the pool is empty).

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>
```

EJB Programmer's Guide: Transactional Behaviour

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how to define the transactional behaviour of an EJB application.

The content of this guide is the following:

1. Target Audience and Content
2. Declarative Transaction Management
3. Bean−managed Transaction
4. Distributed Transaction Management

Declarative Transaction Management

For container−managed transaction management, the transactional behaviour 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 behaviour for all the methods of the bean, or to define the behaviour 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, else, 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	−

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.

Bean−managed Transaction

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 example of the session bean).

```
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.

Distributed Transaction Management

As explained in the previous section, the transactional behaviour 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, i.e. the JTM must be launched previously in another JVM,
- if set to `false`, the JTM is local, i.e. it will run in the same JVM as the EJB Server.

Example:

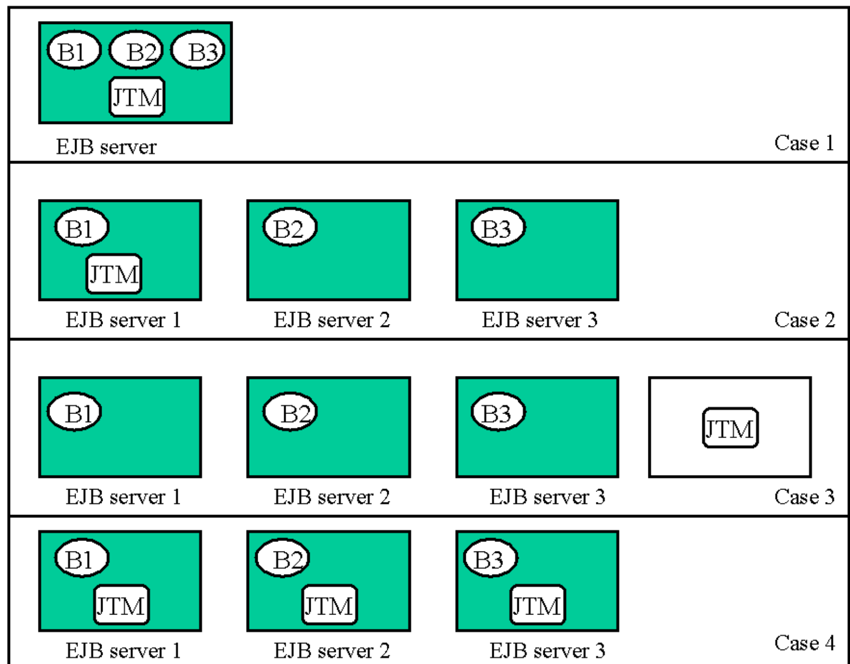
```
jonas.service.ejb.descriptors       Bean1.jar, Bean2.jar
jonas.service.jtm.remote            false
```

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.



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 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 pointers:

- 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 favours local transaction management;
- if the beans should run on the same machine, but require different server configurations, case 2 is a good approach.

EJB Programmer's Guide: Enterprise Bean Environment

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how an enterprise component can refer to values, resources, or other components in a way that is configurable at deployment time.

The content of this guide is the following:

1. Target Audience and Content
2. Introduction
3. Environment Entries
4. Resource References
5. Resource Environment References
6. EJB References
7. Deprecated EJBContext.getEnvironment() method

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 (i.e. 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/".

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>
```

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>
```

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>
```

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-local-ref>
```

If the referred bean is defined in the same ejb−jar or EAR file, the optional **ejb−link** element of the ejb−ref or ejb−local−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, i.e. 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 should be the name of the target bean, prefixed by the name of the containing ejb−jar file followed by '#' (e.g. "My_EJBs.jar#bean1"). 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 was not in the same ejb−jar file as SSA, but in another file named product_ejbs.jar, the ejb−link element would have been:

```
    <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. In this case, to resolve the reference, 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);
```

Deprecated EJBContext.getEnvironment() method

JOnAS provides support for EJB 1.0–style definition of environment properties. EJB1.0 environment must be declared in the **ejb10–properties** sub–context. For example:

```
<env-entry>
  <env-entry-name>ejb10-properties/foo</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>foo value</env-entry-value>
</env-entry>
```

The bean can retrieve its environment with the following code:

```
SessionContext ctx;
Properties prop;
public void setSessionContext(SessionContext sc) {
    ctx = sc;
    prop = ctx.getEnvironment();
}
public mymethod() {
    String foo = prop.getProperty("foo");
    ...
}
```

EJB Programmer's Guide: Security Management

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It explains how security behavior should be defined.

The content of this guide is the following:

1. Target Audience and Content
2. Introduction
3. Declarative Security Management
4. Programmatic Security Management

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.

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.

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>
```

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 is allowed to 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>
...
```

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) ;
   ...
}
```

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.

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.

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-nameOp</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.

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 (e.g. groups on Unix systems). However, this last mapping step is not currently available in JOnAS.

EJB Programmer's Guide: Defining the Deployment Descriptor

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how the bean provider should build the deployment descriptors of its components.

The content of this guide is the following:

1. Target Audience and Content
2. Principles
3. Example of Session Descriptors
4. Example of Container–managed Persistence Entity Descriptors (CMP1.1)
5. Tips

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 is to provide an XML deployment descriptor that conforms to the deployment descriptor's XML schema as defined in the EBJ specification version 2.1. (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 contain structural information for each enterprise bean that includes the following:

- 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.

Example of Session Descriptors

```
<?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>
```

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>
      <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>
```

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

EJB Programmer's Guide: EJB Packaging

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how the bean components should be packaged.

The content of this guide is the following:

1. Target Audience and Content
2. Principles

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.

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
```

Web Application Programmer's Guide

Target Audience and Content

The target audience for this guide is the Web component provider, i.e. the person in charge of developing the Web components on the server side. It describes how the Web component provider should build the deployment descriptors of its Web components and how the web components should be packaged.

The content of this guide is the following:

1. Target Audience and Content
2. Developing Web Components
    - Introduction
    - The JSP Pages
    - The Servlets
    - Accessing an EJB from a Servlet or JSP page
3. Defining the Web Deployment Descriptor
    - Principles
    - Examples of Web Deployment Descriptors
    - Tips
4. WAR Packaging

Developing Web Components

Introduction

A Web Component is a generic term which 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 container* service. Web components can be integrated in a J2EE application by packing the `.war` file in an `.ear` file (refer to the J2EE Application Programmer's Guide).

The JOnAS distribution includes a Web application example: `The EarSample example.`

The directory structure of this application is the following:

etc/xml	contains the web.xml file describing 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

The bean directory is not needed if beans coming from another application will be used.

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™ and the Quickstart guide for more details.

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`, where the .jsp files communicate with the EJB via a proxy java bean `ViewProxy.java`.

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 JavaTM Servlet Technology and the Servlets tutorial.

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.

```
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();
    }
}
```

Accessing an EJB from a Servlet or JSP page

Through the JOnAS *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 that all the following code examples are taken from the `The EarSample example` provided in the JOnAS distribution.

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 thru 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>
```

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>
```

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 the env-entry</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>
```

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

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 TM Servlet Specification Version 2.4. (Refer to `$JONAS_ROOT/xml/web-app_2_4.xsd` or http://java.sun.com/xml/ns/j2ee/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 bean's 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: The name of the context on which the application will be deployed should be specified. 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 (i.e, the context−root is /jadmin for jadmin.war).

If the context−root is / or empty, the web application is deployed as ROOT context (i.e., 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.

Examples of Web Deployment Descriptors

- Example of a standard Web Deployment Descriptor (web.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/we
         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>
```

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

WAR Packaging

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 a *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 JOnAS class loader hierarchy.

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 Configuring Web Container Service for information about how to use directories for the web application.

J2EE Connector Programmer's Guide

The content of this guide is the following:

1. Target Audience and Content
2. Principles
3. Defining the JOnAS Connector Deployment Descriptor
4. Resource Adapter (RAR) Packaging
5. Use and Deployment of a Resource Adapter
6. JDBC Resource Adapters
7. Appendix: Connector Architecture Principles

Target Audience and Content

This chapter is provided for advanced JOnAS users concerned with EAI (Enterprise Application Integration) and using the J2EE Connector Architecture principles (refer to the Appendix for an introduction to the connectors). The target audience for this guide is the Resource Adapter deployer and programmer. It describes the JOnAS specific deployment file (*jonas−ra.xml*) and the sample code to access deployed RARs.

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 for creating the *jonas−ra.xml* file is to use the RAConfig tool. For a complete description refer to RAConfig.

Defining the JOnAS Connector Deployment Descriptor

The *jonas−ra.xml* contains JOnAS specific information describing deployment information, logging, pooling, jdbc connections, and RAR config 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: Jndiname 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 RAR is a *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 needed with the JDBC RA of JOnAS or another database vendor's RAR. Value of 0 is unlimited and −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.

Deployment Descriptor Examples

The following portion of a *jonas−ra.xml* file shows the 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>
```

Resource Adapter (RAR) Packaging

Resource Adapters are packaged for deployment in a standard Java programming language Archive file called an *RAR* file (Resource Adapter ARchive). This file can contain the following:

Resource Adapters' deployment descriptor

The RAR file must contain the deployment descriptors, which are made up of:

- ◊ The standard xml deployment descriptor, in the format defined in the J2EE 1.4 specification. Refer to `$JONAS_ROOT/xml/connector_1_5.xsd` or http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd. This deployment descriptor must be stored with the name *META−INF/ra.xml* in the RAR file.
- ◊ The JOnAS−specific XML deployment descriptor in the format defined in *$JONAS_ROOT/xml/jonas−ra_X_Y.xsd*. This JOnAS deployment descriptor must be stored with the name *META−INF/jonas−ra.xml* in the RAR file.

Resource adapter components (jar)

One or more jars which contain the java interfaces, implementation, and utility classes required by the resource adapter.

Platform−specific native libraries

One or more native libraries used by the resource adapter

Misc

One or more html, image files, or locale files used by the resource adapter.

Before deploying an RAR file, the JOnAS−specific XML must be configured and added. Refer to the **`RAConfig section`** for information.

Use and Deployment of a Resource Adapter

Accessing Resource Adapter involves the following steps:

- 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 to the actual JNDI name of the *connection factory* (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, it must be closed using the `close` method on the `Connection` interface:

```
cx.close();
```

- ♦ The resource adapter must be deployed before being used by the application. Deploying the resource adapter requires the following:
    - ◊ 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 **RAConfig** that is capable of building this XML file from an RA deployment descriptor inside an RAR file. 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 getter and setter method for each of its supported properties (as it is required in the Connector Architecture specification).

After configuring the jonas–ra.xml file created above, it can be added to the resource adapter by executing the following:

```
RAConfig -u jonas-ra.xml ra
```

This will add the xml file to the ra.rar file, which is now ready for deployment.

- The JOnAS *resource service* must be configured and started at JOnAS launching time:
  In the `jonas.properties` file:
  - Verify that the name `resource` is included in the `jonas.services` property.
  - Use one of the following methods to deploy an RAR file:
    - The names of the *resource adapter* files (the '.rar' suffix is optional) must be added in the list of Resource Adapters to be used in the `jonas.service.resource.resources` property. If the '.rar' suffix is not used on the property, it will be used when trying to allocate the specified Resource Adapter.

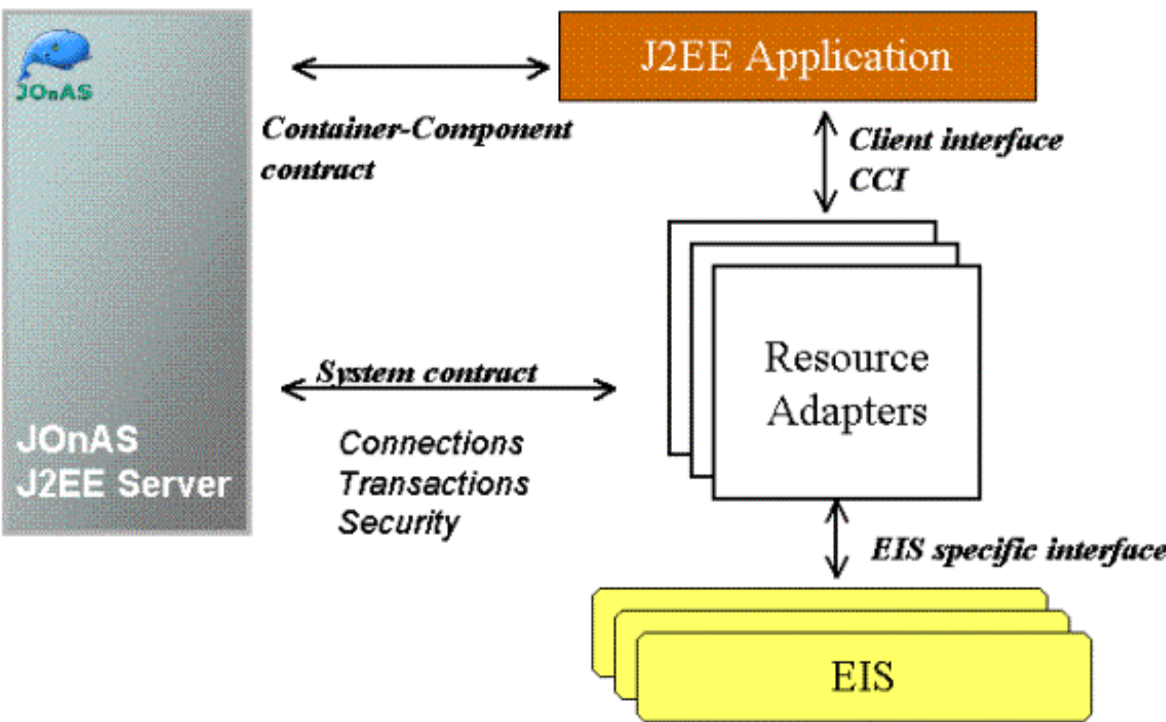
      ```
      jonas.service.resource.resources  MyEIS.rar, MyEIS1
      ```
    - Place the RAR file in the connectors autoload directory of $JONAS_BASE, default value is $JONAS_BASE/rars/autoload. Note that it may be different if `jonas.service.resource.autoload` in jonas.properties is configured differently.
    - Add the RAR via the "`jonas admin -a xxx.rar`" command.
    - Add the RAR via the JonasAdmin console.

JDBC Resource Adapters

These generic JDBC resource adapters are supplied by JOnAS and are a replacement for the DBM service. Refer to **Configuring JDBC Resource Adapters** for a complete description and usage guide.

Appendix: Connector Architecture Principles

The Java Connector Architecture allows the connection of different Enterprise Information Systems (EIS) to an application server such as JOnAS. It defines a way for enterprise applications (based on EJB, servlet, JSP or J2EE clients) to communicate with existing EIS. This requires the use of a third party software component called "Resource Adapter" for each type of EIS, which should be previously deployed on the application server. The Resource Adapter is an architecture component comparable to a software driver, which connects the EIS, the application server, and the enterprise application (J2EE components in the case of JOnAS as application server). The RA provides an interface (the Common Client Interface or CCI) to the enterprise application (J2EE components) for accessing the EIS. The RA also provides standard interfaces for plugging into the application server, so that they can collaborate to keep all system–level mechanisms (transactions, security, and connection management) transparent from the application components.



The resource adapter plugs into JOnAS and provides connectivity between the EIS, JOnAS, and the application. 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).

J2EE Client Application Programmer's Guide

Target Audience and Content

The target audience for this guide is the Client component provider, i.e. the person in charge of developing the Client components on the client side. It describes how the Client component provider should build the deployment descriptors of its Client components and how the client components should be packaged.

The content of this guide is the following:

1. Target Audience and Content
2. Launching J2EE Client Applications
    - Launching clients
    - Configuring client container
    - Examples
3. Defining the Client Deployment Descriptor
    - Principles
    - Examples of Client Deployment Descriptors
    - Tips
4. Client Packaging
    - Principles

Launching J2EE Client Applications

Launching clients

The J2EE client application can be

- a standalone client in a .jar file,
- a client bundle in an .ear file. An ear can contain many java clients.
- a class name which must be found in the CLASSPATH.

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 will launch 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.
example : `java -jar client.jar myApplication.ear`

or `java -jar client.jar myClient.jar`.

Configuring client container

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, use the `jar -uf client.jar carol.properties` command.

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.
You can replace the file in the `client.jar` with the `jar -uf client.jar traceclient.properties` command.

Classpath configuration

Some jars/classes can be added to the client container. For example if a class requires some extra libraries/classes, the option `-cp path/to/classes` can be used.

The classloader of the client container will use the libraries/classes provided by the `-cp` flag.

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.

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.

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. The second example demonstrates the use of a standalone client.

Defining the Client Deployment Descriptor

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

Examples of Client Deployment Descriptors

- Example of a standard Client Deployment Descriptor (application−client.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 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-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>
```

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

Client Packaging

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 1.4 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.

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.
  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.
  Refer to the clientjars target.

J2EE Application Assembler's Guide

Target Audience and Content

The target audience for this guide is the Application Provider and Assembler, i.e. the person in charge of combining one or more 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.

The content of this guide is the following:

1. Target Audience and Content
2. Defining the Ear Deployment Descriptor
    - Principles
    - Simple example of Application deployment Descriptors
    - Advanced example of Application deployment Descriptors with alternate DD and security
    - Tips
3. EAR Packaging

Defining the Ear Deployment Descriptor

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` or http://java.sun.com/xml/ns/j2ee/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:

- The Alarm demo
- The Cmp2 example
- The EarSample example
- 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.

Simple example of Application Deployment Descriptor

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

<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>
```

Advanced example of Application Deployment Descriptors with alternate DD and security

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

<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>
```

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

EAR Packaging

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 which 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 classes of EJBs accessed by the web components in the *WEB−INF/lib* directory.
Details about this class loader hierarchy are described in JOnAS class loader hierarchy.

The EJB components (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. See

$JONAS_ROOT/xml/application_1_4.xsd. This deployment descriptor must be stored with the name *META−INF/application.xml* in the ear file.

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 *
```

Deployment and Installation Guide

Target audience

The target audience for this guide is the application deployer.

The content of this guide is the following:

1. Deployment and installation process principles
2. Example of deploying and installing an EJB using an ejb–jar file
3. Deploying and installing a Web application
4. Deploying and installing a J2EE application

Deployment and installation process principles

The deployment and installation of Enterprise Beans

This guide assumes that the Enterprise Bean provider followed the Enterprise Beans Programmer's Guide and packaged the beans's classes together with the deployment descriptors in a ejb–jar file. To deploy un–packed Enterprise Beans, refer to Configuring 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 GenIC 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.

The deployment and installation of Web and J2EE applications

Once the packaging of the application components has been completed as described in the WAR Packaging or 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

Example of deploying and installing an EJB using an ejb−jar file

For this example, it is assumed that a user wants 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`. The user will 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 deployment**
  - Build an ejbjar 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:

```
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 `$JONAS_ROOT/examples/src` directory to build and install all `ejb-jar.jar` files for the examples.
To write a `build.xml` file for ANT, use the `ejbjar` task, which is one of the optional EJB tasks defined in ANT. The `ejbjar` task contains a nested element called `jonas`, which implements the deployment process described above (interposition classes generation and EJB−JAR file update).
Generally, the latest version of the EJB task containing an updated implementation of the `jonas` nested element is provided with JOnAS, in `$JONAS_ROOT/lib/common/ow_jonas_ant.jar`. Click here for the documentation corresponding to this new version of the jonas nested element.
As an example, this code snippet is taken from the `$JONAS_ROOT/examples/src/alarm/build.xml`:

```
<!-- 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>
```

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 the WAR packaging guide.
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.

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 the EAR packaging guide.
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.

Administration Guide

The target audience for this guide is the JOnAS server administrator.

JOnAS provides the following two tools for performing some administration tasks on a running JOnAS Server:

- **`jonas admin`**, a command line tool
- **`JonasAdmin`**, a graphical tool based on the Struts framework and the JMX technology
    - Installing JonasAdmin
    - Using JonasAdmin

These tools also allow administration of several JOnAS Servers. Each JOnAS Server is identified by a name, which is the value of the `-n` option used in the `jonas start` command (the default name is `jonas`).
Begining with JOnAS 4, we also provide the ***J2EE Management EJB component (MEJB)***, as specified by the J2EE Management Specification which defines the J2EE Management Model.

jonas admin

jonas admin is described in the JOnAS Commands chapter.

JonasAdmin

This chapter provides information about installing, configuring, and using the *JonasAdmin* administration console.

JonasAdmin is the new administration tool for JOnAS and replaces the deprecated *Jadmin* tool.

JonasAdmin was developed using the Struts framework; it uses standard technologies such as Java Servlets and JavaServer Pages. JonasAdmin is more ergonomic than Jadmin and provides integrated administration facilities for a Tomcat server running embedded in JOnAS.

Installing JonasAdmin

Designed as a web application, JonasAdmin is packed in a WAR and installed under the `JONAS_ROOT/webapps/autoload/` directory. This WAR can be installed in `JONAS_BASE/webapps/autoload` if a JONAS_BASE variable has been defined in the environment. When installed in the `autoload` directory, JonasAdmin is deployed when starting the JOnAS server, thus the administration console is automatically accessible.

As with any web application, JonasAdmin requires a servlet server to be installed. Additionally, the JOnAS server running JonasAdmin must have the web container service present in the list of services defined in the `jonas.properties` configuration file.

When accessing JonasAdmin, the administrator must provide identification and authentication.
The `jonas-realm.xml` configuration file contains a memory realm definition named `memrlm_1`, which is referenced in both `server.xml` (for Tomcat) and `jetty.xml` (for Jetty) configuration files. The default user name (***jonas***) and password (***jonas***) corresponding to the `admin` role can be modified here.

Using JonasAdmin

Once started, JonasAdmin can administer the JOnAS server in which it is running, as well as other JOnAS servers with which it shares the same registry. Typically, this is used to administer JOnAS servers running without the WEB container service.
Note that the administered JOnAS servers can be running on the same host or on different hosts. Also, if Tomcat is used as the WEB container service implementation, it can be administered using JonasAdmin.

Running JonasAdmin

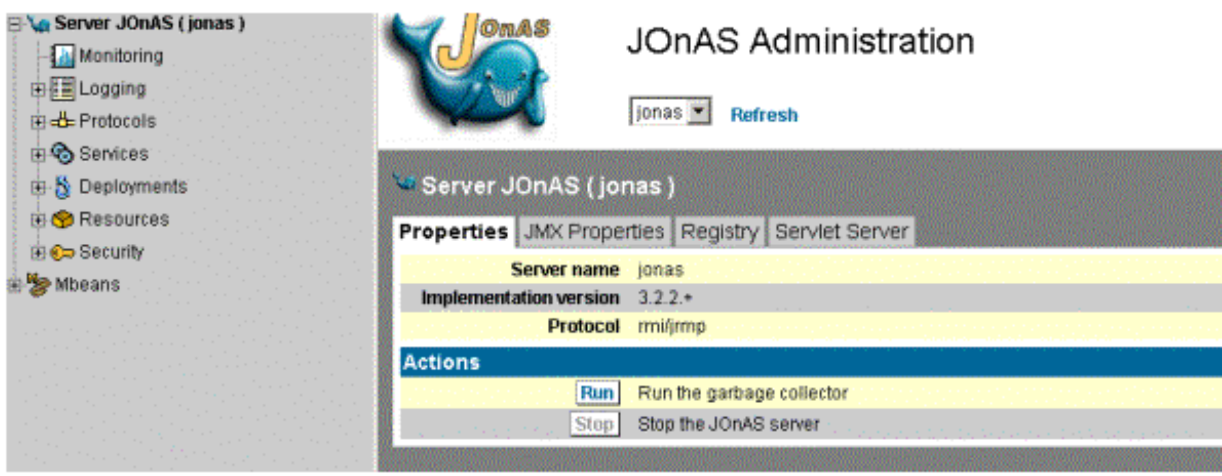
Ensure that the `web` service is listed in the `jonas.services` property in the `jonas.properties` configuration file. If you are not using a jonas–tomcat or jonas–jetty package, depending on the Servlet container being used, the `CATALINA_HOME` or the `JETTY_HOME` environment variable must have been previously set. Note that when running the Servlet container on top of Unix, the `DISPLAY` environment variable must be set in order to use the JOnAS server monitoring feature of JonasAdmin.

Once JOnAS is launched, JonasAdmin must be loaded if it was not installed in the `autoload` directory. The administration console is accessible at the URL: `http://`*`<hostname>`*`:`*`<portnumber>`*`/jonasAdmin/` using any web browser.

<hostname> is the name of the host where the Servlet container is running and *<portnumber>* is the http port number (default is 9000).

After logging in, the left–hand frame in the Welcome page displays the management tree associated with the JOnAS server running the administration application (the following example assumes that its name is *jonas*). If other servers are running that share the same registry with *jonas*, the selector in the upper frame can be used to select the server to administer.

Click on the `Server JOnAS` node to display the following page:



The management tree in this figure allows access to the following main management facilities:

- General information concerning the administered server
- Server monitoring
- Logging management
- Communication protocols management
- Active services presentation and configuration
- Dynamic deployment of application modules
- Resources management
- Security management

The console also allows browsing of MBeans registered in the MBean server that are associated with the currently

managed JOnAS server.

Server management

Displays general information about the administered JOnAS server, including the JMX server and the WEB server, and provides the capability of listing the content of the Registry.

Server monitoring

Presents memory usage, a count of the threads created by JOnAS, and other monitoring information concerning managed services and resources.

Logging management

Allows the administrator to configure the JOnAS Logging system. Additionally, if Tomcat is used as the WEB container service implementation, it allows creation of HTTP access loggers.

Communication protocols management

This management facility relates to the integration of Tomcat management in JonasAdmin.
It currently presents connectors defined in the Tomcat configuration and allows for the creation of new HTTP, HTTPS, or AJP connectors.
Note that the `Protocols` sub–tree is not presented if Jetty is used as the WEB container service implementation.

Active services presentation and configuration

All the active services have a corresponding sub–tree in the `Services` tree.

Managing the various container services consists of presenting information about the components deployed in these containers. New components can be deployed using the dynamic deployment facilities presented in the next section. However, it may be necessary to create a new context for WEB components (WAR package) to be deployed in a Tomcat server before the deployment step, if a customized context is required by the component. This operation is performed using the `New web application` button.

Similarly, the services that allow management of the different types of resources (DataSources, Resource Adapters, Jms and Mail resources) also provide information about the resources being deployed. Additionally, deployed resources (DataSources or MailFactories) can be reconfigured and their new configuration made persistent by using a `Save` button.

The transaction service management allows reconfiguration (possibly persistent) and presents monitoring information about transactions managed by JOnAS.

Dynamic deployment with JonasAdmin

A very useful management operation is the capability of loading stand–alone J2EE components (JAR, WAR, RAR packages) or J2EE applications (EAR packages) using the `Deployments` sub–tree in the JonasAdmin console. The administrator's task is facilitated by the display of the list of deployable modules, the list of deployed modules, and the capability of transferring modules from one list to another. The deployable modules are those installed in directories specific to their type. For example, the deployable JARs are un–deployed JARs installed in `JONAS_BASE/ejbjars/` or in a `JONAS_BASE/ejbjars/`*`autoload`*`/` directory.

Resources management

The `Resources` sub–tree provides the capability of loading or creating new resources managed by the active services. For example, if the JMS service is running, the JMS sub–tree in `Resources` presents the existing JMS destinations (Topics and Queues), and allows the removal of unused destinations and the creation of new JMS destinations.
Adding or removing resources implies reconfiguration of the corresponding service. If this new configuration is saved using the `Save` button, the JOnAS configuration file is updated. As in the JMS service example, the removed topics are deleted from the list assigned to the `jonas.service.jms.topics` property and the newly created topics are added to this list.

Security management

The `Security` sub–tree presents existing security realms and allows the creation of new realms of different types: memory, datasource, and ldap realms.

Note concerning persistent reconfiguration facilities

It is important to note that JOnAS and Tomcat have different approaches to reconfiguration persistency. In JOnAS, every `Save` operation is related to a service or a resource reconfiguration. For example, the administrator can reconfigure a service and a resource, but choose to save only the new resource configuration.
In Tomcat, the `Save` operation is global to all configuration changes that have been performed. For example, if a new HTTP connector is reconfigured and a new context created for a web application, both configuration changes are saved when using the `Save` button.

Management EJB Component

The MEJB component exposes the managed objects within the JOnAS platform as JMX manageable resources. It is packed in an ejb–jar file installed in the *$JONAS_ROOT/ejbjars/autoload* directory, and therefor it is loaded at server start–up.

The MEJB component is registered under the name `java:comp/env/ejb/MEJB`.

The current implementation allows access only to the manageable resources within the current server (the server containing the MEJB's container).

The JOnAS distribution was enriched with a new example called `j2eemanagement`, which shows how the MEJB can be used. You can find details about this management application in *$JONAS_ROOT/j2eemanagement/README* file.

JOnAS Commands Reference Guide

Commands provided with JOnAS are described in this chapter.

jonas
: JOnAS manager

jclient
: Starting a JOnASclient

newbean
: Bean generator

registry
: Java Remote Object Registry

GenIC
: Container classes generator

JmsServer
: JMS Server

RAConfig
: Resource Adapter configuration tool

jonas

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]`
: administrate a JOnAS server

`jonas check`
: check JOnAS environment

`jonas version`
: print JOnAS version

Description

This command replaces the deprecated commands EJBServer, JonasAdmin, and CheckEnv. It provides the capability to start, stop, or administrate JOnAS servers.

The outcome of this program may depend on the directory from which the command is run (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 following two scripts can be reviewed and possibly modified for assistance with problems or for obtaining additional information:

`jonas` for UNIX systems
`jonas.bat` for WINDOWS systems

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
: Administrate 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 shell scripts or bat scripts, for example. Refer to the option list for a description of each. Another way to manage JOnAS is to use the graphical tool JonasAdmin. The functionality is essentially the same for ***JonasAdmin*** as it is for ***jonas admin***.

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

jonas version
: Print the current version of JOnAS.

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 at the end of the process.

−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 will lead 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 exit command.

interactive command	on−line matching command
addbeans	**−a fileName**
env	**−e**
gc	**−gc**

help	**–?**
jndinames	**–j**
listbeans	**–l**
removebeans	**–r fileName**
sync	**–sync**
trace	**–debug topic**
ttimeout	**–tt timeout**
quit	exit interactive mode

Examples

```
jonas check
jonas start -n jonas1
jonas admin -n jonas1 -a bean1.jar
jonas stop -n jonas1
```

jclient

Synopsis

jclient [options] java–class [args]

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.

It is not the J2EE compliant way to run a java client which is to use to package the java client in a J2EE container client (refer to Client Packaging).
The *jclient* command may be deprecated in a future release.

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. (Used for automatic stubs downloading)

Examples

```
jclient package.javaclassname args
```

newbean

Synopsis

newbean

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 only creates 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.

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:

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

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 representing the primary key. Only needed for entity beans. Possible values are:

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

Example

```
newbean
Bean Name
> MyEntity

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

Persistance manager
  C  Container
  B  Bean
> C

Bean location
  R  Remote
  L  Local
> R

Package name
> truc.machin

Jar name
> machin

Primary Key class
0  S  String
  I  Integer
  O  Object
> S

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

registry

Synopsis

registry [<port>]

Description

The *registry* tool creates and starts a remote object registry on the specified port of the current host, based on the *ORB* type defined in the JOnAS configuration (*RMI* or *Jeremie*).

If the port is omitted, the registry is started on port 1099 on *RMI*, or on port 1234 on *Jeremie*.

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.

Options

port

Port number.

Example

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

- `registry &` on Unix, or
- `start registry.bat` on Windows

GenIC

Synopsis

GenIC [Options] <InputFileName>

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, and
4) if the *InputFile* is an ejb–jar file, adds the generated classes in this ejb–jar file.

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
: Invokes directly, in some cases, the method of the java class corresponding to the command. This is useful on Windows in the event of a CreateProcess Exception (this occurs when the command line is too long).

–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 within jeremie, jrmp, iiop, cmi) for which stubs should be generated. Default is `jrmp,jeremie`.

–rmiopts *options*
: Specifies the options to pass to the *rmi* compiler.

–verbose
: Displays additional information about command execution.

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.

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.

JmsServer

Synopsis

JmsServer

Description

Launches the Joram Server (ie the MOM) with its default options.

Options

none

Example

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

- `JmsServer &` on Unix, or
- `start JmsServer` on Windows.

RAConfig

Synopsis

RAConfig [Options] <InputFileName> [<OutputFileName>]

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 the resource adapter.

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

Options: Only capital letters or the full option name must be specified

−? or **−HELP** *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.

−Jndiname *jndiname*
: It is a mandatory option with a 1.0 Resource Adapter. 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
: Turn 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. The default value when not specified 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 an 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 add security information to jonas−ra.xml. This will map the specified principal name to the user on the EIS system. The specified file must be in the following form: `principal = user::password`. When used in conjunction with the −ENcrypt option, then the resulting information will be 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`.

Example

- **`RAConfig -j adapt_1 MyRA.rar`**
  Generates the `jonas–ra.xml` file from the *ra.xml* file.

  After the 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.

Creating a New JOnAS Service

The content of this guide is the following:

1. Target Audience and Rationale
2. Introducing a new Service
3. Advanced Understanding

Target Audience and Rationale

This chapter is intended for advanced JOnAS users who require that some "external" services run along with the JOnAS 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 JONAS Server. The JOnAS pre−defined services are listed in Configuring JOnAS services.

J2EE 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 application 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 JOnAS server.

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.

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 the following section.

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;
    }
}
```

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 (e.g. "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
```

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);
        }
```

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.

Advanced Understanding

Refer to the JOnAS sources for more details about the classes mentioned in this section.

JOnAS built−in services

The existing JOnAS services are the following:

Service name	***Service class***

registry	RegistryServiceImpl
ejb	EJBServiceImpl
web	CatalinaJWebContainerServiceImpl / JettyJWebContainerServiceImpl
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.
(Note that for reasons of compatability with previous versions of JOnAS, if *registry* is unintentionally not set as the first service to launch, JOnAS will automatically launch the *registry* service.)

Note that *dbm*, *jms*, *resource*, and *ejb* depend on *jtm*.
Note that *ear* depends on *ejb* and *web* (that provide the ejb and web containers), thus these services must be launched before the *ear* service.
Note that ear and web depends on ws, thus the ws service must be launched before the *ear* and web service.

It is possible to launch a stand–alone Transaction Manager with only the *registry* and *jtm* services.

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.mdbthreadpoolsize      10

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
```

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`.

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.

JMS User's Guide

1. JMS installation and configuration aspects
2. Writing JMS operations within an application component
3. Some programming rules and restrictions when using JMS within EJB
4. JMS administration
5. Running an EJB performing JMS operations
6. A JMS EJB example

As required by the J2EE v1.4 specification, application components (servlets, JSP pages and enterprise beans) can use JMS for Java messaging. Furthermore, applications can use Message−driven Beans for asynchronous EJB method invocation, as specified by the EJB 2.1 specification.

Starting with the JOnAS 3.1 version, JOnAS supports the Java Message Service Specification 1.1. Previously in JMS 1.0.2, client programming for Point−to−point and Pub/Sub domains was achieved using similar, but separate, class hierarchies. Now, JMS 1.1 offers a domain−independent approach to programming the client application. Thus, the programming model is simpler and 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) 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 and WebSphere MQ, may easily be integrated.

Starting with release 4.1, 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 later JOnAS releases. Also, this method allows deployment of 2.1 MDBs (not possible with the JMS service).

For performing JMS operations, JMS−administered objects will be used by the application components, such as connection factories and destinations. Refer to the JMS Administration section for an explanation of how to create those objects.

JMS installation and configuration aspects

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, the SwiftMQ product and IBM's WebSphere MQ have been used with JOnAS.

Writing JMS operations within an application component

To send (or synchronously receive) JMS messages, the component requires access to JMS−administered objects, i.e. Connection Factories for creating connections to JMS resources and 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.

Refer to the JOnAS example **`jms`** as a supplement to this present reading. This example jms is described here.

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 (`javax.jms.ConnectionFactory`, `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory`) are connection factories for JMS connection objects.
- Java Mail Connection factories (`javax.mail.Session` or `javax.mail.internet.MimePartDataSource`) are connection factories for Java Mail connection objects.

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 only the `javax.jms.ConnectionFactory` be used (rather than `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`). However, the new 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>
```

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 (e.g. "myQueue") being 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>
```

Writing JMS Operations

A typical method performing a message sending JMS operations 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. An EJB method performing synchronous message reception on a queue is illustrated in the following:

```
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.

Be sure to never send and receive a particular message in the same transaction, since the JMS sending operations are actually performed at commit time only.

The previous examples illustrate point–to–point messaging. However, application components can also be developed using the publish/subscribe JMS API, i.e. 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();
    }
```

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 `acknowledgeMode = 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.

Authentication

If your JMS implementation performs user authentication, the following methods can be used on Connection Factories:

- `createConnection(String userName, String password)` on `ConnectionFactory`
- `createQueueConnection(String userName, String password)` on `QueueConnectionFactory`
- `createTopicConnection(String userName, String password)` on `TopicConnectionFactory`

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.

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). 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.

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.

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 perform 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.

JMS service administration

For using the JMS service in the default configuration, it is only necessary to require 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.service.jms.topics    t1,t2    // JOnAS server creates 2 topic destinations (t1,t2)
jonas.service.jms.queues    myQueue  // JOnAS server creates 1 queue destination (myQueue)
```

It is recommended that programmers use *resource references* and *resource environment references* to access the connection factories and destination objects created by JOnAS, as already presented in the "Writing JMS operations within an application component" section.

JMS resource adapter configuration

Starting with JOnAS release 4.1, it is recommended that a JMS resource adapter be deployed instead of using the *jms* service. Refer to the JMS Resource Adapters configuration guide for an explanation.

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
```

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 (e.g. the JORAM MOM or the SwiftMQ MOM).

For launching the MOM, consider the following possibilities:

1. **Launching the MOM automatically in the JOnAS JVM**
   This is done using the default values for the configuration options, i.e. 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 for using the JORAM MOM from a distant host:

- 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.

2. **Launching the MOM in a separate JVM on the same host**
   The JORAM MOM can be launched with its default options using 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
   ```
3. **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, i.e. 16010 (For SwiftMQ, the value of the URL is similar to `smqp://host:4001/timeout=10000`).

   ```
   jonas.service.jms.collocated false
   jonas.service.jms.url        joram://host2:16010
   ```
4. **Launching the MOM on another port number (for JORAM)**
   To change 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 class="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.

5. **Specifying JORAM's persistence mode**
   When automatically starting JORAM, or when starting JORAM with the `JmsServer` command, the default mode is non–persistent. Meaning that in the event of a crash, the non–delivered and non–acknowledged messages are lost.

In order to start a persistent JORAM server, guaranteeing message delivery even in case of failures, the **Transaction** system property should be set to `fr.dyade.aaa.util.ATransaction`.

Note: the MOM may be directly launched by the proprietary command. The command for JORAM is:

```
 java -DTransaction=fr.dyade.aaa.util.NullTransaction
fr.dyade.aaa.agent.AgentServer 0 ./s0
```

This command corresponds to the default options used by the `JmsServer` command.

The server is not persistent when launched with this command. If persistence is required, the `-DTransaction=fr.dyade.aaa.util.NullTransaction` option should be replaced with the `-DTransaction=fr.dyade.aaa.util.ATransaction` option.

To change other MOM configurations (distribution, multi−servers, ...), refer to the JORAM documentation on http://joram.objectweb.org.

... or as a J2EE1.4 adapter

Starting with JOnAS release 4.1, a JMS server can be accessed through a resource adapter which may be deployed.

For deploying such a resource adapter, place the corresponding archive file (`*.rar`) in the JOnAS's `rars/autoload` directory, or 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 the Configuring JMS Resource Adapters section.

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 (i.e. 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.

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");
    }
    ...
}
```

This code has been intentionally cleared from all the elements in which it is not necessary for understanding the JMS logic aspects of the example, e.g. exception management.

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.

The Entity Bean

The example uses the simple entity bean Account for writing data into a database. Refer to the sample eb.

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).

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 User Manual

New JOnAS (Java Open Application Server) element for the current JOnAS version

The `<jonas>` nested element uses the `GenIC`–specific tool to build JOnAS–specific stubs and skeletons and construct a JAR file which may be deployed to the JOnAS Application Server. The build process will always determine if the EJB stubs/skeletons and the EJB–JAR file are up to date, and it will perform 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 will search 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 behaviour 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). Refer to the appropriate documentation for more details.

Parameters:

Attribute	Description	Required
destdir	The base directory into which the generated JAR files will be written. Each JAR file is written in directories which 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		No

	`true` if the intermediate Java source files generated by GenIC must not be deleted. If omitted, it defaults to `false`.	
nocompil	`true` if the generated source files must not be compiled via the java and rmi compilers. If omitted, it defaults to `false`.	No
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 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 args to GenIC.	No
keepgeneric	`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

As noted above, the jonas element supports additional <classpath> nested elements.

Examples

Note : To avoid `java.lang.OutOfMemoryError`, the element `jvmopts` can be used to change the default memory usage.

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}"
         protocols="jrmp,iiop"/>
  <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"
         protocols="${genic.protocols}"/>
  <include name="**/ejb-jar.xml"/>
  <exclude name="**/jonas-ejb-jar.xml"/>
</ejbjar>
```

Login Modules in a Java Client Guide

The content of this guide is the following:

1. Configuring an environment to use login modules with java clients
2. Example of a client

Configuring an environment to use login modules with java clients

The login modules for use by clients are defined in the file `$JONAS_ROOT/conf/jaas.config`. 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.

Example of a client

- First, the CallbackHandler to use is declared. 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, the LoginContext method with the previously defined CallbackHandler and the entry to use from the `JONAS_ROOT/conf/jaas.config` file is called.
  This example uses the dialog callbackhandler.

```
LoginContext loginContext = new LoginContext("jaasclient", handler2);
```

- Finally, the login method on the LoginContext instance is called.

```
loginContext.login();
```

  If there are no exceptions, the authentication is successful.
  Authentication can fail if the supplied password is incorrect.

Web Services with JOnAS

Starting with JOnAS 3.3, Web Services can be used within EJBs and/or servlets/JSPs. This integration conforms to the JSR 921(Web Service for J2EE v1.1) specification.

1. Web Services

A. Some Definitions

WSDL: WSDL (Web Service Description Language v1.1) is 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.
SOAP: SOAP (Simple Object Access Protocol v1.2) is 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.
JAX−RPC: JAX−RPC (Java Api for XML RPC v1.1) is the Java API for XML−based RPC. RPC (Remote Procedure Call) allows 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 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.

B. 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 (refer to www.w3.org/TR/WSDL) as a contract between client and server (also called endpoint). 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 servlet(s)). Web Services can be implemented as Stateless Session Beans or as JAXRPC classes (a simple Java class, no inheritance needed).

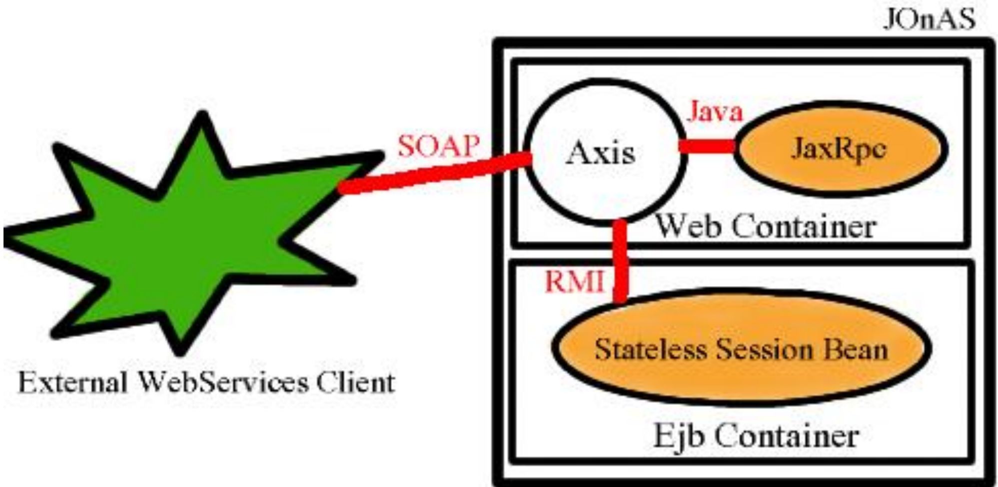


Figure 1. Web Services endpoints deployed within JOnAS (an external client code can access the endpoint via AxisServlet)

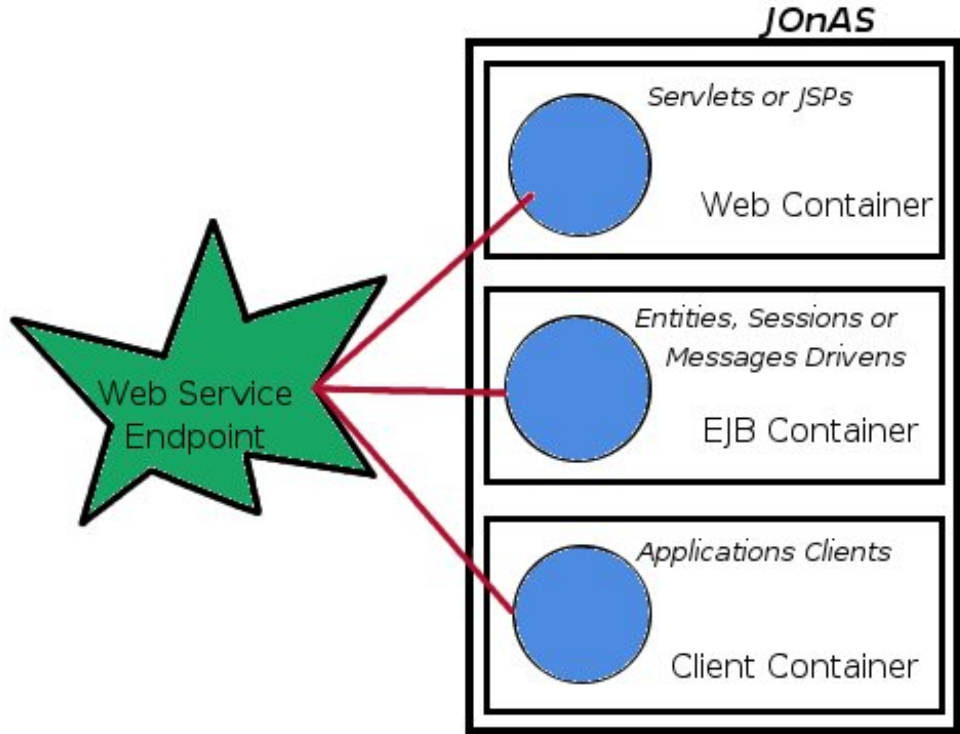


Figure 2. Web Services client deployed within JOnAS (can access external Web Services)
The servlet is used to respond to a client request and dispatch the call to the designated instance of servant (the SSB or JAXRPC class exposed as Web Service). It handles the deserialization of incoming SOAP message to transform SOAP XML into a Java Object, perform the call, and serialize the call result (or the thrown exception) into SOAP XML before sending the response message to the client.

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 webapp (in WEB−INF/) or in an EjbJar (in META−INF/). This Descriptor has its JOnAS−specific Deployment Descriptor (jonas−webservices.xml is optional).

Refer to the webServices sample for example files.

A. JAX−RPC Endpoint

A JAX−RPC endpoint is a simple class running in the servlet container (Tomcat or Jetty). 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 WebApp (the WAR file must contain a "WEB−INF/webservices.xml").

B. Stateless Session Bean Endpoint

An 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 EjbJar, and a "META−INF/webservices.xml" is located in the EjbJar file.

C. Usage

In this Descriptor, the developer describes the components that will be exposed as Web Services' endpoints; these are called the port−component(s). 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 and/or modify the SOAP Message before transmission and/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 actor(s) 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 (i.e., 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 (i.e., 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 uses a port not implemented/linked with a component, the client call will systematically fail).

As for all other Deployment Descriptor, a standard XML Schema is used to constrain the XML.

D. Simple Example (expose a JAX−RPC Endpoint) of webservices.xml

```
<?xml version="1.0"?>

<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 contain 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 -->
     <!-- Should not contain 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>
```

E. The optionnal jonas−webservices.xml

The jonas−webservices.xml file is collocated with the webservices.xml. It is an optional Deployment Descriptor (only required in some case). Its role is to link a webservices.xml to the WebApp in charge of the SOAP request dispatching. In fact, it is only needed for an EjbJar (the only one that depends on another servlet to be accessible with HTTP/HTTPS) that does not use the Default Naming Convention used to retrieve a webapp name from the EjbJar name.

Convention: <ejbjar−name>.jar will have an <ejbjar−name>.war WebApp.

Example:

```
<?xml version="1.0"?>
<jonas-webservices 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_j2ee_web_services_4_0.xsd">
    <!-- the name of the webapp in the EAR              -->
    <!-- (it is the same as the one in application.xml) -->
    <war>dispatchingWebApp.war</war>
</jonas-webservices>
```

F. 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 of the dispatching WebApp.
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 only be determined when only one host is set for the web container.
− The port can only be determined when only one connector is used by the web container.

3. The client of a Web Service

An EJB or a servlet that wants to use a Web Service (as client) must declare a dependency on the Web Service with a service−ref element (same principle as for all *−ref elements).

A. The service−ref element

The service−ref element declares reference to a Web Service used by a J2EE component in the web, EJB and Client application Deployment Descriptor.

The component uses a logical name called a service–ref–name to lookup 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, ejb–jar.xml or application–client.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 -->
  <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 thus
      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
       of a namespaceURI and a localpart. It must be defined if more than 1
```

```
        service is declared in the WSDL. -->
      <service-qname xmlns:ns="http://beans.ws.objectweb.org">
        ns:MyWSDLService
    </service-qname>

    <!-- Declares a client dependency on the container for 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>

      <!-- Defines a link to a port component declared in another unit
           of the application -->
      <!-- link is only used when an application module wants to access a  -->
      <!-- web service colocated in the same application unit             -->

<port-component-link>ejb_module.jar#PortComponentName</port-component-link>
    </port-component-ref>

    <!--A list of Handlers 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-params (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 QNames specifying the SOAP Headers the handler
            will work on. Namespace and locapart values must be found
            inside the WSDL. -->
       <soap-header xmlns:ns="http://ws.objectweb.org">
            ns:MyWSDLHeader
       </soap-header>

       <!-- A list of SOAP actor definitions 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 elements that 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>
```

B. 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,
application-client.xml).
           used as a key to associate a service-ref to its correspondent
           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>
```

4. WsGen

WsGen is a new JOnAS tool that works in the same way as GenIC. It takes archive files (EJB–JAR, WAR, JAR client, EAR) and generates all the necessay components related to web services:

- 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 WebApp for each EJB−JAR exposing web services.
- Generates and compiles client side artifacts (Services and Port Bindings implementations classes).

For example, to provide an EJB−exposing method as a web service, a developer creates a `webservices.xml` file packaged in EjbJar's META−INF directory. WsGen automatically creates a configured webapp (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 ejbjars, web applications, or clients), WsGen automatically generates a Stub from WSDL (if a generated service interface name is provided).

Usage

WsGen is used typically from an ant build file. Simply add this taskdef under the ejbjar 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 ejbjar/webapp/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 ejbjar ant task of JOnAS; you must ensure that they have been placed inside the srcdir given to the ejbjar task (otherwise, the ejbjar task cannot find them and will produce an error).

This task is a directory−based task and, as such, forms an implicit Fileset. 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 (EjbJar, War, Client, Ear) is located	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

Wsgen is also usable from the command line with WsGen script (available on *nix and Windows).

6. Limitations

- The jaxrpx−mapping−file is used only to retrieve XML namespace to java package information mapping. All other information is not used at this time (Axis limitation).
- service−endpoint−interface in port−component and port−component−ref is read, but not used.

last update : 03 June 2004

Working with Management Beans

The content of this guide is the following:

1. Target Audience and Rationale
2. About JOnAS MBeans and their use in JonaSAdmin
3. Using JOnAS MBeans in a Management Application
4. Registering User MBeans

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) compliant to JMX Specification. Moreover, JOnAS implements JSR 77 specification, which defines the management model for J2EE platforms.

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 the different JOnAS services, but also by the 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 to the JSR 160 specification. See more information about connectors here.

JonasAdmin application implements the management functions listed above using the different MBeans registered in the MBeanServer of the JOnAS instance 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 operations. In the future, JonasAdmin will probably be extended to allow setting attributes and invoking operations.

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.

Local MBean Server

When the caller is located in the same JVM as the MBean Server, it can use `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);
}
```

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:1
JMXConnector connector = JMXConnectorFactory.newJMXConnector(connURL, null);
connector.connect(null);
MBeanServerConnection conn = connector.getMBeanServerConnection();
return conn;
```

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.jar` 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`. JOnAS distribution also contains an example using the MEJB in a J2EE application, the `j2eemanagement` sample.

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 a m–let text file. Refer to JMX implementation documentation for details concerning this file. In addition, the howto document 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, it can be installed in the `JONAS_ROOT/conf` directory.

Howto: JOnAS Versions Migration Guide

The content of this guide is the following:

- JOnAS 4.1 to JOnAS 4.3.2
- JOnAS 3.3.x to JOnAS 4.1
- JOnAS 3.1 to JOnAS 3.1.4
- JOnAS 3.0 to JOnAS 3.1
- JOnAS 2.6.4 to JOnAS 3.0
    1. Application with entity beans CMP 1.1
- JOnAS 2.6 to JOnAS 2.6.1
    1. Installation procedure
    2. EJB container creation
- JOnAS 2.5 to JOnAS 2.6
    1. Use of Make discontinued for building
    2. Building with Ant 1.5
    3. New jonas command
    4. EJB container creation
    5. Tomcat 4.0.x Support
    6. Tomcat Service and Web Container Service
- JOnAS 2.4.4 to JOnAS 2.5
    1. Tomcat 4.0.x Support
    2. trace.properties
- JOnAS 2.4.3 to JOnAS 2.4.4
- JOnAS 2.3 to JOnAS 2.4
    1. Introduction
    2. Upgrading the jonas.properties file
    3. Upgrading the Jonathan configuration file

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 to report your customizations in the new JOnAS 4.3.2 configuration files, especially for the ones mentioned below.

Configuration changes

The most visible configuration changes are the following:

1. JORAM's rar usage is set by default 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. 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.
- The configuration file of JacORB is the `$JONAS_BASE/conf/jacorb.properties` file.

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(e.g. 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.

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.

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, e.g. 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, e.g. 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 war), the package name of the Realm class has changed, e.g. 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***: many changes
    - some properties for web services
    - some package names have changed (e.g. 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.

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 the Configuring JMS Service and Configuring JMS Resource Adapters sections of the Configuration Guide.

Deploying Resource Adapters

The RAConfig Resource Adapter configuration tool 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.

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, i.e. 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 the JOnAS Configuration Guide for details about Communication Protocols 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 (order 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 the JOnAS Configuration Guide for details about Security configuration.

The ***preferred steps*** for migrating from JOnAS 3.1 are the following:

1. Create a new JOnAS_BASE (e.g. 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, 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.

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
```

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
```

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"/>
```

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>
```

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 within 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.

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`.

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/con.

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` are searched for in $JONAS_BASE/(ejbjars|apps|webapps), if not, under

$JONAS_ROOT/(ejbjars|apps|webapps).

JOnAS 2.6.4 to JOnAS 3.0

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.

JOnAS 2.6 to JOnAS 2.6.1

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 (i.e. from RMI to JEREMIE) is switched, or if the web environment (i.e. switching from CATALINA to JETTY) changes.

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).

JOnAS 2.5 to JOnAS 2.6

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.

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.

New jonas command

A new **`jonas`** command is delivered that provides the capability to:

- start a JOnASserver,
- stop a JOnASserver,
- administrate a JOnASserver,
- check the JOnASenvironment,
- print the JOnASversion.

The `EJBServer`, `JonasAdmin` and `CheckEnv` commands are now deprecated.
Refer to the JOnAS Commands Reference Guide in the JOnAS documentation for details.

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,
- or an argument of the `–a` option of the `jonas admin` command.

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.

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 the section "Integrating Tomcat and JOnAS" for details.

JOnAS 2.4.4 to JOnAS 2.5

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 `jonas.properties` file to `org.objectweb.jonas.tomcat.EmbeddedTomcatImpl40` must be set. Refer to the How to use Tomcat with JOnAS document for more details.

trace.properties

Starting with JOnAS 2.5, the Objectweb Monolog logging API is used for JOnAS traces. The JOnAS traces configuration file `JONAS_ROOT/conf/trace.properties` has changed. Refer to the Configuration Guide in the JOnAS documentation for details.

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 2.4.3	**from JOnAS 2.4.4**
jonas services	– jonas.services *jmx,security,jtm,dbm,resource,jms,ejb*	– jonas.services *registry,tomcat,jmx,security,jtm,dbm,resource,jms,ejb*
Registry service configuration	–	– jonas.service.registry.class *org.objectweb.jonas.registry.RegistryServiceImpl* – *jonas.service.registry.mode automatic*
Tomcat service configuration	–	– jonas.service.tomcat.class *org.objectweb.jonas.tomcat.EmbeddedTomcatImpl33* – *jonas.service.tomcat.args*

JOnAS 2.3 to JOnAS 2.4

Introduction

This chapter 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 on JOnAS services in the JOnAS documentation), 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, since JOnAS has embedded a new version of Jonathan.

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 this order listed (1, 2, 3), 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:

	before JOnAS 2.4	**from JOnAS 2.4**
jonas services (new)	–	– jonas.services *jmx,security,jtm,dbm,resource,jms,ejb*
JMX service configuration	–	– jonas.service.jmx.class *org.objectweb.jonas.jmx.JmxServiceImpl*
JOnAS EJB service configuration (beans to be loaded)	– jonas.beans.descriptors ...	– jonas.service.ejb.class *org.objectweb.jonas.container.EJBServiceImpl* – jonas.service.ejb.descriptors ...
JOnAS DBM service configuration	– jonas.datasources ...	– jonas.service.dbm.class *org.objectweb.jonas.dbm.DataBaseServiceImpl* – jonas.service.dbm.datasources ...
JOnAS JTM service	– jonas.tm.remote *false* – jonas.tm.timeout *60*	– jonas.service.jtm.class *org.objectweb.jonas.jtm.TransactionServiceImpl*

configuration		– jonas.service.jtm.remote *false* – jonas.service.jtm.timeout *60*
JOnAS SECURITY service configuration	–	– jonas.service.security.class *org.objectweb.jonas.security.JonasSecurityServiceImpl*
JOnAS JMS service configuration	– jonas.jms.mom *org.objectweb.jonas_jms.JmsAdminForJoram* – jonas.jms.collocated *true* – jonas.jms.url *joram://localhost:16010* – jonas.jms.threadpoolsize *10* – jonas.jms.topics *sampleTopic* – jonas.jms.queues ...	– 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.ejb.mdbthreadpoolsize *10* – jonas.service.jms.topics *sampleTopic* – jonas.service.jms.queues ...
JOnAS RESOURCE service configuration (Resource Adapters to be installed)	–	– 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).

Upgrading the Jonathan configuration file

In the new version of Jonathan, the jonathan.prop has been replaced by jonathan.xml.

Howto: Installing JOnAS from scratch

This guide provides instructions for installing JOnAS from scratch on Unix−compatible systems.

The content is organized into the following steps

1. JDK 1.4 installation
2. Ant 1.6 installation
3. Tomcat 5.0.x installation
4. Jetty 5.0.x installation
5. JOnAS installation
6. Setup

JDK 1.4 installation

Download the binary version of JDK 1.4 from the java Sun web site into the appropriate directory. Launch the executable file:

./j2sdk−1_<version number>−<system>.bin for Unix
./j2sdk−1_<version number>−<system>.sh for Linux
j2sdk−1_<version number>−windows−i586.exe for Windows

Set the JAVA_HOME environment variable and update the path:

export JAVA_HOME=<Installation Directory>
PATH=$JAVA_HOME/bin:$PATH (on Windows: PATH=%JAVA_HOME%/bin;%PATH%)

Ant 1.6 installation

Download the binary version of Ant 1.6 from the Ant Apache web site. Untar or Unzip it into the appropriate directory:

tar −jxvf apache−ant−1.6.x−bin.tar.bz2
(*or unzip apache−ant−1.6.x−bin.zip*)

Set the ANT_HOME environment variable and update the path:

export ANT_HOME=<Installation Directory>
PATH=$PATH;$ANT_HOME/bin (on Windows: PATH=%ANT_HOME%/bin;%PATH%)

Download bcel−5.x.tar.gz from the Jakarta web site and install bcel−5.x.jar in the directory $ANT_HOME/lib.

Tomcat 5.0.x installation

Download the binary version of Tomcat 5.0.x from the Jakarta Tomcat web site. Untar it into the appropriate directory:

```
tar -xvf jakarta-tomcat-5.0.x.tar.gz
```

Set the CATALINA_HOME environment variable:

```
export CATALINA_HOME=<Installation Directory>
```

JONAS_BASE directory can be used as CATALINA_BASE: `export CATALINA_BASE=$JONAS_BASE`
Configuration information for the Realm and users is provided at the Setup process.

Jetty 5.0.x installation

Download the binary version of Jetty 5.0.x from the Jetty web site. Untar it into the appropriate directory:

```
tar -xvf jetty-5.0.x-all.tar.gz
```

Set the JETTY_HOME environment variable:

```
export JETTY_HOME=<Installation Directory>
```

JOnAS installation

Download the binary version of JOnAS from the ObjectWeb web site.
Choose a location for the JOnAS installation.
Be aware that if you have already installed a previous version of JOnAS in this location, the new installation will overwrite existing files, thus customized configuration files may be lost. Therefore, it is prudent to save these files before starting the installation process.
The installation process consists of untaring the downloaded file.
Change to the directory in which JOnAS is to be installed and untar this file, using the `tar -zxvf jonas.tgz` command.
After installing the JOnAS product, set the following environment variable:

```
export JONAS_ROOT = <Installation Directory>
PATH = $JONAS_ROOT/bin/unix:$PATH
```

Do an `ant install` in the `JONAS_ROOT` directory to unpack the jar files necessary to build `jonas.jar` and `client.jar`.

Setup

Before using JOnAS, the following setup activities must be completed:

- Based on the data source being used, create a file `<data source>.properties` (templates are located in the directory `$JONAS_ROOT/conf`). Then, add the data source file name (without the extension `.properties`) to the `jonas.properties` file:

  ```
  jonas.service.dbm.datasource      <data source>
  ```

  Add the JDBC driver in `$JONAS_ROOT/lib/ext` or in `$JONAS_BASE/lib/ext` directory.
- JONAS_BASE directory can be used as CATALINA_BASE: `export CATALINA_BASE=$JONAS_BASE` so it will use the JOnAS realms.
- If required, configure the Mail service (for PetStore or the example mailsb for instance). Two types of files that can be adapted to fit your installation are located in the directory `$JONAS_ROOT/conf`: `MailSession1.properties` and `MailMimePartDS1.properties`. Then, in the `jonas.properties` file, define the `jonas.service.mail.factories` property:

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

Compile the examples as follows:
`ant install` in the directory `$JONAS_ROOT/examples`

JOnAS installation is now complete. For more information, refer to the JOnAS Documentation.

Howto: Installing the packaging JOnAS with a web container
(JOnAS/Tomcat or JOnAS/Jetty) from scratch

This guide provides instructions for installing JOnAS (with a web container already included) from scratch.

The content is organized into the following steps:

1. JDK 1.4 installation
2. ANT 1.6 installation
3. JOnAS/ Web Container installation
4. Setup
5. Starting JOnAS and running examples

JDK 1.4 installation

Download the binary version of a JDK 1.4 from the java Sun web site into the appropriate directory. Launch the executable file:

`./j2sdk-1_<version number>-<system>.bin` for Unix
`./j2sdk-1_<version number>-<system>.sh` for Linux
`j2sdk-1_<version number>-windows-i586.exe` for Windows

Set the JAVA_HOME environment variable and update the path:

```
export JAVA_HOME=<Installation Directory>
PATH=$JAVA_HOME/bin:$PATH (on Windows : PATH=%JAVA_HOME%/bin;%PATH%)
```

ANT 1.6 installation

Download the binary version of Ant 1.6 from the Ant Apache web site. Untar or Unzip it into the appropriate directory:

`tar -jxvf apache-ant-1.6.1-bin.tar.bz2`
(*or* *`unzip apache-ant-1.6.1-bin.zip`*)

Set the ANT_HOME environment variable and update the path:

```
export ANT_HOME=<Installation Directory>
PATH=$PATH;$ANT_HOME/bin (on Windows : PATH=%ANT_HOME%/bin;%PATH%)
```

Download `bcel-5.1.tar.gz` from the Jakarta web site and install bcel–5.1.jar in the directory `$ANT_HOME/lib`.

JOnAS/Web Container installation

Download the binary version of JOnAS with Tomcat or Jetty from the ObjectWeb forge web site.
Choose a location for the JOnAS installation.
Be aware that if you have already installed a previous version of JOnAS in this location, the new installation will overwrite the existing files, thus customized configuration files may be lost. Therefore, it is prudent to save these files before starting the installation process.
The installation process consists of untaring the downloaded file.
Change to the directory in which JOnAS will be installed and untar this file, using
the `tar -zxvf jonas.tgz` command. Note that this file can be opened with winzip on Windows.
After installing the JOnAS product, set the following environment variable:

```
export JONAS_ROOT = <Installation Directory>
PATH = $JONAS_ROOT/bin/unix:$PATH (on Windows: PATH=%JONAS_ROOT%/bin/nt;%PATH%)
```

Setup

Before using JOnAS, complete the following setup activities:

- If a `CATALINA_HOME` or `CATALINA_BASE` or `JETTY_HOME` environment variable has already been set, it should be unset. JOnAS will set these variables, without requiring any modifications.
- Based on the data source being used, create a file `<data source>.properties` (templates are located in the directory `$JONAS_ROOT/conf`). Then add the data source file name (without the extension `.properties`) to the `jonas.properties` file:
  ```
  jonas.service.dbm.datasource     <data source>
  ```
  Add the JDBC driver in `$JONAS_ROOT/lib/ext` or in `$JONAS_BASE/lib/ext` directory.
- If required, configure the Mail service (for PetStore or the example mailsb, for example). JOnAS provides two types of mail factories: `javax.mail.Session` and `javax.mail.internet.MimePartDataSource`. Two types of files that can be adapted to fit your installation are located in the directory `$JONAS_ROOT/conf`: `MailSession1.properties` and `MailMimePartDS1.properties`. Then, in the `jonas.properties` file, define the `jonas.service.mail.factories` property:
  ```
  jonas.service.mail.factories     MailSession1,MailMimePartDS1
  ```

Starting JOnAS and running some examples

If the **Setup step** has not been completed as described, JOnAS may not **work**.
Use the command **`jonas check`** to verify that the environment is correct.

If the environment is **correct**, JOnAS is **ready** to use.

Do a **`jonas start`**, then use a browser to go http://localhost:9000/. (*Modify this url with the appropriate hostname*.)
From the root context in which JOnAS was deployed, you can execute the earsample, access the JOnAS administration application, as well as perform other functions.
For more information, consult the JOnAS Documentation.
Follow the getting started guide to run the examples provided with JOnAS.

Howto: How to compile JOnAS

The content of this guide is the following:

1. Target Audience and Rationale
2. Getting the JOnAS Source
3. Recompiling JOnAS from the Source
4. Recompiling the package JOnAS/Jetty/Axis from the Source
5. Recompiling the package JOnAS/Tomcat/Axis from the Source

Target Audience and Rationale

The target audience for this chapter is the JOnAS user who wants to build a JOnAS version from the source code obtained from CVS.

Getting the JOnAS Source

CVS (Concurrent Version System) provides network−transparent source control for groups of developers. It runs on most UNIX systems and on Windows NT systems. Refer to http://www.cyclic.com for more information

CVS provides many *read−only cvs* commands, such as `cvs status` or `cvs diff`. However, because it is read only, an individual user cannot commit changes. To start working with CVS on JOnAS, make a `checkout` of the `jonas` module, using the following command:

```
cvs -d :pserver:anonymous@cvs.jonas.forge.objectweb.org:/JOnAS login
(hit enter key when prompted for password)
cvs -d :pserver:anonymous@cvs.jonas.forge.objectweb.org:/JOnAS co jonas
```

The `CVSROOT` variable can be set, instead of using the ***−d*** option.

Recompiling JOnAS from the Source

1. Download *Ant* from the Ant project site.

   The `build.xml` file used for building JOnAS is located in the `objectweb/jonas` directory.
2. The JDK and ANT must have been successfully configured. (`JAVA_HOME, ANT_HOME, PATH` environment variables)
3. Set the following environment variables with the appropriate value:
   - JONAS_ROOT: directory where jonas will be installed
   - CATALINA_HOME: optional (mandatory for using the web container Tomcat)
   - JETTY_HOME: optional (mandatory for using the web container Jetty)
4. Perform the following:

cd $OBJECTWEB_HOME/jonas and choose a target:

- **ant install** to install a JOnAS binary distribution version in the `JONAS_ROOT` directory.
- **ant all** to build JOnAS and keep the files in the `output` directory.
- **ant archive** to build a JOnAS archive (.tgz extension) that contains a binary version of JOnAS. The archive is built in the `HOME` directory.

Recompiling the package JOnAS/Jetty/Axis from the Source

1. Download *Ant* from the Ant project site.

   The `build.xml` file used for building JOnAS is located in the `objectweb/jonas` directory.
2. Place the bcel.jar in the `ANT_HOME/lib` directory. It is available for downloading from the Jakarta web site.
3. The JDK and ANT must have been successfully configured. (`JAVA_HOME, ANT_HOME, PATH` environment variables)
4. Set the following environment variables with the appropriate value:
   - JONAS_ROOT: directory where jonas will be installed
   - JETTY_HOME: mandatory
5. Perform the following:

   cd $OBJECTWEB_HOME/jonas and choose a target:

   - **ant install_jetty** to install a JOnAS/Jetty/Axis binary distribution version into your `JONAS_ROOT` directory.
   - **ant all_jetty** to build JOnAS/Jetty/Axis and keep the files in the `output` directory.
   - **ant archive_jetty** to build a JOnAS/Jetty/Axis archive (.tgz extension) that contains a binary version of JOnAS/Jetty/Axis. The archive is built into the `HOME` directory.

Recompiling the package JOnAS/Tomcat/Axis from the Source

1. Download *Ant* from the Ant project site.

   The `build.xml` file used for building JOnAS is located in the `objectweb/jonas` directory.
2. Place the bcel.jar in the `ANT_HOME/lib` directory. It is available for downloading from the Jakarta web site.
3. The JDK and ANT must have been successfully configured. (`JAVA_HOME, ANT_HOME, PATH` environment variables)
4. Set the following environment variables with the appropriate value:
   - JONAS_ROOT: directory where jonas will be installed
   - CATALINA_HOME: mandatory
5. Perform the following:

   cd $OBJECTWEB_HOME/jonas and choose a target:

- **`ant install_tomcat`** to install a JOnAS/Tomcat/Axis binary distribution version into the `JONAS_ROOT` directory.
- **`ant all_tomcat`** to build JOnAS/Tomcat/Axis and keep the files in the `output` directory.
- **`ant archive_tomcat`** to build a JOnAs/Tomcat/Axis archive (.tgz extension) that contains a binary version of JOnAS/Tomcat/Axis. The archive is built in the `HOME` directory.

Howto: Clustering with JOnAS

This guide describes how to configure Apache, Tomcat, and JOnAS to install a cluster.

This configuration uses the Apache/Tomcat plug–in mod_jk. This plug–in allows use of the Apache HTTP server in front of one or several Tomcat JSP/Servlet engines, and provides the capability of forwarding some of the HTTP requests (typically those concerning the dynamic pages, i.e. JSP and Servlet requests) to Tomcat instances.

It also uses the In–Memory–Session–Replication technique based on the group communication protocol JavaGroups to provide failover at servlet/JSP level.

For the load balancing at EJB level, a clustered JNDI called cmi is used.

This document describes one architecture with all the clustering functionalities available in JOnAS, the configuration of architectures integrating one of those functionalities, and other possible configurations.

The content of this guide is the following:

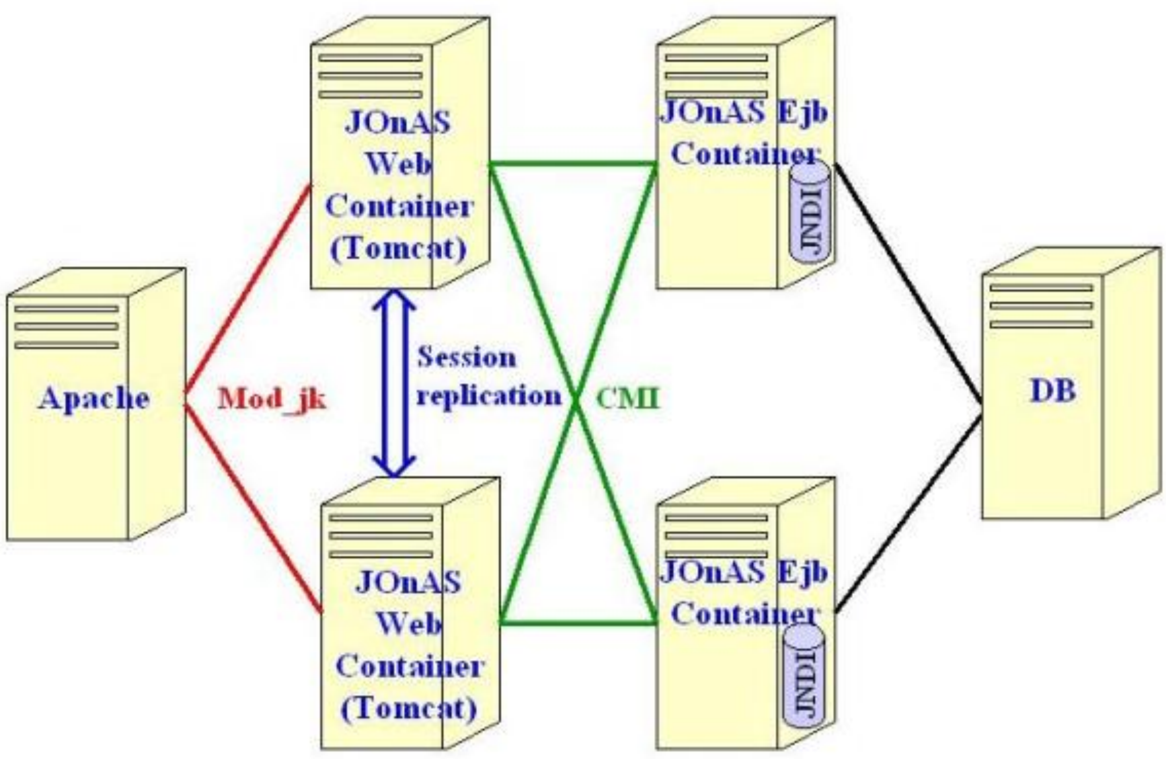
- Architecture
- Products Installation
    - Installing Apache
    - Installing the package JOnAS/Tomcat
- Load Balancing at web level with mod_JK
    - Configuring the JK Module
    - Configuring JOnAS
    - Running a Web Application
- Session Replication at web level
    - Running your application
- Load Balancing at EJB level
    - CMI Principles
    - CMI Configuration
- Preview of a coming version
- Used symbols
- References

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_jk 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 versions assumed here are: Apache 2.0, JOnAS 3.1.x/Tomcat 4.1.x package.

The architecture presented in this document is shown in the following illustration:



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, i.e. 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.

Products Installation

This chapter provides information about installing Apache and JOnAS / Tomcat. The versions assumed here are: Apache 2.0 and the package JOnAS 3.1.x /Tomcat 4.1.x.

Installing Apache

1. Download Apache HTTP server source code from the Apache site.
2. Extract the source code.
   ```
   gunzip httpd-2_0_XX.tar.gz
   tar xvf httpd-2_0_XX.tar
   ```
3. Compile and Install.
   ```
   ./configure
   make
   make install
   ```

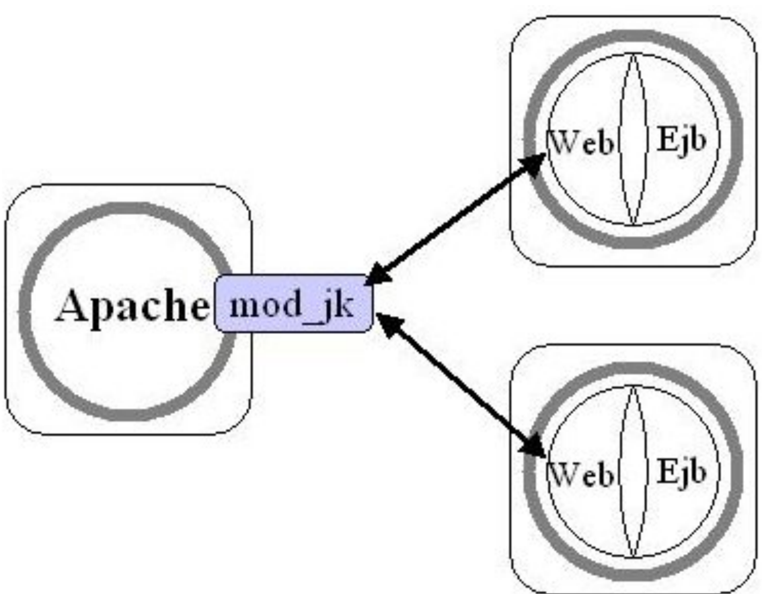
A binary version is also available for installation at the Apache site.

Installing the package JOnAS / Tomcat

Refer to Installing JOnAS with a web container from scratch.

Load balancing at web level with mod_jk

This chapter describes how to configure Apache, Tomcat, and JOnAS to run the architecture shown in the following illustration:



Configuring the JK Module

JK module principles

Mod_jk is a plug–in that handles the communication between Apache and Tomcat.
Mod_jk uses the concept of 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 a file called worker.properties.

Note: this module can also be used for site partitioning.

install Mod_jk

The easiest way to obtain this plug–in is to download the binary from the Tomcat Site and place it in the directory libexec (for unix) or modules (for windows or Mandrake) of the Apache installation directory.

Configure Apache

- httpd.conf
  Create a file tomcat_jk.conf, which must be included in $APACHE_HOME/conf/httpd.conf.
  This file should load the module mod_jk:

```
LoadModule jk_module modules/mod_jk.so (for windows)
LoadModule jk_module libexec/mod_jk.so (for Unix)
AddModule mod_jk.c
```

And configure mod_jk:

```
# Location of the worker file
JkWorkersFile "/etc/httpd/conf/jk/workers.properties"
# Location of the log file
JkLogFile "/etc/httpd/jk/logs/mod_jk.log"
# Log level : debug, info, error or emerg
JkLogLevel emerg
# Assign specific URL to Tomcat workers
JkMount /admin loadbalancer
JkMount /admin/* loadbalancer
JkMount /examples loadbalancer
JkMount /examples/* loadbalancer
```

- worker.properties
  This file should contain the list of workers first:
  `worker.list=<a comma separated list of worker names>`
  then the properties of each worker:
  `worker.<worker name>.<property>=<property value>`

  The following is an example of a worker.properties file:

```
# List the workers name
worker.list=worker1,worker2,loadbalancer
# ----------------
# First worker
# ----------------
worker.worker1.port=8009
worker.worker1.host=server1
worker.worker1.type=ajp13
# Load balance factor
worker.worker1.lbfactor=1
# ----------------
# Second worker
# ----------------
worker.worker2.port=8009
worker.worker2.host=server2
worker.worker2.type=ajp13
worker.worker2.lbfactor=1
# ----------------------
# Load Balancer worker
# ----------------------
worker.loadbalancer.type=lb
worker.loadbalancer.balanced_workers=worker1,worker2
```

Configure Tomcat

To configure Tomcat, perform the following configuration steps for each Tomcat server.

1. Configure Tomcat for the connector AJP13. In the file conf/server.xml of the JOnAS installation directory, add (if not already there):

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector className="org.apache.ajp.tomcat4.Ajp13Connector"
    port="8009" minProcessors="5" maxProcessors="75"
    acceptCount="10" debug="20"/>
```

2. Define the jvmRoute.
   In the file conf/server.xml of the JOnAS installation directory, 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">`

   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.

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>
```

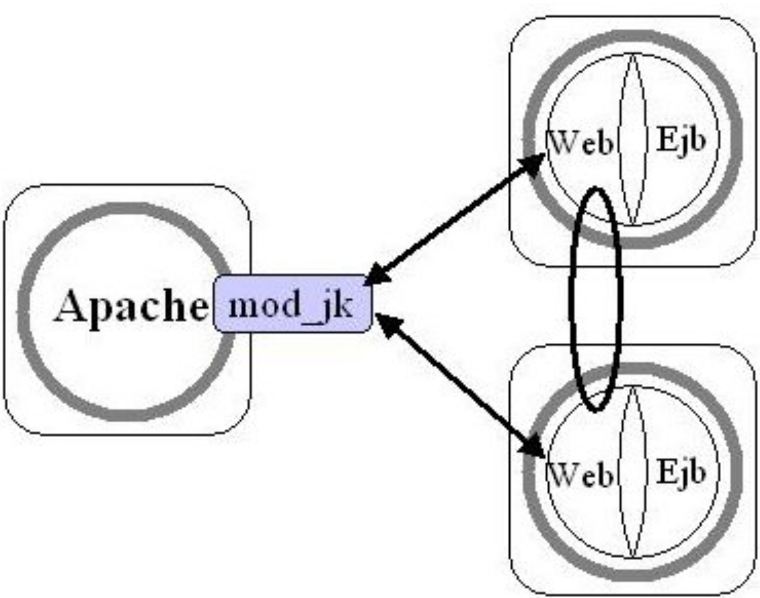
Running a Web Application

The web application is now ready to run:

1. Start the jonas servers: `jonas start`.
2. Restart Apache: `/usr/local/apache2/bin/apachectl restart`.
3. Use a browser to access the welcome page, usually index.html.

Session Replication at web level

The intent of this chapter is to configure Apache, Tomcat, and JOnAS to run the following architecture:



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 in−memory−session−replication. It uses a group communication protocol written entirely in Java, called JavaGroups. JavaGroups is a communication protocol based on the concept of virtual synchrony and probabilistic broadcasting.

The follow describes the steps for achieving Session replication with JOnAS.

- The mod_jk is used to illustrate the Session Replication. Therefore, first perform the configuration steps presented in the chapter Load Balancing at Web level with mod_jk.
- On the JOnAS servers, open the `<JONAS_BASE>/conf/server.xml` file and configure the `<context>` as described:

```
<Context path="/replication−example" docBase="replication−example" debug="99"
                  reloadable="true" crossContext="true"
                  className="org.objectweb.jonas.web.catalina41.JOnASStandardContext">
          <Logger className="org.apache.catalina.logger.FileLogger"
                    prefix="localhost_replication_log." suffix=".txt"
                    timestamp="true"/>
          <Valve className="org.apache.catalina.session.ReplicationValve"
                    filter=".*\.gif;.*\.jpg;.*\.jpeg;.*\.js" debug="0"/>
          <Manager className="org.apache.catalina.session.InMemoryReplicationManager"
                      debug="10"
                      printToScreen="true"
```

```
                    saveOnRestart="false"
                    maxActiveSessions="−1"
                    minIdleSwap="−1"
                    maxIdleSwap="−1"
                    maxIdleBackup="−1"
                    pathname="null"
                    printSessionInfo="true"
                    checkInterval="10"
                    expireSessionsOnShutdown="false"
                    serviceclass="org.apache.catalina.cluster.mcast.McastService"
                    mcastAddr="237.0.0.1"
                    mcastPort="45566"
                    mcastFrequency="500"
                    mcastDropTime="5000"
                    tcpListenAddress="auto"
                    tcpListenPort="4001"
                    tcpSelectorTimeout="100"
                    tcpThreadCount="2"
                    useDirtyFlag="true">
          </Manager>
</Context>
```

Note: The multicast address and port must be identically configured for all JOnAS/Tomcat instances.

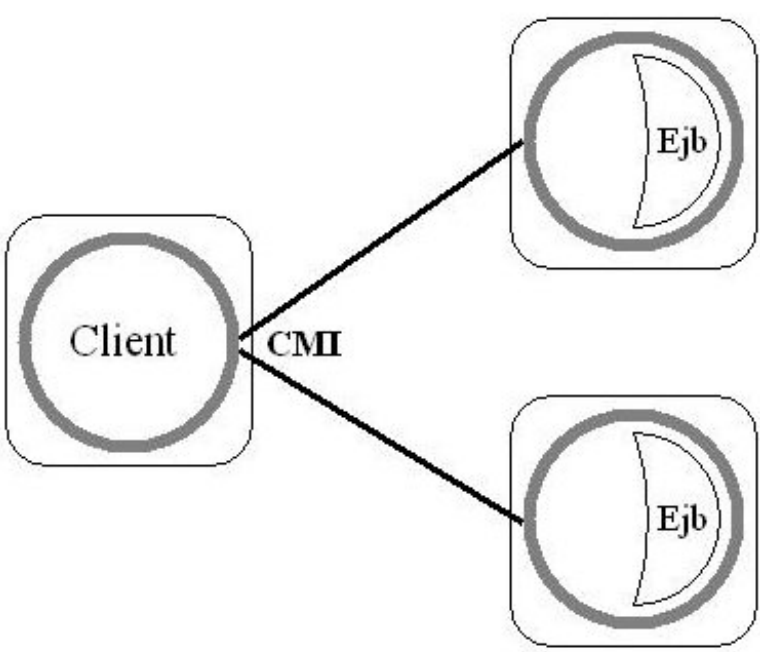
Running your Web Application

The web application is now ready to run in the cluster:

1. Start the JOnAS servers : `jonas start`.
2. Restart Apache : `/usr/local/apache2/bin/apachectl restart`.
3. Use a browser to access the welcome page, usually index.html.

Load Balancing at EJB level

The intent of this chapter is to configure JOnAS to run the following architecture:



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.

CMI Configuration

In the case of EJB level clustering (CMI), the client may be either a fat Java client (e.g. a Swing application), or a Web application (i.e. 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

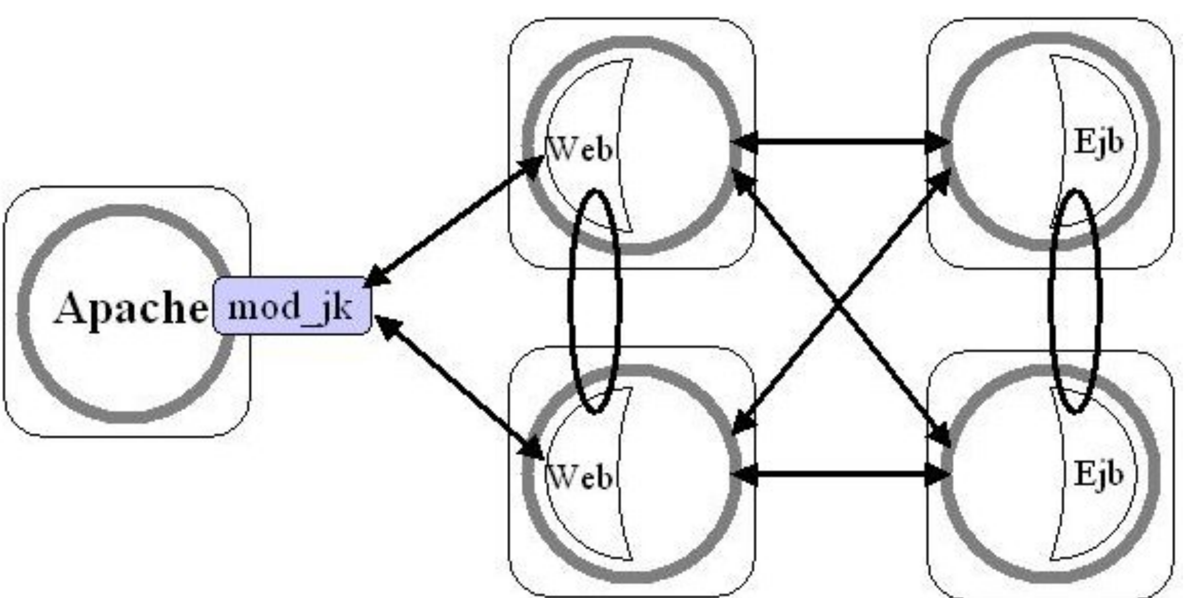
 # Factor used for this server in weighted round robin algorithms
carol.cmi.rr.factor=100
- For a fat Java client, specify the list of registries available in the carol.properties file:
  carol.cmi.url=cmi://server1:port1[,server2:port2...]

Note 1: The multicast address and group name must be the same for all JOnAS servers in the cluster.

Note 2: If Tomcat Replication associated to cmi is used, the multicast addresses of the two configurations must be different.

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:



Used symbols

Symbol	Description	Symbol	Description
	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		
Apache mod_jk	An Apache server with the mod_jk module		

References

- Working with mod_jk
- Tomcat workers Howto
- Apache JServ Protocol version 1.3 (ajp13)
- Apache – Tomcat HOWTO
- Apache 1.3.23 + Tomcat 4.0.2 + Load Balancing
- Tomcat 4 Clustering

Howto: Usage of AXIS in JOnAS

This guide describes basic Axis use within JOnAS. It assumes that the reader does not require any explanation about Axis−specific tasks (e.g., axis deployment with **WSDD**). Before deployment in Axis, the user must verify that the deploy.wsdd file **matches the site machine configuration** (jndiURL parameter in particular:*<parameter name="jndiURL" value="rmi://localhost:1099"/>*).

This document describes two ways to make an EJB (stateless SB) available as a Web Service with JOnAS:

1. Axis runs in a unique Webapp, the **stateless SB** (Session Bean) is packaged in a separate ejb−jar (or even EAR). The intent of this approach is to make EJBs from differents 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**.
2. 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.

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, ...)

(**JAX−M** and **JAX−R** are parts of the Web Services Development Pack from Sun.)

1. Unique Axis Webapp

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.

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*

Example: Refer to the *separate_axis* example (in the $JONAS_ROOT/examples directory).

2. Embedded Axis Webapp

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**.

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 AxisServlet.*

Example: Refer to the *embedded_axis* example (in the $JONAS_ROOT/examples directory).

3. 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.).

Tools:

Use jclient to deploy your Web Services (in the Axis way):
jclient org.apache.axis.client.AdminClient [OPTIONS] <WSDD−file>
[OPTIONS] :
−l<*URL*>: the **location of the AxisServlet** servlet (default : 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)

Howto: Using WebSphere MQ JMS guide

This document explains how WebSphere MQ (formerly MQSeries) can be used as JMS provider within a JOnAS application server.

WebSphere MQ is the messaging platform developed by IBM. It provides Java and JMS interfaces. Documentation is located at http://www–3.ibm.com/software/integration/mqfamily/library/manualsa/.

This document was written after integration work done with JOnAS 3.3 and 3.3.1 and WebSphere MQ 5.3.

The content of this guide is the following:

- Architectural rules
- Setting the JOnAS environment
    - Configuring the "Registry" server
    - Configuring the "EJB" server
- Configuring WebSphere MQ
- Starting the application
- Limitations

Architectural rules

WebSphere MQ, contrary to JORAM or SwiftMQ, cannot run collocated with JOnAS. WebSphere MQ is an external software which must be independently administered and configured.

Administering WebSphere MQ consists of the following:

- 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 via the registry. WebSphere MQ JMS objects are bound to the JOnAS registry. JMS lookups will then return the WebSphere MQ JMS objects, and messaging will take 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 that WebSphere MQ JMS objects have already been bound to the registry. Thus it is necessary to start an independent registry to which WebSphere MQ can bind its JMS objects, and which can also be used by the starting JOnAS server. The start–up sequence looks like the following:

1. Starting a registry.
2. Creating and binding WebSphere MQ JMS objects.
3. Launching the JOnAS server.

The following architecture is proposed:

- A JOnAS server (e.g., called "Registry") providing only a registry.
- A JOnAS server (e.g., called "EJB") using the registry service of server "Registry."
- A WebSphere MQ server running locally.

Setting the JOnAS Environment

The proposed architecture requires running two JOnAS server instances. For this, the following steps are proposed:

1. Create two base directories: e.g., `JONAS_REGISTRY` and `JONAS_EJB`.
2. Set the JONAS_BASE environment variable so that it points to 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 to the `JONAS_EJB` directory.
5. In the $JONAS_ROOT directory, type: *ant create_jonasbase*.

The JOnAS servers can now be configured independently.

Configuring the "Registry" server

The "Registry" server is the JOnAS server that will host 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 jeremie protocol:

```
carol.protocols=jeremie
```

Its port can also be configured:

```
carol.jeremie.url=jrmi://localhost:2000
```

Configuring the "EJB" server

The "EJB" server is the JOnAS server that will be 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
```

... and the JMS service as WebSphere MQ:

```
jonas.service.jms.mom    org.objectweb.jonas_jms.JmsAdminForWSMQ
```

In the carol.properties file, declare the jeremie protocol and set the correct port:

```
carol.protocols=jeremie
carol.jeremie.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.

Configuring WebSphere MQ

WebSphere MQ JMS administration is documented in chapter 5 of 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.jeremie.libs.services.registry.
                        jndi.JRMIInitialContextFactory
PROVIDER_URL=jrmi://localhost:2000
```

The JOnAS's `client.jar` library must also be added to the classpath for WebSphere MQ.

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 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 will print a message that looks like the following:

JmsAdminForWSMQ.start : WebSphere MQ XAConnectionFactory could not be retrieved from JNDI

This does not prevent JOnAS from working. However, if there is no connection factory available, no JMS operations will be possible from JOnAS.

If destinations have been declared in the `jonas.properties` file, JOnAS will also expect 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 destination 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 allows destinations to be created. Since it is not possible to create WebSphere MQ JMS objects from JOnAS, this will work only if the destinations are previously created and bound to the registry.

For example, if you want to create a queue named "myQueue" through the JonasAdmin tool, this will only work if:

- A queue has been created through the WebSphere MQ Explorer tool.
- The corresponding JMS `Queue` has been created and bound to the registry with the name "myQueue".

To launch WebSphere MQ administration tool, type: *JMSAdmin*

The following prompt appears: *InitCtx>*

To create a `QueueConnectionFactory` and binding it with name "JQCF", type:

InitCtx> DEF QCF(JQCF)

More parameters can be entered (for example for specifying the queue manager).

To create a `Queue` that represents a WebSphere MQ queue named "myWSMQqueue", and to bind it with name "sampleQueue", type:

InitCtx> DEF Q(sampleQueue) QUEUE(myWSMQqueue)

Objects bound in the registry can be viewed by typing:

InitCtx> DIS CTX

Starting the application

To start the registry server:

1. Clean the local CLASSPATH: *set/export CLASSPATH="".*
2. Set the JONAS_BASE variable so that it points to `JONAS_REGISTRY`.
3. Start the JOnAS server: *jonas start −n Registry.*

To administer WebSphere MQ:

1. In WebSphere MQ's `Java/bin` directory, launch the JMSAdmin tool: *JMSAdmin.*
2. Create the needed JMS objects.

To start the EJB server:

1. Clean the local CLASSPATH: *set/export CLASSPATH="".*
2. Set the JONAS_BASE variable so that it points to `JONAS_EJB`.
3. Start the JOnAS server: *jonas start −n EJB.*

To start 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* ...

Limitations

Using WebSphere MQ as JMS transport within JOnAS has some limitations, as compared with using JORAM or SwiftMQ.

First of all, WebSphere MQ is compliant with the old 1.0.2b JMS specifications. Code that is written following the JMS 1.1 latest spec (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. For this reason, a specific sample is provided.

Finally, for an unknown reason, asynchronous consumption of messages (through message−driven beans) does not work in transactional mode. Further inquiry is needed to resolve this issue.

Howto: Web Service Interoperability between JOnAS and Weblogic

This guide 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.).

This document describes the following two aspects:

1. Access a web service deployed on JOnAS from an EJB deployed on Weblogic server.
2. Access a web service deployed on Weblogic server from an EJB deployed on JOnAS.

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, including:

- **All libraries for using webservice are contained in webserviceclient.jar.**

Access a web service deployed on JOnAS from an EJB deployed on Weblogic server

Web Service Development on JOnAS

Also refer to the document How to use Axis with JOnAS, which describes how to develop and deploy web services on JOnAS.

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.

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"/>

<!-- 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.SerializerFactory"
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.SerializerFactory, and the deserializer used is my.java.DeserializerFactory.

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*
Note: from JOnAS 3.3, jclient no more include WebServices libraries in the CLASSPATH. So you have to manually add this jar in your CLASSPATH:
export CLASSPATH=$CLASSPATH:$JONAS_ROOT/lib/webservices_axis.jar

The web service WSDL is accessible from the url: http://<host>:<port>/<url−servlet>/<webservicename>?wsdl.

EJB proxy development for Weblogic Server

This EJB provides access to the web service deployed on JOnAS from Weblogic server.

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" keepGenerated="True"/>

This command creates four classes:

- Service implementation
- Java Interface
- Stub class
- Service interface corresponding web service

The tool can also generate the java classes corresponding to future complex types of web service.

Build the EJB

Then, call the web service in the EJB proxy code using these 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);
    };
```

Deploy the EJB on Weblogic Server

Deploy this EJB using the weblogic administration console.

Access a web service deployed on Weblogic server from an EJB deployed on JOnAS

Web Service Development for Weblogic Server

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>
```

THE ANT USED IS PROVIDED BY WEBLOGIC

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.

EJB proxy development for JOnAS

This EJB provides access to the web service deployed on Weblogic from JOnAS.

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 corresponding web service

The tool also generates the java class corresponding to future complex types of web service.

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);
    };
```

Deploy the EJB on JOnAS

Deploy the EJB using the JOnAS administration console or command.

Howto: RMI–IIOP interoperability between JOnAS and Weblogic

This guide describes the basic interoperability between JOnAS and Weblogic Server using RMI–IIOP (the examples in this document assume that the Sun rmi/iiop of the JDK is used).

The content of this guide is the following:

1. Accessing an EJB deployed on JOnAS from an EJB deployed on Weblogic server using RMI–IIOP.
2. Accessing an EJB deployed on Weblogic Server from an EJB deployed on JOnAS using RMI–IIOP.

Accessing an EJB deployed on JOnAS from an EJB deployed on Weblogic server using RMI–IIOP

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 JOnAS Configuration Guide.

For the JOnAS server to use RMI–IIOP, the JOnAS configuration requires modification. The iiop protocol must be selected in the
file carol.properties. Refer also to the JOnAS Configuration Guide for details about configuring the communication protocol.
This modification will allow an EJB to be created using the RMI–IIOP protocol.

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.

Access an EJB deployed on Weblogic Server by an EJB deployed on JOnAS using RMI−IIOP

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}"
```

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 type: "iiop://<server>:port."
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.

Howto: Interoperability between JOnAS and CORBA

This guide describes the basic interoperability between JOnAS and CORBA using RMI–IIOP (the examples in this document assume that the Sun rmi/iiop of the JDK 1.4 is used).

The content of this guide is the following:

1. Accessing an EJB deployed on JOnAS server by a CORBA client
2. Accessing a CORBA service by an EJB deployed on JOnAS server

Accessing an EJB deployed a on JOnAS server by a CORBA client

JOnAS Configuration

No modification to the EJB code is necessary. However, the EJB should be deployed for the iiop protocol (e.g. 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 JOnAS Configuration Guide 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 JOnAS Configuration Guide for details about configuring the communication protocol.
These modifications will make it possible to create an EJB using the RMI–IIOP protocol.

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. 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.

CORBA Client Development

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. Example: `idlj -fclient -emitAll package1/Hello.idl`

The idlj tool also generates bugged classes. Be sure to put the _read and _write method in comment in the class _Exception.java, CreateException.java, RemoveException.java.

Additionally, the class OutputStream.java, PrintStream.java, PrintWriter.java, Writer.java, 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.

2. 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);
        }
    }
}
```

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).

Accessing a CORBA service by an EJB deployed on JOnAS server

CORBA Service

Create the CORBA service.

Create the idl file corresponding to this service (e.g. the interface name, which is "Hello").

Generate the java file corresponding to the idl service with the `idlj -fall Hello.idl` tool.

Implement the java interface (in this example, the service will be bound with the name "Hello" in the server implementation).

Start the orb.

Start the CORBA service.

EJB on JOnAS

To call the CORBA service, generate the java file corresponding to the idl file.

For this, apply the idlj tool on the idl file corresponding to the CORBA service description:.

```
idlj -fclient Hello.idl
```

Then, create an EJB.

For calling the CORBA service, initialize the orb by specifying the host and the port.

Then, get the environment.

Get the java object corresponding to the CORBA service with the environment.

Call the method on this object.

Example code:

```
try {
          String[] h=new String[4];
          h[0]="-ORBInitialPort";
          h[1]=port;
          h[2]="-ORBInitialHost";
          h[3]=host;

          ORB orb=ORB.init(h,null);

          // get a reference on the context handling all services
          org.omg.CORBA.Object objRef=orb.resolve_initial_references("NameService");
```

```
NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);
Hello hello=HelloHelper.narrow(ncRef.resolve_str("Hello"));
System.out.println(hello.sayHello());
return hello.sayHello();
}
catch (Exception e) {
    ...
}
```

Howto: Migrate the New World Cruises application to JOnAS

This guide describes the modifications required for migrating the J2EE application New World Cruise to JOnAS server.

The content of this guide is the following:

1. JOnAS configuration
2. New World Cruise Application
3. SUN Web service
4. JOnAS Web service

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.

New World Cruise Application

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;
```

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 specify 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>)

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.

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.

SUN web service

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>;
```

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);.
```

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
```

JOnAS web service

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.xml
        -->
        <parameter name="beanJndiName"
                        value="ejb/CruiseManager"/>

        <!--
            use of remote interfaces to access the EJB is allowed
        -->
          <parameter name="homeInterfaceName"
                        value="cruisePack.CruiseManagerHome"/>
          <parameter name="remoteInterfaceName"
                        value="cruisePack.CruiseManager"/>


        <!--
            Specify here 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>
```

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>;
```

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.

Web Application

Finally, create the web application: the 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
```

Howto: Execute JOnAS as a WIN32 Service

This document describes the procedures necessary to run JOnAS as a system service on Microsoft Windows platforms. This applies starting from JOnAS 3.3.2.

Instructions

This procedure uses ANT targets that are introduced in JOnAS 3.3.2. The procedure also uses the Java Service Wrapper open source project which must be downloaded and installed separately.

Download and install Java Service Wrapper

1. Download Java Service Wrapper version 3.0.5 or later, and unzip the package to a directory in the local filesystem.
2. Set `WRAPPER_HOME` environment variable to the root directory for Java Service Wrapper.

   For example, if the package for Wrapper version 3.0.5 is unzipped into c:\jsw, then SET WRAPPER_HOME=c:\jsw\wrapper_win32_3.0.5

create_win32service

Before JOnAS can be run as a WIN32 service, it is necessary to copy Java Service Wrapper executable files to the %JONAS_BASE% directory and create a Java Service Wrapper configuration file. Prior to executing the steps in this section, it is necessary to create a JONAS_BASE directory as described in the JOnAS Configuration Guide.

1. Verify that JONAS_BASE and WRAPPER_HOME environment variables are set.
2. Set %JONAS_ROOT% as the current directory.
3. Execute **`ant [-Djonas.name=<server_name>] create_win32service`**.

   The `-Djonas.name=<server_name>` parameter is optional. If not specified, the default server name is 'jonas'.

   NOTE: it is necessary to execute `create_win32service` to regenerate wrapper configuration files whenever the JOnAS configuration is modified. Refer to the Modify JOnAS Configuration section for more information.

install_win32service

After the %JONAS_BASE% directory has been updated for use with Java Service Wrapper, JOnAS can be installed as a WIN32 service using the `install_win32service` ant target. Prior to installing the configuration as a WIN32 service, the configuration can be tested as a standard console application. Refer to the Testing configuration section for more information. The following steps will install the service.

1. Verify that JONAS_BASE and WRAPPER_HOME environment variables are set.
2. Set %JONAS_ROOT% as the current directory.
3. Execute **`ant install_win32service`**.

By default, the service is configured to start automatically each time Windows starts. If the administrator would prefer to start the service manually, modify the `wrapper.ntservice.starttype` parameter in the %JONAS_BASE%\conf\wrapper.conf file. Set the value as described in the comments found in the wrapper.conf file.

uninstall_win32service

When it is no longer desirable to run JOnAS as a Windows service, the service can be uninstalled using the `uninstall_win32service` ant target.

1. Verify that JONAS_BASE and WRAPPER_HOME environment variables are set.
2. Set %JONAS_ROOT% as the current directory.
3. Verify that the service has been stopped.
4. Execute **`ant uninstall_win32service`**.

start JOnAS service

To start the JOnAS service, open the Service Control Manager (Control Panel Services) window, select the JOnAS service and start the service.

By default, JOnAS will be started automatically each time Windows is started. After installing the service, it can be started manually to avoid the need to reboot Windows.

stop JOnAS service

To stop the JOnAS service, open the Service Control Manager window, select the JOnAS service and stop the service.

Files managed by create_win32service

The `create_win32service` ant target copies executable files from the Java Service Wrapper installation directory and generates a configuration file in the %JONAS_BASE% directory. The following files are managed by the `create_win32service` ant target.

- bin\Wrapper.exe
- bin\server.bat
- bin\InstallService–NT.bat
- bin\UninstallService–NT.bat

- lib\wrapper.jar
- lib\wrapper.dll

- conf\wrapper.conf
- conf\wrapper_ext.conf

NOTE: `wrapper_ext.conf` contains Java Service Wrapper configuration properties specific to the JOnAS service. This file is generated by the `create_win32service` ant target. Any changes made to this file will be lost when the `create_win32service` target is executed.

Modify JOnAS Configuration

Most of the JOnAS configuration is specified using property and XML files in the %JONAS_BASE%\conf directory. Changes to the files located in the %JONAS_BASE%\conf directory take effect the next time the service is started. It is only necessary to stop the service and restart the service for the changes to take effect.

In addition to the files located in the conf directory, JOnAS configuration is affected by the contents of %JONAS_ROOT%\bin\nt\config_env.bat, and by the CLASSPATH and JAVA_OPTS environment variables. If changes are made to config_env.bat, or to the CLASSPATH or JAVA_OPTS environment variables, it is necessary to update the Java Service Wrapper configuration files.

1. Using the Windows Service Control Manager, stop the JOnAS service.
2. Update the Java Service Wrapper configuration. Refer to the create_win32service section for details.
3. Test the updated configuration. Refer to the Testing configuration section for more information.
4. Using the Windows Service Control Manager, start the JOnAS service.

Testing configuration

After the Java Service Wrapper configuration files have been generated, it is possible to test the configuration in a console window before installing the configuration as a WIN32 service.

1. Verify that JONAS_BASE environment variable is set.
2. Execute **`%JONAS_BASE%\bin\server`**.

The Java Service Wrapper will start as a console application and load JOnAS using the configuration generated by the `create_win32service` ant target.

Enter **`CTRL-C`** to terminate JOnAS. After pressing Ctrl–C, the Java Service Wrapper displays the following messages to the execution report, and/or log file.

```
wrapper  | CTRL-C trapped.  Shutting down.
jvm 1    | 2003-12-02 15:25:20,578 : AbsJWebContainerServiceImpl.unRegisterWar
                                   : War /G:/w32svc/webapps/autoload/ctxroot.war no longer available
jvm 1    | Stopping service Tomcat-JOnAS.
wrapper  | JVM exited unexpectedly while stopping the application.
wrapper  | <-- Wrapper Stopped.
```

Howto: Getting Started with WebServices and JOnAS 3.X

- WebServices and J2EE
- Early Integration of Axis in JOnAS 3.X series
- How to use WebServices
    - Endpoint Creation
        - Exposing Stateless Session Bean
        - Exposing Simple class (JAX–RPC Class)
    - Client Creation
        - WSDL Knowledge
        - no WSDL Knowledge
    - Using Axis WSDD Configuration file
        - Service
        - Parameter
        - Optional
        - Mappings
        - Deployment
    - Deploy Created Web Service

WebServices and J2EE

WebServices are fully integrated in J2EE 1.4 compliant Application Servers.
Web services technologies can be seen as another communication protocol used to interoperate heterogeneous systems. Web services are based on the SOAP protocol. The SOAP messages can be transported with various transport mode (JMS, HTTP, mail, ...) but the focus of this document is the HTTP transport (because Web Services for J2EE v1.1 use only SOAP/HTTP).

Early Integration of Axis in JOnAS 3.X series

JOnAS hosts Apache Axis starting with version 3.0. This means that the user no longer must add Axis jars in its webapp.
To use an Axis class (AdminClient, WSDL2Java, ...) from the command line, simply use the jclient script in place of the common java executable (jclient will create the appropriate classpath).

How to use WebServices

Endpoint Creation

When it is desirable to expose some business methods as web services, a servlet container such as Tomcat or Jetty should be used to hold the Axis servlet where the SOAP messages of clients will ultimately end up.

To configure Axis, the developer must write a .wsdd file holding Axis−specific configuration parameters. This can be in one of two different forms: a simple wsdd containing only information about the web services the developer wants to expose (see an example here), or a complete wsdd containing exposed web services information AND Axis base configuration (see an example here).

The first form is preferred for development purposes, because the developer can easily change configuration values and submit them again to the Axis servlet (the Axis servlet will merge the current WEB−INF/server−config.wsdd with this new configuration file). It is usually named deploy.wsdd (its location is user−defined).

The second form is appropriate when used in a production environment where configuration changes are minor (ultimately, both forms results in the same Axis wsdd file). It MUST be named server−config.wsdd and located in the WEB−INF directory of the servlet.

Exposing Stateless Session Bean

1. Create a simple WebApp containing minimal Axis information, such as a WEB−INF/web.xml declaring the Axis servlet and servlet mapping (sample here).
2. Create a wsdd file (use the form you prefer)

- add a service with a unique name (used to find the appropriate service to invoke from the URL String)
- set a provider for the service: `java:EJB`

3. Add mandatory parameters for EJB exposition:

- `beanJndiName`: name used in Axis lookup. (a java:comp/env name can be set here if ejb−ref or ejb−local−ref is added in the web−app descriptor)
- `homeInterfaceName`: EJB Home interface fully−qualified class name
- `remoteInterfaceName`: EJB Remote interface fully−qualified class name
- `localHomeInterfaceName`: Local EJB Home interface fully−qualified class name
- `localInterfaceName`: Local EJB interface fully−qualified class name
- `allowedMethods`: comma−separated list of method names accessible via the web service

The developer must set AT LEAST local OR remote interface names. If local AND remote interfaces are specified, Axis will use the REMOTE interface, even if the interfaces are different.

Refer to "Using Axis WSDD configuration files" for more information.

Exposing Simple class (JAX−RPC class)

JAX−RPC classes are normal classes with no particular inheritance requirements, exposed as a web service.
1. Add the Axis servlet declaration in the web−app descriptor (sample here) with a servlet mapping.
2. Create a wsdd file (use whichever form you prefer).

- add a service with a unique name (used to find the appropriate service to invoke from the URL String)
- set a provider for the service: `java:RPC`

3. Add mandatory parameters to expose JAX−RPC classes:

- `className`: the fully−qualified name of the class to be exposed as a web service.
- `allowedMethods`: comma−separated list of method names accessible via the web service.

Client Creation

Creation of a web services client is heavily based on WSDL knowledge, even if it is not mandatory.

WSDL Knowledge

WSDL is the easiest way to create a web service client.
All that is necessary is to generate the files needed to access the web service, then compile them and add them into your component archive.
This can be done from the command line using the `org.apache.axis.wsdl.WSDL2Java` tool.

example:

```
  jclient org.apache.axis.wsdl.WSDL2Java --output <destination directory>
--NStoPkg
    <namespace>=<package> <wsdl url>
```

The `--NStoPkg` option is used to place generated classes in a convenient package according to the namespace of the WSDL Definition and the namespace(s) of the XML Schema(s).

Axis provides an Ant Task for automated build:

```
<taskdef name="axis-wsdl2java"
         classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask">
    <!-- classpath holds jonas.jar, webservices_axis.jar, ... -->
    <classpath refid="base.classpath"/>
</taskdef>

<axis-wsdl2java url="${ws.google.wsdl}/GoogleSearch.wsdl"
                output="${src.dir}">
    <mapping namespace="urn:GoogleSearch"
             package="org.objectweb.wssample.gen.google"/>
</axis-wsdl2java>
```

code :

```
import path.to.your.generated.classes.*;
[...]
<ServiceInterface> service = new <Service>Locator();
```

```
<PortInterface> port = service.get<PortName>();
<ObjectType> result = port.<methodName>(<arguments>);
```

No WSDL Knowledge

When the client does not have the WSDL of the service to access, the developer must use Service agnostic interfaces.

With Axis, you must instantiate an org.apache.axis.client.Service class (implementation of javax.xml.rpc.Service), create a javax.xml.rpc.Call instance from the Service, configure it manually, and invoke the Call.

Refer to the following example:

```
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.namespace.QName;
import javax.xml.rpc.encoding.TypeMappingRegistry;
import javax.xml.rpc.encoding.TypeMapping;
import org.apache.axis.encoding.ser.BeanSerializerFactory;
import org.apache.axis.encoding.ser.BeanDeserializerFactory;
[...]
// create a Service instance (other constructor usable, see the doc)
Service service = new org.apache.axis.client.Service();

// configure mappings if web service use Complex types
// first get the mapping registry instance
TypeMappingRegistry tmr = service.getTypeMappingRegistry();

// create a QName representing the fully qualified name of the
// XML type to serialize/deserialize (namespace+locapart pair)
QName xmlType = new QName("namespace-of-complex-type", "complex-type-name");

// get a TypeMapping (default for axis is SOAP encoding TypeMapping)
TypeMapping tm = tmr.getDefaultTypeMapping();

// register the XML type with a Java class by specifying
// Serializer/Deserializer factories to use.
tm.register(your.java.Type.class,
            xmlType,
            new BeanSerializerFactory(your.java.Type.class, xmlType),
            new BeanDeserializerFactory(your.java.Type.class, xmlType));

// get a Call instance from the Service
```

```
// specify port to use and operation name to be invoked
// see the doc for other createCall methods usage
Call call = service.createCall(new QName("port-name", new
QName("operation-name")));

// where is the web service ?
call.setTargetEndpointAddress("url-address-of-the-endpoint");

// now, we can invoke the web service with its parameters
String result = call.invoke(new Object[] {"Hello, World!"});
```

Using Axis WSDD Configuration file

This section covers the basic functionality of wsdd configuration files.
For detailed information refer to the Axis User Guide and Reference Guide.

Service

`<service>` is the tag used to declare a web service.
Attributes:

- `name`: unique name of the service
- `provider`: bridge used by Axis to invoke different objects: EJB, normal classes, CORBA objects, ...
- `style`: defines global format of the SOAP Message
- `use`: SOAP encoded message or message enclosing classic XML Elements (depends on the style attributes)

Provider possible values: java:EJB (used to expose EJB Stateless Session Bean), java:RPC (used to expose Simple classes), java:MSG (used to ???)
Style possible values: document, rpc
With RPC, SOAP Messages must have an element named as the operation to be invoked as their first soap:body element.
With Document, SOAP Messages cannot have a first element with the operation name.
Use possible values: encoded, literal
With encoded, SOAP refs can be used in the SOAP Message.
With literal, no SOAP refs can be used in the SOAP Message.
Note: a SOAP ref is similar to a pointer in an XML SOAP Message: an Element is defined with an ID that can be referenced from somewhere else in the SOAP Message.

Refer to the following snippet of Axis javadoc for style and use combinations:

Description of the different styles
style=rpc, use=encoded

First element of the SOAP body is the operation. The operation contains elements describing the parameters, which are serialized as encoded (possibly multi−ref)

```
<soap:body>
   <operation>
      <arg1>...</arg1>
      <arg2>...</arg2>
   </operation>
```

style=RPC, use=literal
First element of the SOAP body is the operation. The operation contains elements describing the parameters, which are serialized as encoded (no multi−ref)\

```
<soap:body>
   <operation>
      <arg1>...</arg1>
      <arg2>...</arg2>
   </operation>
```

style=document, use=literal
Elements of the SOAP body are the names of the parameters (there is no wrapper operation...no multi−ref)

```
<soap:body>
      <arg1>...</arg1>
      <arg2>...</arg2>
```

style=wrapped
Special case of DOCLIT where there is only one parameter and it has the same qname as the operation. In such cases, there is no actual type with the name...the elements are treated as parameters to the operation.

```
<soap:body>
   <one-arg-same-name-as-operation>
      <elemofarg1>...</elemofarg1>
      <elemofarg2>...</elemofarg2>
```

style=document, use=encoded
There is no enclosing operation name element, but the parameters are encoded using SOAP encoding This mode is not (well?) supported by Axis.

Parameter

`<parameter>` is the tag used to configure a service. It is basically a name−value pair.
Attributes:

- `name`: parameter name (key)
- `value`: parameter value

Common parameter:

- `className`: fully−qualified class name of service class (only used for provider RPC & MSG)
- `beanJndiName`: name used in Axis lookup. (a java:comp/env name can be set here if ejb−ref or ejb−local−ref is added in the web−app descriptor)
- `localHomeInterfaceName`: Local EJB Home interface fully−qualified class name
- `localInterfaceName`: Local EJB interface fully−qualified class name
- `homeInterfaceName`: EJB Home interface fully−qualified class name
- `remoteInterfaceName`: Remote EJB interface fully−qualified class name
- `allowedMethods`: comma−separated list of method names accessible via the web service (for all providers)
- `scope`: scope of the web service (`Request`: create a new Object for each request, `Session`: keep the same Object for a given Session ID, `Application`: keep the same Object for the entire application run)

Optional

Other options can be specified to a service without parameter tags:

- `wsdlFile`: specify the static wsdl file to be served when a ?WSDL request comes to the endpoint
- `namespace`: specify the web service namespace (override default namespace created from URL location of the service). The developer can handle the namespace value with this tag.

  By default a namespace will look like like the following:
  `http://ws.servlets.wssample.objectweb.org` (for a JAX−RPC class named `org.objectweb.wssample.servlets.ws`). The developer can write a more concise namespace as desired: for example, `urn:JaxRpcWsSample`.

    - `operation`: used to describe the operation exposed (avoid Axis to use reflection to discover the service interface). Refer to the Axis doc for details.

Mappings

Mappings in WSDD can be set at different levels: mappings commons for all services are direct children of the deployment tag, and mappings specific for a given web service are children of the service tag.
The developer can specify the type of mappings in the WSDD configuration file: beanMapping and typeMapping. beanMapping is a write shortcut for typeMapping when mapping bean types.
Mapping is used to register a java type, with an XML QName and a serializer/deserializer pair.

examples:

```
<typeMapping xmlns:ns="urn:my.namespace"
             serializer="your.java.serializer.factory"
             deserializer="your.java.deserializer.factory"
             qname="ns:my-xml-type-name"
             type="java:your.java.classname"
             encodingStyle="encoding-namespace"/>
Notes :
type value must be a qname in java prefixed namespace (java:XXX.YYY)
by default encodingStyle is set to http://schemas.xmlsoap.org/soap/encoding/
(SOAP 1.1 Encoding)
When Arrays of Complex Types are serialized and/or deserailized the factories
to be used are :
org.apache.axis.encoding.ser.ArraySerializerFactory
org.apache.axis.encoding.ser.ArrayDeserializerFactory

<beanMapping xmlns:ns="urn:my.namespace"
             languageSpecificType="java:your.java.classname"
             qname="ns:my-xml-type-name"/>
Notes :
serializer and deserializer are automatically set to BeanSerializerFactory and
BeanDeserializerFactory
encodingStyle is automatically set to null (cannot be overridden)
```

Deployment

A deployment Element is the root Element of any Axis WSDD file. It holds commons namespace definitions.

A normal deployment Element looks like the following:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <!-- ... services definitions here ... -->
</deployment>
```

Deploy Created Web Service

Deployment of the web service in JOnAS depends on the form of the wsdd file:
For full wsdd configuration files, simply add the wsdd file in the WEB–INF/ directory of the servlet with the name server–config.wsdd. Deployment will be performed automatically by Axis when the first client attempts to connect.
For simpler wsdd configuration files, deploy the web–app (or ear) into JOnAS (normal process). Then, use the Axis AdminClient tool:

```
jclient org.apache.axis.client.AdminClient
-lhttp://<hostname>:<port>/<web-app-context>/<servlet-mapping-to-Axis-Servlet>
```

`<your-deploy>.wsdd`

Apache−Axis org.apache.axis.transport.http.AxisHTTPSessionListener AxisServlet Apache−Axis Servlet org.apache.axis.transport.http.AxisServlet AxisServlet /servlet/AxisServlet AxisServlet /services/* 5 wsdl text/xml xsd text/xml index.html index.jsp index.jws

Howto: Distributed Message Beans in JOnAS 4.1

JOnAS release 4.1 dramatically simplifies the use of a distributed JORAM platform from within JOnAS servers. For example, such a configuration allows a bean hosted by JOnAS instance "A" to send messages on a JORAM queue, to which a MDB hosted by JOnAS instance "B" listens.

This advancement is due to the following:

- JORAM Resource Adapter allows a much more refined configuration than the JMS service did.
- JORAM provides a distributed JNDI server which allows JOnAS instances to share information.

Before going through this chapter, it is highly recommended that the JORAM Resource Adapter configuration guide be reviewed.

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 providing a method for sending a message on a JORAM queue. JOnAS B hosts a message–driven bean listening 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 that will be used for storing the shared information (basically, the queue's naming reference).

Common configuration

The JORAM servers are part of the same 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 up of two servers, "s0" and "s1", hosted by machines "hostA" and "hostB", listening on ports 16010, providing a distributed JNDI service (more info on JORAM's JNDI may be found here).

Each JOnAS server must hold a copy of this file in its `conf/` directory. In its `jonas.properties` file, each must declare the `joram_for_jonas_ra.rar` as a resource to be deployed (and each should remove *jms* from its list of services).

Specific configuration

JOnAS A embedds 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 embedds 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*** will be hosted by JORAM server s1. It must then be declared in the JOnAS B's `joram-admin.cfg` file as follows:

```
Queue   scn:comp/sharedQueue
```

The `scn:comp/` prefix is a standard way to specify which JNDI provider should be used. In this case, the shared queue will be 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 placed 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 placed 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
```

And now, 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. For doing so, 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>
```

Howto: install jUDDI server on JOnAS

- I. UDDI Servers
- II. jUDDI Overview
- III. How to Find the Latest Version
- IV. Install Steps
    - A. Create the jUDDI webapp
        - 1. Compilation
        - 2. Customization
    - B. Create the Database
        - 1. Retrieve the sql scripts for your database
        - 2. Setup the database
    - C. Configure JOnAS Datasource
    - D. Deploy and test jUDDI
- V. Links

I. 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, description, ...) inside these registries. Web service Consumers can browse these registries to choose a web services that fits their needs.

II. jUDDI Overview

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.

Refer to http://ws.apache.org/juddi for more information.

III. How to Find the Latest Version

JOnAS already includes jUDDI v0.8 as a preconfigured webapp. For the latest jUDDI version, refer to http://ws.apache.org/juddi.

IV. Install Steps

jUDDI requires a minimum of configuration steps to be successfully deployed inside JOnAS.
If you use the JOnAS provided juddi.war, skip step 1., "Compilation," and start with Step 2., "Customization."

A. Create the juddi webapp

1. Compilation

Go to the directory where the jUDDI sources are located (JUDDI_HOME).
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
```

Launch compilation with ant war. That will produce a juddi.war inside JUDDI_HOME/build/ directory.

2. Customization

JOnAS provide 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.

B. Create the Database

1. Retrieve the SQL scripts for your database

jUDDI comes with SQL files for many databases (MySQL, DB2, HSQL, Sybase, PostgreSQL, Oracle, TotalXML, JDataStore).
SQL Scripts are different for each version of jUDDI:

- MySQL
  - 0.8 : juddi_mysql.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- DB2
  - 0.8 : juddi_db2.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- HSQL
  - 0.8 : juddi_hsql.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- Sybase
  - 0.8 : juddi_ase.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- PostgreSQL
  - 0.8 : juddi_postgresql.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- Oracle
  - 0.8 : juddi_oracle.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- TotalXML
  - 0.8 : juddi_totalxml.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql
- JDataStore
  - 0.8 : juddi_jds.sql
  - 0.9rc1 : create_database.sql insert_publishers.sql

2. Setup the database

For the 0.8 jUDDI release, the given SQL script must be executed.
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.

For latest releases (0.9rc1), execute the given scripts (table creation, tmodels insertions and publishers insertions).

C. Configure JOnAS Datasource

As jUDDI uses a DataSource to connect to the database, JOnAS must be configured to create this DataSource.
This is done as usual in JOnAS:
Create a file (named ws–juddi–datasource.properties for example) and populate it with the appropriate database configuration information.

```
###################### MySQL DataSource configuration example
#  datasource.name is the jndi-name set in the jonas-web.xml
datasource.name           jdbc/juddiDB
# datasource.url is the URL where database can be accessed
datasource.url            jdbc:mysql://localhost/db_juddi
# datasource.classname is the JDBC Driver classname
datasource.classname      com.mysql.Driver
# Set here DB username and password
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 your
($JONAS_BASE|$JONAS_ROOT)/conf/jonas.properties
`jonas.service.dbm.datasources <datasource-filename>`.

D. Deploy and test jUDDI

Deploy jUDDI on JOnAS with a command line similar to the following:

```
$ jonas admin -a ~/juddi/juddi.war
```

The following output should display:

```
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 setup is successful, then your UDDI server can be accessed through any UDDIv2.0–compliant browser.

inquiryURL = http://localhost:9000/juddi/inquiry
publishURL = http://localhost:9000/juddi/publish

V. Links

- UDDI web site (http://uddi.org/)
- jUDDI web site (http://ws.apache.org/juddi)
- UDDI4J Java implementation (http://www.uddi4j.org)
- IBM UDDI Test Registry (https://uddi.ibm.com/testregistry/registry.html)
- Microsoft UDDI Test Registry (http://uddi.microsoft.com/)
- methods web Site (http://www.xmethods.net/)
- UDDI Browser (http://www.uddibrowser.org/)

Howto: JOnAS and JORAM: Distributed Message Beans

A How−To Document for JOnAS version 3.3

By Rob Jellinghaus, robj at nimblefish dot com
16 February 2004

We are developing an enterprise application which uses messaging to provide scalable data processing. Our application includes the following components:

- A servlet which provides a web user interface.
- Stateless session beans which implement our core business logic.
- Message beans which implement data processing tasks.

We wanted to arrange the application as follows:

- A front−end server, server1, running the servlet and session beans.
- Two back−end servers, server2 and server3, running the message beans.

We wanted to use JOnAS and JORAM as the platform for developing this system. We encountered a number of configuration challenges in developing the prototype. This document describes those challenges and provides solutions.

We constructed our system using JOnAS 3.3.1 −− many of these issues will be addressed and simplified in future JOnAS and JORAM releases. In the meantime we hope this document is helpful.

Thanks to Frederic Maistre (frederic.maistre at objectweb dot org) without whom we would never have figured it out!

JOnAS and JORAM: Configuration Basics

The JORAM runtime by default is launched collocated with the JOnAS server (see http://jonas.objectweb.org/current/doc/PG_JmsGuide.html#Running). However, in this configuration the JORAM lifetime is bound to the JOnAS lifetime. If the local JOnAS process terminates, so will the local JORAM. For reliability it is preferable to separate the JOnAS and JORAM processes, moreover given that a collocated JORAM server is by default non persistent.

The simplest configuration to separate JOnAS and JORAM, once they are non−collocated, is to create one JORAM instance on one machine in the system, and to couple all JOnAS instances to that one JORAM. However, this also is failure−prone as if that one JORAM instance quits, all the JOnAS instances will lose their connection −− and will not afterwards reconnect!

Hence, the preferred solution is to have one JOnAS instance and one JORAM instance on each participating server. The JORAM instances must then be configured to communicate with each other. Then each JOnAS instance must

be configured to connect to its local JORAM instance. This provides the greatest degree of recoverability, given that the JORAM instances are run in persistent mode (mode providing message persistence and thus, guarantee of delivery even in case of a server crash).

JORAM Topics and JOnAS Administration

The default configuration done by JOnAS is to create all queues and topics specified in jonas.properties when the JOnAS server starts up. In a multi−server configuration, this is not desired. JORAM topics and queues are hosted on one specific JORAM server. Other JORAM servers wishing to use those topics and queues must use JNDI lookups to retrieve remote instances of those topics and queues, and must bind them locally.

Moreover, each JORAM server must be launched with knowledge of its identity in the system, and each JOnAS instance must take different configuration actions depending on its role in the system. Hence, the configuration of each machine must be customized.

Finally, the default permissions for running a distributed JORAM environment are not compatible with JOnAS:

- Each JORAM instance must be launched with a "root" administration user whose password is "root", or the local JOnAS instance will not be able to establish its JORAM connection.
- Each JORAM instance must have an "anonymous" user created for it, or JOnAS message beans (which are anonymous users as far as JORAM is concerned) will be unable to receive messages. The JOnAS instance which creates the application's topics and queues will create its anonymous user as part of the topic and queue creation. The other JOnAS instances will not have any anonymous user, and must have one created for them.
- Each JORAM topic or queue used by the system must have its permissions set to allow all users to read and write to it, or the JOnAS anonymous message beans will be unauthorized to receive messages.

All this configuration is not part of JOnAS's or JORAM'S default administration logic. So it must be performed specifically by application code, which must perform this lookup and binding before any application JOnAS message operations can succeed.

The Solution

All these challenges can be addressed with the following set of configurations and supporting mechanisms.

Many variations are possible; we provide just the configuration that we have proved to work for us. It is possible to rearrange the configuration significantly (to have some queues hosted on some machines, and other queues on other machines; to use a distributed JNDI lookup rather than a centralized one; etc.), but we have not as yet done so.

Throughout we use our server1,server2, and server3 names as concrete examples of the configuration.

1. JORAM must be configured for distributed operation, roughly as described in section 3.2 of the JORAM adminstration guide (http://joram.objectweb.org/current/doc/joram3_7_ADMIN.pdf).
2. Each separate server machine must have its own instance of JORAM and its own instance of JOnAS.
3. Each JOnAS instance must be configured (via jonas.properties) to connect to its local JORAM.

4. The "server 0" JORAM instance must be launched first, followed by its associated JOnAS. This JOnAS instance, and this JOnAS instance only, is configured to create the queues and topics used in the system.
5. The second and third servers must then launch their JORAM and JOnAS (first JORAM, then JOnAS, then on to the next server) instances.
6. Each JOnAS server must implement a custom service (see http://jonas.objectweb.org/current/doc/Services.html) which, on startup, will perform the appropriate configuration for that specific server. We name this service the JoramDistributionService and provide source code for it below. This performs all the customized configuration described in the permission section above.
7. Since the configuration varies from server to server, the JoramDistributionService must read configuration information from a local configuration file. We place this file in the $JONAS_BASE/conf directory, from where it is loadable as a classloader resource. (This is a little−known JOnAS technique and it is not clear that it is guaranteed to work! −− if you know otherwise, please let me know: robj at nimblefish dot com.)

Summing up, the total configuration elements involved are:

1. $JONAS_BASE/conf/a3servers.xml −− the JORAM configuration file which specifies the distributed JORAM configuration. This file is identical on all participating servers.
2. $JONAS_ROOT/bin/<platform>/JmsServer −− the JORAM launch script which starts up JORAM. This varies on each server, the startup arguments (i.e. "0 ./s0", "1 ./s1", etc.) initialize the local JORAM instance with knowledge of its role in the JORAM configuration.
3. $JONAS_BASE/conf/jonas.properties −− the JOnAS configuration file. On all servers, this is extended to include the initialization of the JoramDistributionService, which must happen after the initialization of the "jms" service, but before the initialization of all deployment services (since application deployment involves subscribing message beans to queues and topics, which must be bound before the deployment can succeed). On the server which is to host the application's topics and queues, the jonas.properties file also specifies those topics and queues; on all other servers, no topics or queues are created. Finally, the jms service is configured as non−collocated on all servers, though customized to use the local JORAM instance's URL.
4. $JONAS_BASE/conf/joramdist.properties−− the configuration file for the JoramDistributionService. This contains properties specifying the local JORAM's port number, which server is hosting the application's topics and queues, and which topics and queues should be bound locally.

Note that the JoramDistributionService must be built and installed in $JONAS_BASE before JOnAS itself can be launched!

The Full Configuration

Here we provide examples of the relevant portions of the configuration for our system, to provide completely specific detail. Our application uses only queues (at the moment).

a3servers.xml:

```
<?xml version="1.0"?>
<config>
    <domain name="D1"/>
    <server id="0" name="S0" hostname="server1">
```

```
      <network domain="D1" port="16301"/>
      <service class="fr.dyade.aaa.ns.NameService"/>
      <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
      <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
               args="16010 root root"/>
    </server>
    <server id="1" name="S1" hostname="server2">
      <network domain="D1" port="16302"/>
      <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
      <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
               args="16011 root root"/>
    </server>
    <server id="2" name="S2" hostname="server3">
      <network domain="D1" port="16303"/>
      <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
      <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
               args="16012 root root"/>
    </server>
</config>
```

JmsServer: (the "export" is required for the script to work with the bash shell which we use on our Linux machines)

server 1:

```
export JAVA_OPTS="$JAVA_OPTS -DTransaction=fr.dyade.aaa.util.ATransaction -Dfr.dyade.aaa.agent.A3CONF_
jclient -cp "$JONAS_ROOT/lib/jonas.jar:$JONAS_ROOT/lib/common/xml/xerces.jar" fr.dyade.aaa.agent.Agent
```

server 2:

```
export JAVA_OPTS="$JAVA_OPTS -DTransaction=fr.dyade.aaa.util.ATransaction -Dfr.dyade.aaa.agent.A3CONF_
jclient -cp "$JONAS_ROOT/lib/jonas.jar:$JONAS_ROOT/lib/common/xml/xerces.jar" fr.dyade.aaa.agent.Agent
```

server 3:

```
export JAVA_OPTS="$JAVA_OPTS -DTransaction=fr.dyade.aaa.util.ATransaction -Dfr.dyade.aaa.agent.A3CONF_
jclient -cp "$JONAS_ROOT/lib/jonas.jar:$JONAS_ROOT/lib/common/xml/xerces.jar" fr.dyade.aaa.agent.Agent
```

The `Transaction` argument specifies the persistence mode of the started JORAM server. The `fr.dyade.aaa.util.ATransaction` mode provides persistence, when starting a server (server "s1" for example) a persistence root (`./s1`) is created. If re–starting s1 after a crash, the info contained in this directory is used to retrieve the pre–crash state. For starting a bright new platform, all servers' persistence roots should be removed.

For starting non persistent servers (which provided better performances), the mode to set is `fr.dyade.aaa.util.NullTransaction`.

jonas.properties: (we show only the portions which vary from the default)

server 1:

```
jonas.services  registry,jmx,jtm,dbm,security,jms,resource,joramdist,ejb,web,ear
jonas.service.joramdist.class   com.nimblefish.sdk.jonas.JoramDistributionService
jonas.service.jms.collocated    false
jonas.service.jms.url           joram://localhost:16010
jonas.service.jms.topics
jonas.service.jms.queues       ApplicationQueue1,ApplicationQueue2,ApplicationQueue3
```

server 2:

```
jonas.services  registry,jmx,jtm,dbm,security,jms,resource,joramdist,ejb,web,ear
jonas.service.joramdist.class   com.nimblefish.sdk.jonas.JoramDistributionService
jonas.service.jms.collocated    false
jonas.service.jms.url           joram://localhost:16011
#jonas.service.jms.topics
#jonas.service.jms.queues
```

server 3:

```
jonas.services  registry,jmx,jtm,dbm,security,jms,resource,joramdist,ejb,web,ear
jonas.service.joramdist.class   com.nimblefish.sdk.jonas.JoramDistributionService
jonas.service.jms.collocated    false
jonas.service.jms.url           joram://localhost:16012
#jonas.service.jms.topics
#jonas.service.jms.queues
```

joramdist.properties:

server 1:

```
joram.createanonuser=false
joram.port=16010
joram.bindremotehost=localhost
joram.bindremotequeues=WorkManagerQueue,StatusManagerQueue,InternalWorkQueue,ExternalWorkQueue
```

server 2:

```
joram.createanonuser=true
joram.port=16011
joram.bindremotehost=server1
joram.bindremotequeues=WorkManagerQueue,StatusManagerQueue,InternalWorkQueue,ExternalWorkQueue
```

server 3:

```
joram.createanonuser=true
joram.port=16012
joram.bindremotehost=server1
```

```
joram.bindremotequeues=WorkManagerQueue,StatusManagerQueue,InternalWorkQueue,ExternalWorkQueue
```

It is a bit odd that server 1, which hosts the queues locally, has a "bindremotequeues" property. In practice, the code which reads "bindremotequeues" actually also sets permissions, and then only binds the queues locally if bindremotehost is other than "localhost". In other words, the code was originally written before the permissions issue came to light, and so the names are a bit stale :–)

The JoramDistributionService

The only remaining piece to describe is the JoramDistributionService itself. Here it is. As mentioned, we do not use topics in our system; adding code to handle topic permission and binding would be completely straightforward.

```
package com.nimblefish.sdk.jonas;

import org.objectweb.jonas.service.Service;
import org.objectweb.jonas.service.ServiceException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.util.Enumeration;
import java.util.Properties;
import java.io.InputStream;
import java.io.IOException;

import javax.naming.Context;
import javax.jms.Destination;

import fr.dyade.aaa.joram.admin.AdminItf;

/**
 * This class implements the JOnAS service interface and performs
 * JOnAS-startup-time configuration actions relating to distributed JORAM servers.
 *
 * It uses a properties file named "joramdist.properties" to configure its activity;
 * this configuration file must be in $JONAS_BASE/conf (which is part of the JOnAS
 * classpath, and hence is reachable as a classloader resource from this class.)
 * This of course can be changed at your discretion.
 *
 * See http://jonas.objectweb.org/current/doc/Services.html
 *
 * Written by Rob Jellinghaus (robj at nimblefish dot com) on 11 February 2004.
 * Thanks very much to Frederic Maistre (frederic.maistre at objectweb dot org)
 * for his indispensable and voluminous help.
 * This file is hereby placed into the public domain for use by JOnAS users at their
 * sole discretion; please include this comment text in your uses of this code.
 */
public class JoramDistributionService implements Service {
    private static Log log = LogFactory.getLog(JoramDistributionService.class);

    private boolean createAnonUser = false;
    private int joramPort = -1;
```

```
    private int joramInstance = -1;
    private String joramBindHost = null;
    private String[] joramBindQueues = null;

    public void init(Context context) throws ServiceException {
        log.info("JoramDistributionService initializing");
        try {
            InputStream propStream = JoramDistributionService.class.getClassLoader()
                .getResourceAsStream("joramdist.properties");
            Properties joramProperties = null;
            joramProperties = new Properties();
            joramProperties.load(propStream);
            Enumeration props2 = joramProperties.propertyNames();
            while (props2.hasMoreElements()) {
                String s = (String) props2.nextElement();
                log.info("joram.properties property: "+s+": "+joramProperties.getProperty(s));
            }

            if (joramProperties.containsKey("joram.createanonuser")
                && joramProperties.get("joram.createanonuser").equals("true")) {
                createAnonUser = true;
            }

            if (joramProperties.containsKey("joram.port")) {
                joramPort = Integer.parseInt(joramProperties.getProperty("joram.port"));
            }

            if (joramProperties.containsKey("joram.instance")) {
                joramInstance = Integer.parseInt(joramProperties.getProperty("joram.instance"));
            }

            if (joramProperties.containsKey("joram.bindremotehost")) {
                joramBindHost = joramProperties.getProperty("joram.bindremotehost");
            }

            if (joramProperties.containsKey("joram.bindremotequeues")) {
                joramBindQueues = joramProperties.getProperty("joram.bindremotequeues").split(",");
            }

        } catch (IOException e) {
            throw new ServiceException("Could not initialize JoramDistributionService", e);
        }

    }

    public void start() throws ServiceException {
        started = true;

        if (joramPort == -1 && joramInstance == -1)  {
            log.info("No joram.port and/or joram.instance defined; performing no JORAM configuration.
            return;
        }

        try {
```

```
    if (joramPort != -1) {
        AdminItf admin = new fr.dyade.aaa.joram.admin.AdminImpl();
        admin.connect("localhost", joramPort, "root", "root", 60);

        if (createAnonUser) {
            log.info("Creating JORAM anonymous user on localhost:"+joramPort+
                " for instance "+joramInstance+"...");
            admin.createUser("anonymous", "anonymous", joramInstance);
        }

        log.info("Created JORAM anonymous user.");

        if (joramBindHost != null && joramBindQueues != null) {
            log.info("Looking up JNDI queues from rmi://"+joramBindHost+":1099");
            javax.naming.Context jndiCtx;

            java.util.Hashtable env = new java.util.Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.rmi.registry.RegistryContextFactory");
            env.put(Context.PROVIDER_URL, "rmi://"+joramBindHost+":1099");
            jndiCtx = new javax.naming.InitialContext(env);

            Object[] remoteTopics = new Object[joramBindQueues.length];
            for (int i = 0; i < joramBindQueues.length; i++) {
                String joramBindQueue = joramBindQueues[i];
                remoteTopics[i] = jndiCtx.lookup(joramBindQueue);
                log.debug("Got queue "+joramBindQueue+": "+remoteTopics[i]);

                // open up all topics to everyone
                admin.setFreeReading((Destination)remoteTopics[i]);
                admin.setFreeWriting((Destination)remoteTopics[i]);
            }

            // if we are on local host, don't rebind
            if (!joramBindHost.equals("localhost")) {
                env.put(Context.PROVIDER_URL, "rmi://localhost:1099");
                jndiCtx = new javax.naming.InitialContext(env);

                for (int i = 0; i < joramBindQueues.length; i++) {
                    jndiCtx.bind(joramBindQueues[i], remoteTopics[i]);
                    log.debug("Bound "+joramBindQueues[i]+" in localhost context");
                }
            }
        }

        // Disconnecting the administrator.
        admin.disconnect();
    }

    log.info("Completed JoramDistributionService startup successfully.");
} catch (Exception e) {
    throw new ServiceException("Could not start JoramDistributionService", e);
}
```

```
    }

    private boolean started = false;
    private String name;

    public void stop() throws ServiceException {
        started = false;
        log.info("JoramDistributionService stopped");
    }

    public boolean isStarted() {
        return started;
    }

    public void setName(String s) {
        name = s;
    }

    public String getName() {
        return name;
    }
}
```

This needs to be built with a simple task that just compiles the class and places it in a JAR file, which then must be placed in the $JONAS_ROOT/lib/ext or $JONAS_BASE/lib/ext directories on each server before launching JOnAS.

Maintaining the configuration

This is clearly a fairly large number of small configuration files on each server. We have automated the process of deploying the servers and their configuration via Ant. Ant 1.6 includes native support for scp and ssh operations, as Ant tasks. We have used these to build Ant tasks which can literally:

1. install JOnAS on all our servers.
2. create JONAS_BASE directories on each server,
3. copy the server−specific configuration over to each server,
4. build the JoramDistributionService and deploy it to each server,
5. launch JORAM and JOnAS on each server in the proper order,
6. build a customized version of our application for each type of server (i.e. a "frontend" version containing no message beans, and a "backend" version containing only message beans),
7. deploy the appropriate application version to each of the three servers,
8. and test the entire system using our system integration test suite.

In fact, we can do all of the above with a single Ant command.

Doing this with Ant is actually quite straightforward. Without support for automating this deployment process, we would be quite concerned with the complexity of the configuration. With automation, it is easy to place the whole configuration process under source code control, and it is easy to make controlled changes to the configuration of

multiple machines.

Conclusion

Sorting out all these details was a long job (all the longer as we are located in San Francisco and Frederic, our lifeline, is in France, time−shifted by half a day!). However, the whole system does work now, and works well. JOnAS and JORAM are very impressive pieces of work, and the 4.0 releases of both promise to be even more so.

We look forward to continued use of the ObjectWeb platform, and we hope to continue to contribute constructively to the ObjectWeb community. You may also be interested in our description of using JOnAS with Hibernate, at http://www.hibernate.org/166.html.

Sincerely,
Rob Jellinghaus (robj at nimblefish dot com)

Howto: JSR 160 support in JOnAS

The content of this document is the following:

1. Target Audience and Rationale
2. What is JSR 160 ?
3. Connector servers created by JOnAS

Target Audience and Rationale

Starting with version 4.1.4 JOnAS provides support for remote connection to the MBean server in a standard way.

The target audience for this document is management application developper or administrator, intending to use standard JMX RMI connectors to access the MBean Server running in a JOnAS server.

What is JSR 160 ?

The JSR 160 is a JMX Remoting specification which 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.

Connector servers created by JOnAS

Previous and current JOnAS versions implement a proprietary remote object allowing to connect 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, JEREMIE – see Choosing the Protocol).

JSR 160 support implies providing standard connector server objects. The JMX service creates at start–up one or several 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. Below we present the URLs that can be used by the clients depending on the protocol they choose.

Currently only 2 protocols can be used by JSR–160 connectors: RMI/JRMP and RMI/IIOP.

Using a RMI/JRMP Connector

This connector can be used if the `jrmp` protocol is set 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:port/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.

Example 1:

```
Hashtable environment = null;
JMXServiceURL address = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://host:1099/jrmpconnect
JMXConnector connector = JMXConnectorFactory.newJMXConnector(address, environment);
connector.connect(environment);
```

Using a RMI/IIOP Connector

This connector can be used if the `iiop` protocol is set in the `carol.protocols` list.

The client code is simmilar to the JRMP case, but the `String` to be used to construct the `JMXServiceURL` has to respect the following model:
`"service:jmx:iiop:///jndi/iiop://host:port/iiopconnector_`*`jonasServerName`*`"`

Example 2:

```
Hashtable environment = null;
JMXServiceURL address = new JMXServiceURL("service:jmx:iiop:///jndi/iiop://host:2001/iiopconne
JMXConnector connector = JMXConnectorFactory.newJMXConnector(address, environment);
connector.connect(environment);
```

Howto: Using the MC4J JMX Console

This guide describes how to connect the MC4J JMX Console to the JMX server running in a JOnAS server.

Recall that the current JMX implementation used by JOnAS is MX4J, and that JOnAS supports JSR 160 for remote connections to the JMX server (see the corresponding Howto document).

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 the JSR 160 Support in JOnAS Howto document for specific JOnAS configurations.

Once connected, you can manage the connection (delete, disconnect or reconnect) and browse the MBeans registered in the JMX server.

Howto: JOnAS and JMX, registering and manipulating MBeans

By Jonny Way.

Introduction

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. The intent of this document is to illustrate one method of registering application−specific MBeans with the JOnAS JMX server based on the m−let service.

ServletContextListener

The basic task of registering an MBean with the JOnAS JMX server is accomplished by the following implementation of the ServletContextListener interface. 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 delminated list (although this is not really
 * needed as multiple mlets can be specifyed 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(fi
                        if(configURL == null){
                                log.error("Cound not load MLet file resource from "+filename+" using c
                        }
                        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 serv
                        }
                        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 registrat
                                        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 "
                                                Set loadedMBeans = (Set)server.invoke(name, "
                                                                new Object[]{url}, new String

                                                processRegisteredMBeans(loadedMBeans);
                                        }
                                        catch (InstanceNotFoundException e) {
                                                log.error("Unable to register MBeans from MLe
                                        }
                                        catch (MBeanException e) {
                                                log.error("Unable to register MBeans from MLe
                                        }
                                        catch (ReflectionException e) {
                                                log.error("Unable to register MBeans from MLe
                                        }
                                }
                        }
                        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 registe
```

```
                        }
                }
                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);
                        }
                }
        }

}
```

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 the web.xml will result in the registeration 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>
```

Library Dependences

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.

HibernateService Extension

The Hibernate distribution provides an implementation of HibernateServiceMBean in the class HibernateService. In the MLet file above, an extension of this class is specified 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 allows 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 can be used directly.

```
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
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 HibernateExc
            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", StringBuf
            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;
    }
}
```

Howto: Using JOnAS through a Firewall

The content of this guide is the following:

1. Target Audience and Rationale
2. RMI/IIOP through a Firewall

Target Audience and Rationale

The target audience for this document is JOnAS administrators intending to specify the range of ports to be used by RMI on JOnAS, in order to pass through firewalls.

RMI/IIOP through a Firewall

This section 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.

The following example uses a range of 10 ports, starting at port 33000. The IP address 150.11.135.17 is also given as an example for the host running JOnAS.

$JONAS_BASE/jacorb.properties file

The following properties of this file should be uncommented and specified:

```
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
```

$JONAS_BASE/carol.properties file

Use of the iiop protocol should be specified, and the port number for the registry must be the first port number identified in the range above.

```
carol.protocols=iiop
carol.iiop.url=iiop://150.11.135.17:33000
```

Launching JOnAS

This configuration for specifying the port numbers will not work when launching JOnAS in the background; it must be launched in the foreground:

```
jonas start -fg
```

How to configure and use xdoclet for JOnAS

This guide contains the following sections:

- Downloading and installing xdoclet
- Ejbdoclet Ant target
- Xdoclet tags for JOnAS

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.

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: JOnAS examples (xdoclet and olstore) use the **`$JONAS_ROOT/lib/example/xdoclet/xdoclet-objectweb-module-1.2.2.jar`**.

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>
```

Xdoclet tags

Field–Level Tags

- **@jonas.bean**
- **@jonas.resource**
- **@jonas.resource–env**
- **@jonas.ejb–ref**
- **@jonas.session–timeout**
- **@jonas.is–modified–method–name**
- **@jonas.shared**
- **@jonas.passivation**
- **@jonas.max–cache–size**
- **@jonas.min–pool–size**
- **@jonas.jdbc–mapping**
- **@jonas.jonas–message–driven–destination**

Method–Level Tags

- **@jonas.ejb–relation**
- **@jonas.cmp–field–jdbc–mapping**

Field–Level Tags

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			false

			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.	
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

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**

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**

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**

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**

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**

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**

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**

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.

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.

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**

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**

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**

Method Level tags

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**
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

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