JOnAS 4.9 EE Configuration guide

Guide for configuring one or a group of servers

JOnAS Team (Philippe Coq, Adriana Danes, Sava# Ali Tokmen)

- January 2008 -

Table of Contents

Chapter 1. Introduction

1.1. configuring JOnAS

Configuration is a task that may be more or less complex. Configuring a unique instance is obviously easier than configuring a cluster of servers.

Configuration task consists mainly in customizing a set of JOnAS configuration files that compose the JOnAS environment see Section 2.1, “Configuring JOnAS Environment : JONAS_BASE”.

First of all, some terms used is this document must be defined:

1.2. Terminology

1.2.1. Server or JOnAS instance

A server or JOnAS instance is a java process started via the *jonas start* command or via the administration tool JonasAdmin.

Several servers may run on the same physical host.

1.2.2. Service

When a server starts, services are started.

A service typically provides system resources to containers. Most of the components of the JOnAS application server are pre-defined services. However, it is possible and easy for an advanced user to define a new service and to integrate it into JOnAS.

JOnAS services are manageable through JMX.

1.2.3. Container

A container consists of a set of Java classes that implement the Java EE specification. The role of the container is to provide the facilities for executing J2EE components.

There are three types of containers:

- EJB container in which Enterprise JavaBeans are deployed and run
- Web container for JSPs and servlets
- Client container

1.2.4. Domain

A domain represents an administration perimeter which is under the control of an administration authority.

This perimeter contains management targets like servers and clusters.

If a domain contains several elements, it provides at least one common administration point represented by a master server.

1.2.5. Master server

A master is a JOnAS instance having particular management capabilities within the domain:

- it is aware of the domain's topology
- it allows management and monitoring of all the elements belonging to the domain

1.2.6. Cluster

A cluster is a group of server instances. It usually allows to run a J2EE application, or a J2EE module, on the cluster members as if they were a single server. The objective is to achieve applications scalability and high availability.

JOnAS supports several cluster types:

- Clusters for Web level load-balancing
- Clusters for high availability of Web components
- Clusters for EJB level load-balancing
- Clusters for high availability of EJB components
- Clusters for JMS destination scalability and high availability
- Clusters for administration purpose which facilitate management operations like deployment /undeployment.

From the administrator point of view, a cluster represents a single administration target.

Note that a particular JOnAS server may belong to zero, one or more clusters.

Chapter 2. Configuring a JOnAS instance

JOnAS is pre-configured and ready to be used directly. The Getting Started book [GettingStarted.html#GettingStarted] has shown that a very sample example may be run after JOnAS installation without any configuration task. But as soon as your application needs to use resources specific to the execution environment, configuration is mandatory.

In this chapter we will see in a first part where are the configuration files and then what that can be configured

2.1. Configuring JOnAS Environment : JONAS_BASE

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

2.1.1. JONAS_BASE structure

JONAS_BASE has the following structure:

- the `apps` directory

  where Java EE application files (*.ear)* are installed. The subdirectory *autoload* contains applications that will be deployed at starting time.

- the `clients` directory

  where Java EE Client Applications (*.jar)* files are installed.

- the `conf` directory

  contains JOnAS configuration files.

- the `ejbjars` directory

  where Enterprise Beans packaged into *ejb-jar* files are installed.The subdirectory *autoload* contains *ejb-jar* files that will be deployed at starting time.

- the `lib` directory [1]

  Used for extending class loaders. It contains four sub directories:

directory	description
`apps`	for apps ClassLoader

`commons`	for the commons ClassLoader
`ext`	the same usage than `commons`
`tools`	for the tools ClassLoader

- the `logs` directory

  where the log files are created at run-time

- the `rars` directory

  where ResourceAdapters packaged into `.rar` files are installed.The subdirectory `autoload` contains ResourceAdapters that will be deployed at starting time.

- the `webapps` directory

  where Web components packaged into `.war` files are installed.The subdirectory `autoload` contains Web applications that will be deployed at starting time.

- the `work` directory

  a working directory forJOnAS

2.1.2. JONAS_BASE creation

1. Create a JONAS_BASE template from scratch :

```
export JONAS_BASE=~/my_jonas_base
cd $JONAS_ROOT
ant create_jonasbase
```

   This will copy all the required files and create all the directories.

2. Another way to create a JONAS_BASE template from scratch :

```
export JONAS_BASE=~/my_jonas_base
newjb
```

   the JONAS_BASE created with the command newjb allow to run the JOnAS conformance test suite.

3. To update a JONAS_BASE from a new JONAS_ROOT:

```
export JONAS_BASE=~/my_jonas_base
cd $JONAS_ROOT
ant update_jonasbase
```

2.1.3. JONAS_BASE/conf description

This directory contains configuration files in various format (properties file, xml files).

The main configuration file is *jonas.properties* but there are also:

- Templates for configuring access to databases (Oracle, PostgreSQL, Sybase, DB2, MySQL, HSQLDB, InterBase, FirebirdSQL, Mckoi SQL, InstantDB) respectively in *Oracle1.properties*, *PostgreSQL1.properties*,etc... All these databases have been tested with JOnAS.
- Mail resources templates : *MailMimePartDS1.properties*, *MailSession1.properties*
- JORAM configuration files : *a3debug.cfg*, *a3servers.xml*, *joramAdmin.xml*

- *carol.properties*, *jacorb.properties*, *jonathan.xml* [2] for configuring the RMI implementation used through CAROL.
- Configuration files for clustering : *clusterd.xml, domain.xml, jgroups-ha.xml, jgroups-cmi.xml*,*jk2.properties*
- Configuration files related to security: *jaas.config, java.policy, jonas-realm.xml*
- Web container configuration files:
  - *server.xml, context.xml ,web.xml* for Tomcat,
  - *jetty5.xml jetty5-webdefault.xml* for Jetty.
- Web services configuration files: *jaxr.properties, uddi.properties, file1.properties.*
- Client container configuration file: *jonas-client.properties*
- JOnAS traces configurations files: *trace.properties, traceclient.properties*
- Speedo configuration file: *speedo-jdo.properties*
- Transaction recovery configuration file : *jotm.properties*
- P6Spy options file: *spy.properties*
- Thread management framework configuration file: *jonas_areas.xml*
- Fractal deployment framework configuration file: *execute.properties*
- Java Service Wrapper configuration file: *wrapper.conf*

Most of these files are described in following sections.

2.1.4. jonas.properties file: the key configuration file

$JONAS_BASE/conf/jonas.properties is the key file for configuring JOnAS.

This file is used for:

- setting some global properties for the JOnAS instance
- choosing the list of JOnAS services wanted
- customizing each services.

2.1.4.1. Global properties of jonas.properties

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

# Enable the rmi Security manager
jonas.security.manager    true

# Enable csiv2 security propagation for rmi/iiop
jonas.csiv2.propagation    true

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

#  Set the name of log configuration file to trace.properties
jonas.log.configfile    trace
```

Properties are self commented.

Note

setting jonas.security.manager to `false` implies a colocated registry and implies to set in `carol.properties`:

```
carol.jvm.rmi.local.registry=true
```

2.1.4.2. List of JOnAS services :

Here is the list of default services activated at starting time:

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

The possible services are:

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.
jmx	this service is needed in order to administrate theJOnAS servers and the JOnASservices via a JMX-based administration console.
jtm	the transaction manager service is used for support of distributed transactions with JOTM.
db	this service is used for launching a Java database implementation. By default, HSQLDB java database is used.
dbm	the database service is needed by application components that require access to one or several relational databases. It may be an alternative to the usage of a JDBC resource adapter via the resource service.
security	this service is needed for enforcing security at runtime.
resource	this service is needed to use resource adapters conformant to the J2EE Connector Architecture Specification
ejb	the EJB container service provides support for EJB components.
ws	the WebServices service provides support for WebServices (WSDL publication).
web	the WEB container service provides support for web components (as Servlets and JSP). JOnAS provides two implementations of this service, one based on Tomcat and another on Jetty.
ear	the EAR service provides support for J2EE applications.
ha	the High Availability service provides stateful session beans replication.
mail	this service is required by applications that need to send e-mail messages.
discovery	this service allows dynamic administration of management domains
jms	this service may be used for components that use the Java Message Service API , or that use message-driven beans (with some restrictions). The compliant way is to use a JMS provider through the deployment of a resource adapter and to use resource service instead.
thread	this service is needed to use the Thread Management Framework [TMF.html].

Services will be started in the order in which they appear in the list. Therefore, the following constraints should be considered:

- **jtm** must precede the following services: **dbm**, **resource**, **jms** and **ejb**
- **ejb**,**ws**,**web** must precede **ear**
- if **security** service use realm in database it must be after **dbm** or **resource**.
- 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 send e-mail messages, **mail** must precede **ejb** in the list of required services.

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 it is automatically launched after the **registry.**

2.1.4.3. Customizing services in jonas.properties

Configuration parameters for services follow 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

each service **XX** must contain the property jonas.service.XX.class indicating the name of the java class that implements the service:

jonas.service.XX.class aa.bb.XXImpl

This allow experimented user to replace built-in service by an alternative implementation.

For example here is the part of `jonas.properties` file related to the customization of **jtm** service:

```
#  Set the name of the implementation class of the jtm service
jonas.service.jtm.class  org.objectweb.jonas.jtm.TransactionServiceImpl
#  Set the Transaction Manager launching mode.
#  If set to 'true', TM is remote: TM must be already launched in an other JVM.
#  If set to 'false', TM is local: TM is going to run into the same JVM
#  than the jonas Server.
jonas.service.jtm.remote    false
#  Set the default transaction timeout, in seconds.
jonas.service.jtm.timeout    60
```

see Section 2.4, “Configuring JOnAS Services” for a complete description of the services configuration.

2.2. Configuring the communication protocol and JNDI

JOnAS provides a multi-protocol support through the integration of the CAROL component.

Supported communication protocols are the following:

- RMI/JRMP is the JRE implementation of RMI on the JRMP protocol. This is the default communication protocol.
- RMI/IIOP is the JacORB [http://www.jacorb.org/] implementation of RMI over the IIOP protocol.
- IRMI is an RMI implementation that can be used with Open Source JDK that doesn't provide `com.sun.*` classes.

- CMI (Cluster Method Invocation) is the communication protocol used for clustered configurations.[3]

2.2.1. Choosing the Protocol

The choice of the protocol is made in the carol.protocols property of `carol.properties` file in `JONAS_BASE/conf` directory.

```
carol.protocols=jrmp
```

2.2.1.1. configuring jrmp protocol

```
carol.protocols=jrmp 1
carol.jrmp.url=rmi://localhost:1099 2
carol.jvm.rmi.local.call=false 3
carol.jvm.rmi.local.registry=false 4
carol.jrmp.server.port=0 5
carol.jrmp.interfaces.bind.single=false 6
```

1. choice of the protocol or list of protocols
2. connexion url to the RMI registry the hostname (localhost) and port number must be changed if needed. In a distributed configuration changing the hostname is mandatory.
3. if true local calls are optimized: calls to methods of the remote interface are treated as call to local methods (it is not always possible depending on the packaging of the application).
4. if true a local Naming context is used. This must be used only with a collocated registry and it is mandatory when the jonas.security.manager property of jonas.properties is set to true.
5. exported objects will listen on this port for remote method invocations. 0 means random port. Specify a port may be useful when the server run behind a firewall.
6. if true use only a single interface (choosen from the url) when creating the registry. False means use all interfaces available.

2.2.1.2. configuring RMI/IIOP protocol

The JacORB implementation of RMI over the IIOP is used. The configuration file of JacORB is the `$JONAS_BASE/conf/jacorb.properties` file.

As for the other protocols RMI over IIOP is ready to used in the default distribution. It is only for tuning purpose that the `$JONAS_BASE/conf/jacorb.properties` file must be customized.

By default the CORBA Naming service is run using the port 2001 (as it is set in the `carol.properties` file)

So the only thing to do for working in RMI over IIOP is to set the property protocols in carol.properties:

```
carol.protocols=iiop
# RMI IIOP URL
carol.iiop.url=iiop://localhost:2001
carol.iiop.server.port=0 1
carol.iiop.server.sslport=2003 2
carol.iiop.PortableRemoteObjectClass=org.objectweb.jonas_lib.naming.JacORBPRODelegate  3
```

1. 0 means random port
2. this port is used only if SSL mode is enabled (default configuration = not used).Is used to set the SSL port of the objects listener
3. delegate used by JOnAS for rmi-iiop protocol.

2.2.1.3. configuring irmi protocol

```
carol.protocols=irmi
carol.irmi.url=rmi://localhost:1098 1
carol.irmi.server.port=0 2
carol.irmi.interfaces.bind.single=false 3
```

1 for irmi the default port is 1098
2 exported objects will listen on this port for remote method invocations:0 means random port.

Caution

if the port is set to n the port 'n + 1' will be used by the JMX server.So, for the firewall configuration, you have to open the port numbers 'n' and 'n+1'

3 if true use only a single interface when creating the registry (specified in carol.irmi.url property). Default configuration = false(use all interfaces available)

2.2.1.4. configuring cmi protocol

CMI is the protocol to use for clustering purpose.

CMI brings its own registry that implements JNDI replication. CMI relies on JGroups [http://www.jgroups.org/javagroupsnew/docs/index.html] group-communication protocol for ensuring the global registry replication. CMI provides jndi high availability, the load-balancing and fail-over at the EJB level.

For using CMI the protocol must be set in `carol.properties`:

```
carol.protocols=cmi
carol.cmi.url=cmi://host:2002 1
carol.cmi.jgroups.conf=jgroups-cmi.xml 2
carol.cmi.multicast.groupname=G1 3
```

1 for CMI the default port is 2002
2 name of the JGroups configuration file
3 groupname for JGroups. All instances in a same cluster must share the same groupname.

The `$JONAS_BASE/conf/jgroups-cmi.xml` file defines the setting of the JGroups protocol stack. By default this stack uses UDP protocol and multicast IP for broadcasting CMI registry updates.

Further information about clustering and CMI configuration can be found in Chapter 4, *Configuring a cluster*

2.2.1.5. multi protocol configuration

JOnAS can be configured 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:

```
carol.protocols=jrmp,cmi
carol.jrmp.url=rmi://localhost:1099
carol.cmi.url=cmi://localhost:2002
```

Caution

When iiop is used in a multiprotocol configuration, it must appear at the first position in the protocol list.

2.3. Configuring the logging System

Monolog [http://monolog.objectweb.org/doc/index.html] is the Objectweb solution for logging. It is not only a new logging implementation but can be seen as a bridge between different logging implementations. A library that uses the Monolog API can be used with any logging implementation at runtime.

Furthermore some components of JOnAS like CAROL, JOTM, Tomcat etc... doesn't use the Monolog API but Jakarta commons loggins or log4j or other implementation. However all these components will be configured via the JOnAS Monolog configuration file.

2.3.1. Monolog

JOnAS Monolog configuration files are:

- `$JONAS_BASE/conf/trace.properties` [4]

  which is the server side Monolog configuration file

- `$JONAS_BASE/conf/traceclient.properties`

  which is used for a client in a client container.

Configuring trace messages inside JOnAS can be done in two ways:

1. Changing the `trace.properties` file to configure the traces statically, before the JOnAS Server is run
2. Using the jonas admin command or the JonasAdmin administration tool to configure the traces dynamically while the JOnAS Server is running. In this case the modification are not persistent (`trace.properties` file is not modified).

2.3.2. trace.properties syntax

Applications make logging calls on *logger* objects.Loggers are organized in a hierarchical namespace and child *loggers* may inherit some logging properties from their parents in the namespace. *Loggers* allocates messages and passes them to *handler* for output; they uses logging *levels* in order to decide if they are interested in by a particular message.

In `trace.properties` it is possible to define *handlers*, *loggers*, *levels*:

- handlers

  A handler represents an output, is identified by its name, has a type, and has some additional properties. By default three handlers are used:

  - **tty** is basic standard output on a console
  - **logf** is a handler for printing messages on a file
  - **mesonly** handler used by generation tools for traces without header

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

  The handler properties are the following:

  - type: is the type of the handler that may be:
    - Console : Log stream ends inside `System.out` or `System.err`
    - File : Log stream is directed into a file
    - Rollingfile : A file set is used to roll the logs
    - JMX : Logging actions are send to the JMX notification system

- pattern:is the message format. A pattern can be composed of elements. An element is prefixed by the % character.The possible items:
  - **%h**: the thread name
  - **%O{1}** : the Class name (basename only)
  - **%M** the method name
  - **%L** the line number
  - **%d** the date
  - **%l** the level
  - **%m** the message itself
  - **%n** a new line
- output: is the output filename.

  If *automatic*[5] is used, JOnAS will replace this tag with a file pointing to `$JONAS_BASE/logs/<jonas_name_server>-<timestamp>.log`

  Switch is used for logging either on `System.out` or `System.err` depending on the level of the log
- fileNumber: is the number of file to use (for RollingFile)
- maxSize:is the maximal size of the file (for Rolling file)

- loggers

  Loggers are identified by names that are structured as a tree. The root of the tree is named *root.* Each logger 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 (see below). By default loggers inherit their level from their parents.

  By default handlers assigned to the parent logger are automatically assign to child loggers. Setting 'additivity' to false inform the system that the logger will use only its own set of handlers.[6]

- levels

  the trace levels are the following:
  - ERROR errors. Should always be printed.
  - WARN warning. Should be printed.
  - INFO informative messages.
  - DEBUG debug messages. Should be printed only for debugging.

2.3.3. default trace.properties file

```
log.config.classname org.objectweb.util.monolog.wrapper.javaLog.LoggerFactory  1

handler.tty.type Console 2
```

```
handler.tty.output Switch 3
handler.tty.pattern  %d : %O{1}.%M : %m%n 4

handler.logf.type File 5
handler.logf.output automatic 6
handler.logf.pattern  %d : %l : %h : %O{1}.%M :    %m%n

logger.root.handler.0 tty 7
logger.root.handler.1 logf 8

logger.root.level INFO 9
logger.org.objectweb.level INFO

#logger.org.objectweb.jonas_ejb.level DEBUG 10

handler.mesonly.type Console 11
handler.mesonly.output Switch
handler.mesonly.pattern  %m%n

logger.org.objectweb.jonas.genic.handler.0 mesonly 12
logger.org.objectweb.jonas.genic.additivity false 13

[...]
```

1 Definition of the wrapper to use: here the java logging API wrapper.
2 Definition of the console handler tty
2 Switch means that the logs will be on System.out or System.err depending of the level of the log.
4 Definition of the message format. here it contains the date followed by ':' the basename of the class followed by '.' the method name followed by ':' the message itself terminated by newline.
5 Definition of the file handler logf
6 Logs are in a file whose name is `$JONAS_BASE/logs/<jonas_name_server>-<timestamp>.log`
7 Definition of the root logger. It uses handler tty
8 Definition of the root logger: It uses also handler logf
9 Definition of the root logger: level INFO is used for all child loggers if there is no overriden definition
10 This line must be uncommented for setting DEBUG level for the logger used in the jonas_ejb module
11 Definition of the console handler mesonly used by GenIC tool which want to log messages without headers
12 Definition of the handler used by the logger org.objectweb.jonas.genic
13 This logger wants to use its own handler.

2.3.4. Tips for setting loggers for JOnAS

When a problem occurs it may be worth to set some debugging traces in the JOnAS server. It is not easy to know which logger to set to obtain the pertinent traces that may help the debbugging process.

The `trace.properties` file contains several commented lines prepared to set loggers in DEBUG level.

Usually the name of loggers are related to the java package name in which it is used.

- To set debug traces of the EJB container uncomment one or more lines related to logger org.objectweb.jonas_ejb for example:

```
logger.org.objectweb.jonas_ejb.interp.level DEBUG
logger.org.objectweb.jonas_ejb.synchro.level DEBUG
logger.org.objectweb.jonas_ejb.tx.level DEBUG
```

- To set traces related to resource adapters:

```
logger.org.objectweb.jonas.jca.level DEBUG
logger.org.objectweb.jonas.jca.pool.level DEBUG
```

- To set traces into the CAROL library::

```
logger.org.objectweb.carol.level DEBUG
```

- To set traces in JORAM:

```
logger.fr.dyade.aaa.level DEBUG  (for the MOM)

# for the JORAM resource adapter:
logger.org.objectweb.joram.client.jms.Client.level DEBUG
logger.org.objectweb.joram.client.connector.Adapter.level DEBUG
```

- To set traces in Tomcat:
  - for all web application :

```
logger.org.apache.catalina.core.ContainerBase.[jonas].[localhost].level DEBUG
```

    jonas is the attribute name of the element Engine in $JONAS_BASE/conf/server.xml

    localhost is the attribute name of the element Host in $JONAS_BASE/conf/server.xml

  - for a particular web application :

```
logger.org.apache.catalina.core.containerBase.[jonas].[localhost].[jonasAdmin].level DEBUG
```

    jonas is the attribute name of the element Engine in $JONAS_BASE/conf/server.xml

    localhost is the attribute name of the element host in $JONAS_BASE/conf/server.xml

    jonasAdmin is the name of the web application

Note

the attributes debug in elements of $JONAS_BASE/conf/server.xml are not used anymore in Tomcat.

- There are a lot of traces possible for management,discovery,jtm,clustering,mail, ear,...

2.3.5. Logging with particular log systems

2.3.5.1. java logging API

If Monolog is configured to use the JDK logger it will replace the JDK logger implementation with its own implementation and so all JDK logs are intercepted by Monolog. By default Monolog is configured to use the JDK logger.

2.3.5.2. Jakarta commons loggin

There is no special configuration file for Jakarta commons login.If it is used on top of the java logging API it is the same case than the previous section.

2.3.5.3. log4j

JOnAS don't provide the corresponding `jar` file so, log4j must be packaged (`.jar` file and `log4j.properties`) in any application that want to use it. The `log4j.properties` file must be configured correctly.

If log4j is used by several applications it is possible to centralize the log4j configuration by putting log4j.properties in `$JONAS_BASE/conf` and log4j jar file in `$JONAS_BASE/lib/commons`.

2.4. Configuring JOnAS Services

2.4.1. registry service

This service is used for accessing the RMI registry, CMI registry, or the CosNaming (rmi/iiop), depending on the configuration of communication protocols specified in `carol.properties`, refer to Section 2.2, “Configuring the communication protocol and JNDI” .

Here is the part of `jonas.properties` file concerning the **registry** service.

```
###################### JOnAS Registry service configuration
#
#  Set the name of the implementation class of the Registry service
jonas.service.registry.class org.objectweb.jonas.registry.RegistryServiceImpl

#  Set the Registry launching mode
#  If set to 'automatic', the registry is launched in the same JVM as Application Server,
#                         if it's not already started.
#  If set to 'collocated', the registry is launched in the same JVM as Application Server
#  If set to 'remote', the registry has to be launched before in a separate JVM
jonas.service.registry.mode    collocated
```

2.4.2. JMX service

The **JMX** service is automatically started in order to administrate or instrument the JOnAS server. It uses the JMX extensions provided by the current Java platform, if the platform provides none then MX4J is used.

Starting from JOnAS 4.9, the JMX service can be started in **secured** or **non-secured** mode:

- In non-secured mode, the JOnAS server accepts JMX connections directly, without requiring the client to provide any credentials (no user names or passwords). This implies that any person that has access to the JOnAS server's JMX port (by default, its RMI port) can also take any action on the server (including remote code execution).
- In secured mode, any client that connects to the JMX service must provide a valid user name and password.
  - When connecting, the client provides a user name and password by setting the **JMXConnector.CREDENTIALS** key of the properties of the connection (**env** variable of the **JMXConnector.connect** method).

    This user name and password is always directly transmitted to the JOnAS server the client is connecting to, and it's that server's decision whether:

    - The user name and password is considered as being valid, therefore the connection will be accepted. This phase is called **Authentication** .
    - That user has the right of accessing a certain method of a certain instance. This phase is called **Authorization**.
  - For authentication, you can use any JAAS LoginModule of the JMX extensions provided by your platform.

    For authorization, you can use any Security Manager provided by your platform.

Here is the part of `jonas.properties` concerning the **JMX**service:

```
###################### JOnAS JMX service configuration
#
#  Set the name of the implementation class of the JMX service
jonas.service.jmx.class                       org.objectweb.jonas.jmx.JmxServiceImpl

#  Set to true if the JMXRemote interface should require the client to provide
#  authentication information. That information is provided when establishing
#  the JMX connection, using the JMXConnector.CREDENTIALS key.
#
#  Note that if you enable JMX security for a server, all clients (including
#  any administration tool such as the domain master) connecting to this
#  instance via JMX must provide a valid user name and password.
jonas.service.jmx.secured                     false

#  If jonas.service.jmx.secured is set to true, defines the authentication
#  method and the method's parameter. For example, to use file-based
#  authentication using the conf/jmx.passwords file, define:
#      jonas.service.jmx.authentication.method    jmx.remote.x.password.file
#      jonas.service.jmx.authentication.parameter conf/jmx.passwords
#  You are free to use the authentication provider you wish.
jonas.service.jmx.authentication.method       jmx.remote.x.password.file
jonas.service.jmx.authentication.parameter    conf/jmx.passwords
#  You may for example choose to use JAAS LoginModule for authentication.
#  In this case define the used configuration in the JAAS configuration file
#  using the jonas.service.jmx.authentication.parameter:
#      jonas.service.jmx.authentication.method      jmx.remote.x.login.config
#      jonas.service.jmx.authentication.parameter   jaas-jmx

#  If jonas.service.jmx.secured is set to true, defines the authorization
#  method and the method's parameter. For example, to use file-based
#  authorization using the conf/jmx.access file, define:
#      jonas.service.jmx.authorization.method    jmx.remote.x.access.file
#      jonas.service.jmx.authorization.parameter conf/jmx.access
#  You are free to use the authorization provider you wish.
jonas.service.jmx.authorization.method        jmx.remote.x.access.file
jonas.service.jmx.authorization.parameter     conf/jmx.access
#  You may for example choose to use role-based authorization manager
#  configured using conf/jmx.rolebased.access file. In this case, define:
#      jonas.service.jmx.authorization.method          jmx.remote.x.access.rolebased.file
#      jonas.service.jmx.authorization.parameter       conf/jmx.rolebased.access
```

2.4.3. ejb service

This service provides containers for EJB components.

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

- The `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.

  Exemple:

  ```
  jonas.service.ejb.descriptors  sb.jar,eb.jar
  ```

  In this example the **ejb** service will create two containers from the `ejb-jar` files named `sb.jar` and `eb.jar`. These files will be searched for in the `$JONAS_BASE/ejbjars` directory.

- The `ejb-jar` file is located in `$JONAS_BASE/ejbjars/autoload`[7] directory. In this case the container will be created at server start-up time.

- The `ejb-jar` file is packaged inside an ear file as a component of a Java EE application. The container will be created when the Java EE application will be deployed via the **ear** service.

- EJB containers may be dynamically created from ejb-jar files using the JonasAdmin tool.

- EJB containers may be dynamically created from ejb-jar files using the command jonas admin:

```
jonas admin -a sb.jar
```

Here is the part of `jonas.properties` concerning the **ejb** service:

```
###################### JOnAS EJB Container service configuration
#
#  Set the name of the implementation class of the ejb service
jonas.service.ejb.class  org.objectweb.jonas.container.EJBServiceImpl

#  Set the list of directories that contains ejbjars that must be deployed by
#  the JOnAS Server at launch time.
#  Here should be given a coma-separated list of directories.
#  If the directory has a relative path, this path is relative from where the
#  Application Server is launched.
#  If the directory is not found it will be searched in JONAS_BASE/ejbjars/
#  directory.
jonas.service.ejb.autoloaddir    autoload

#  Set the list of ejbjars that must be deployed by the JOnAS Server at launch time.
#  Here should be given a coma-separated list of ejb-jar files names or standard XML deployment
#  descriptors files names.
#  If the file name has a relative path, this path is relative from where the
#  Application Server is launched.
jonas.service.ejb.descriptors

#  Set the XML deployment descriptors parsing mode (with or without validation)
jonas.service.ejb.parsingwithvalidation    true

#  Set the size of the worker thread pool
jonas.service.ejb.minworkthreads   3

#  Set the maximun size of the worker thread pool
jonas.service.ejb.maxworkthreads   80

#  Set the max # of seconds that a thread will wait for work
#  This is used to shrink the worker thread pool back to minimum
jonas.service.ejb.threadwaittimeout 60


# If enabled, the GenIC tool will be called if :
#  - JOnAS version of the ejb-jar is not the same version than the running JOnAS instance
#  - Stubs/Skels stored in the ejb-jar are not the same than the JOnAS current protocols.
# By default, this is enabled
jonas.service.ejb.auto-genic true

# Arguments for the auto GenIC
#jonas.service.ejb.auto-genic.args.0 -invokecmd
#jonas.service.ejb.auto-genic.args.1 -verbose
```

For customizing the **ejb** service it is possible to:

- Change the directory name for automatic deployment: property jonas.service.ejb.autoloaddir
- Give a list of containers to create: property jonas.service.ejb.descriptors
- Set or not the XML validation at the deployment descriptor parsing time
- Set or not the automatic generation via the GenIC tool
- Customize the worker threads (see below):

  This is only useful when the **jms** service is used. The **jms** service relies on the **ejb** thread pool (*Work-Thread* pool) to run message-driven bean `onMessage()` methods. Thus, it could be important to configure this thread pool. Min and max values can be set for the *WorkThread* pool :

  - jonas.service.ejb.minworkthreads defines the number of threads that will be created when the pool is started. (default value = 3)

- jonas.service.ejb.maxworkthreads defines the maximum number of these threads that can be created at any time. (default value = 60)

- jonas.service.ejb.threadwaittimeout defines the max number of seconds a thread will wait for work. After the timeout, the thread is removed. This is used to shrink the worker thread pool back to minimum. (default value = 60)

Since this pool is used mainly for message-driven beans, there is a link between the value to choose, and the values chosen for max-cache-size values of the message-driven beans. There is no need to set more threads than the sum of all the max-cache-size values of all the message-driven beans. Setting a value lower than this could lead to deadlock in some situations, for example, if service **jms** waits for a thread to run a message that would free other threads blocked, and that the maxworkthreads value has been reached.

2.4.4. web service

This service provides containers for the web components used by the Java EE 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 in order to use the JonasAdmin tool. 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.

A web container can be created from a file using one of the following possibilities:

- The `war` file name is listed in the jonas.service.web.descriptors property of the `jonas.properties` file.

  If the file name does not contain an absolute path name, it should be located in the $JONAS_BASE/webapps/ directory. The container is created when the JOnAS server starts.

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

  In this example the **web** service will create a container from the war file named `Bank.war`. It will search for this file in the `$JONAS_BASE/webapps` directory.

- The `war` file is located in `$JONAS_BASE/webapps/autoload`[8] directory. In this case the container will be created at server start-up time.

- The `war` file is packaged inside an `ear` file as a component of a Java EE application. The container will be created when the Java EE application will be deployed via the **ear** service.

- Web containers may be dynamically created from war files using the JonasAdmin tool.

- Web containers may be dynamically created from war files using the command **jonas admin**:

```
jonas admin -a Bank.war
```

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: `$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.

Here is the part of `jonas.properties` concerning the **web** service:

```
###################### JOnAS Web container service configuration
#
#  Set the name of the implementation class of the web container service.
jonas.service.web.class    org.objectweb.jonas.web.wrapper.catalina55.CatalinaJWebContainerServiceWrapper
#jonas.service.web.class    org.objectweb.jonas.web.jetty50.JettyJWebContainerServiceImpl

#  Set the list of directories that contains wars that must be deployed by
#  the JOnAS Server at launch time.
#  Here should be given a coma-separated list of directories.
#  If the directory has a relative path, this path is relative from where the
#  Application Server is launched.
#  If the directory is not found it will be searched in JONAS_BASE/webapps/
#  directory.
jonas.service.web.autoloaddir    autoload

#  Set the list of wars that must be depoyed by the JOnAS Server at launch time.
#  Here should be given a coma-separated list of war files names.
#  If the file name has a relative path, this path is relative from where the
#  Application Server is launched.
jonas.service.web.descriptors

#  Set the XML deployment descriptors parsing mode for the WEB container
#  service (with or without validation).
jonas.service.web.parsingwithvalidation    true
```

For customizing the **web** service It is possible to

- Change the directory name for automatic deployment: jonas.service.web.autoloaddir property
- Give a list of containers to create: jonas.service.web.descriptors property
- Set or not the XML validation at the deployment descriptor parsing time

2.4.5. resource service

The **resource** service must be started when Java EE 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 application components. The **resource** service implements the Java EE Connector Architecture 1.5[9].

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.`

  Exemple:

[9]There is no real acronym for this specification JCA was the acronym for Java Cryptography Architecture" . In the rest of this document we will use J2CA

```
jonas.service.resource.resources  MyEIS.rar
```

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 in J2EE Connector Programmer's Guide [PG_Connector.html#PG_Connector-Descriptor] for additional information.

Here is the part of `jonas.properties` related to **resource** service:

```
###################### JOnAS J2CA resource service configuration
#
#  Set the name of the implementation class of the J2CA resource service
jonas.service.resource.class org.objectweb.jonas.resource.ResourceServiceImpl

#  Set the list of directories that contains rars that must be deployed by
#  the JOnAS Server at launch time.
#  Here should be given a comma-separated list of directories.
#  If the directory has a relative path, this path is relative from where the
#  Application Server is launched.
#  If the directory is not found it will be searched in JONAS_BASE/rars/
#  directory.
jonas.service.resource.autoloaddir    autoload

#  Set the XML connector deployment descriptors parsing mode (with or without validation)
jonas.service.resource.parsingwithvalidation    true

#  Set the min size of the worker thread pool used for all J2CA 1.5 Resource Adapters deployed
jonas.service.resource.minworkthreads  5

#  Set the max size of the worker thread pool used for all J2CA 1.5 Resource Adapters deployed
jonas.service.resource.maxworkthreads  80

#  Set the max # of seconds that a thread will wait for work
#  This is used to shrink the worker thread pool back to minimum
jonas.service.resource.threadwaittimeout 60

#  Set the max # of seconds of execution time for a work object
#  This functionality may not be supported by all Resource Adapter
jonas.service.resource.execworktimeout 0

#  Set the list of Resource Adapter to be used.
#  This enables the JOnAS server to configure the resource adapter and register it into JNDI.
#  This property is set with a coma-separated list of rar file names
#   (with/without the '.rar' suffix).
#  Ex: XXXX,YYYY (while the rar file names are XXXX.rar and YYYY.rar)
jonas.service.resource.resources
```

For customizing the **resource** service it is possible to:

- Change the directory name for automatic deployment: property jonas.service.resource.autoloaddir
- Give a list of resourceadapter to deploy: property jonas.service.resource.resources
- Set or not the XML validation at the deplyment descriptor parsing time
- Customize the worker thread pool used for all J2CA 1.5 Resource Adapters deployed
  - jonas.service.resource.minworkthreads defines the number of threads that will be created when the pool is started. (default value = 5)

- jonas.service.resource.maxworkthreads defines the maximum number of these threads that can be created at any time. (default value = 80)
- jonas.service.resource.threadwaittimeout defines the max number of seconds a thread will wait for work. After the timeout, the thread is removed. This is used to shrink the worker thread pool back to minimum. (default value = 60)
- jonas.service.resource.execworktimeout defines the max time (in second) for execution of a work thread (default value = 0 i.e infinite).

resource service is mainly used in JOnAS for accessing databases via a JDBC resource adapter (in this case it replace **dbm** service) and for providing JMS facilities (and it replace **jms** service).

JOnAS provides several JDBC resource adapters and a JMS resource adapter on top of JORAM [http://joram.objectweb.org/] More information about configuring resource adapters can be found in Section 2.6, “Configuring JDBC Resource Adapters”

2.4.6. ear service

The **ear** service allows deployment of complete Java EE applications (including `ejb-jar`, `war` and rar files packed in an `ear` file). This service is based on the **web** service for deploying the included `wars`, the **ejb** service for deploying the EJB containers for the included `ejb-jars` and the **resource** service for deploying the included `rars`.

A J2EE application can be deployed by the **ear** service using one of the following possibilities:

- The `ear` file name is listed in the jonas.service.ear.descriptors property of the `jonas.properties` file.

  If the file name does not contain an absolute path name, it should be located in the `$JONAS_BASE/apps/` directory. The application is deployed when the JOnAS server starts.

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

- The `ear` file is located in `$JONAS_BASE/apps/autoload`[10] directory. In this case the application will be deployed at server start-up time.
- Java EE application may be dynamically deployed from `ear` files using the JonasAdmin tool.
- Java EE application may be dynamically deployed from `ear` files using the command jonas admin:

```
jonas admin -a Bank.ear
```

Here is the part of `jonas.properties` concerning the **ear** service:

```
###################### JOnAS EAR service configuration
#
#  Set the name of the implementation class of the ear service.
jonas.service.ear.class  org.objectweb.jonas.ear.EarServiceImpl

#  Set the list of directories that contains ears that must be deployed by
#  the JOnAS Server at launch time.
#  Here should be given a coma-separated list of directories.
#  If the directory has a relative path, this path is relative from where the
#  Application Server is launched.
#  If the directory is not found it will be searched in JONAS_BASE/apps/
#  directory.
jonas.service.ear.autoloaddir    autoload

#  Set the list of ears that must be depoyed by the JOnAS Server at launch time.
#  Here should be given a coma-separated list of ear files names.
```

```
#  If the file name has a relative path, this path is relative from where the
#  Application Server is launched.
jonas.service.ear.descriptors

#  Set the XML deployment descriptors parsing mode for the EAR service
#  (with or without validation).
jonas.service.ear.parsingwithvalidation    true
```

For customizing the **ear** service It is possible to :

- Change the directory name for automatic deployment: jonas.service.ear.autoloaddir property
- Give a list of J2EE application to deploy : jonas.service.ear.descriptors property
- Set or not the XML validation at the deployment descriptor parsing time

2.4.7. jtm service

The **jtm** service is used by **ejb** service in order to provide transaction management for EJB components as defined in the deployment descriptor. The **jtm** 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 jtm 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.

By default JOTM [http://jotm.objectweb.org/] is the Transaction manager used.

Here is the part of `jonas.properties` concerning the **jtm** service:

```
###################### JOnAS JTM Transaction service configuration
#

#  Set the name of the implementation class of the jtm service
jonas.service.jtm.class  org.objectweb.jonas.jtm.TransactionServiceImpl

#  Set the Transaction Manager launching mode.
#  If set to 'true', TM is remote: TM must be already launched in an other JVM.
#  If set to 'false', TM is local: TM is going to run into the same JVM
#  than the jonas Server.
jonas.service.jtm.remote    false

#  Set the default transaction timeout, in seconds.
jonas.service.jtm.timeout    60
```

For customizing the **jtm** service It is possible to

- Indicate if the Transaction Manager used in this instance is collocated or remote: jonas.service.jtm.remote property
- Setting the value of the transaction time-out, in seconds: jonas.service.jtm.timeout property

2.4.8. ws service

The **ws** service use the Axis implementation.

Here is the part of `jonas.properties` concerning the **ws** service:

```
###################### JOnAS WebServices service configuration
#
#  Set the name of the implementation class of the WebServices service.
jonas.service.ws.class    org.objectweb.jonas.ws.axis.AxisWSServiceImpl

# Set the JServiceFactory to use
jonas.service.ws.factory.class org.objectweb.jonas.ws.axis.JAxisServiceFactory

#  Set the XML deployment descriptors parsing mode for the WebServices
```

```
#  service (with or without validation).
jonas.service.ws.parsingwithvalidation    true

#  Set the WSDL Handler list for WSDL publication
#  A minimum of 1 WSDLHandler is required !
#  This property is set with a coma-separated list of WSDLHandler properties
#  file names (without the '.properties' suffix).
#  Ex: file1,uddi (while the properties file names are
#                  file1.properties and uddi.properties)
jonas.service.ws.wsdlhandlers file1

# Set the Generator to use with wsgen
jonas.service.ws.wsgen.generator.factory     org.objectweb.jonas_ws.wsgen.generator.ews.EWSGeneratorFactory

# Set the prefix that will be used to compute URL endpoints for web services
#jonas.service.ws.url-prefix http://www.mydomain.com:8888

# Set automatic WsGen mode on/off
# If set to 'true', WsGen will automatically be applied to all deployed archives (EjbJars, Webapps, Applications)
# default to 'true'
#jonas.service.ws.auto-wsgen.engaged false
```

or customizing the **ws** service It is possible to :

- Set or not the XML validation at the deployment descriptor parsing time: property jonas.service.ws.parsingwithvalidation
- Choose one or more WSDL Handler(s): property jonas.service.ws.wsdlhandlers

  WSDL Handlers are used to locate and publish all WSDL documents. WSDL handlers are configure in WSDLHandler properties files in which it is possible to:

  - set the directory where WSDLs will be copied: property jonas.service.publish.file.directory
  - set the encoding of the file: property jonas.service.publish.file.encoding (default=UTF-8)
- Enforce the URL to be used for the deployed endpoints: property:jonas.service.ws.url-prefix

  This is interesting when there is a cluster of JOnAS instances and an HTTP frontend for load balancing. For example You want all your web services endpoint to use the http://www.mywebserver.com URL instead of the usual http://localhost:9000 (that has a meaning only at local level).
- Enable or not to run the WSGen tool on ejb-jar,war,ear,jar client at deplyoment time.

More information on How to Develop a J2EE-Compliant Web Service (endpoint + client) [http://jonas.objectweb.org/current/doc/howto/J2EEWebServicesDevelopment.html] and Web Services with JOnAS [JOnASWebServices.html].

2.4.9. 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.The default implementation of the **mail** service rely on the GNUMail implementation of these 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 according 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.

Here is the part of `jonas.properties` concerning the **mail** service:

```
###################### JOnAS Mail service configuration
#
#  Set the name of the implementation class of the mail service
jonas.service.mail.class org.objectweb.jonas.mail.MailServiceImpl

#  Set the jonas mail factories.
#  This property is set with a coma-separated list of MailFactory properties
#  file names (without the '.properties' suffix).
#  Ex: MailSession1,MailMimePartDS1 (while the properties file names are
#                  MailSession1.properties and MailMimePartDS1.properties)
jonas.service.mail.factories
```

Mail factory objects created by JOnAS must be given a name. In the mailsb example (see $JONAS_ROOT/examples/src/mailsb) , 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).

For this example jonas.service.mail.factories property must be set to:

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

2.4.9.1. Configuring Session mail factory

A template `MailSession1.properties` file is supplied in `$JONAS_BASE/conf.` It contains two mandatory properties :

```
#Factory Name/Type
mail.factory.name mailSession_1
mail.factory.type javax.mail.Session
```

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.

2.4.9.2. Configuring MimePartDataSource mail factory

A template `MimePartDS1.properties` is supplied in `$JONAS_BASE/conf`. It contains two mandatory properties :

```
#Factory Name/Type
mail.factory.name mailMimePartDS_1
mail.factory.type javax.mail.internet.MimePartDataSource
```

The JNDI name of the mail factory object is mailMimePartDS_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.

2.4.9.3. Configuring a mail factory

Here are the possible properties

- Required properties:

Property name	Description
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

Property name	Description
mail.authentication.username	Set the username for the authentication.
mail.authentication.password	Set the password for the authentication.

- Optional properties: javax.mail.Session.properties (refer to JavaMail documentation for more information)

Property name	Description
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.

- Optional properties:MimePartDataSource properties (Only used if mail.factory.type is javax.mail.internet.MimePartDataSource)

Property name	Description
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
mail.bcc	Set the list of Blind Carbon Copy recipients ("bcc") of the message.
mail.subject	Set the subject of the message.

2.4.10. security service

Here is the part of `jonas.properties` related to **security** service:

```
###################### JOnAS SECURITY service configuration
#
#  Set the name of the implementation class of the security service
jonas.service.security.class org.objectweb.jonas.security.JonasSecurityServiceImpl

# Realm used for CsiV2 authentication
jonas.service.security.csiv2.realm    memrlm_1

# Realm used for Web Service authentication
jonas.service.security.ws.realm    memrlm_1

# Registration of realm resources into JNDI
# Disable by default so configuration is not available with clients
jonas.service.security.realm.jndi.registration[11]    false
```

In fact properties jonas.service.security.csiv2.realm and jonas.service.security.ws.realm are only useful for users that use security on top of rmi/iiop or on top of web services . in these case with `memrlm_1` it is possible to make a link to the memomyrealm named `memrlm_1` in the $JONAS_BASE/conf/jonas-realm.xml file and retrieve users name and roles.

Don't forget that for using security the global property jonas.security.propagation to true and that an important property related to security is jonas.security.manager see Section 2.1.4.1, “Global properties of jonas.properties”

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 Section 2.5, “Configuring Security” for a complete description of security configuration.

2.4.11. db service

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

By default the database used is HSQLDB. [http://hsqldb.org/]

Here is the part of `jonas.properties` related to **db** service:

```
###################### JOnAS DB service configuration
#
#  Set the name of the implementation class of the db service (hsql for example)
jonas.service.db.class    org.objectweb.jonas.db.hsqldb.HsqlDBServiceImpl
jonas.service.db.port    9001
jonas.service.db.dbname    db_jonas
jonas.service.db.user1    jonas:jonas
#jonas.service.db.user2    login:password
```

[11] is deprecated and only kept for compatibility with older version.

Here it is possible to customize :

- the listening port
- the database name
- user password must be declared as:

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

The database may be used by Java EE component via JDBC resource adapters or via the **dbm** service. For the former case the same information (listening port, database name, login,password) must appear in the JOnAS connector deployment descriptor, in the latter they appear in the `$JONAS_BASE/conf/HSQLDB1.properties`. So, if these previous properties must be changed in jonas.properties, they must be also changed in these files.

Via the **db** service HSQLDB is configured in a default mode where the tables are not persistent and exist entirely in random access memory.

The **db** service has been provided in the jonas distribution mainly to run easily the JOnAS exemple, without having to set a database first. For most usages, the jonas users should remove it from the list of services and remove also HSQL1 from jonas.service.dbm.datasources property in `$JONAS_BASE/conf/jonas.properties` file.

For users that choose HSQLDB as database it is highly recommended to refer to the Hsqldb User Guide [http://hsqldb.org/web/hsqlDocsFrame.html]. It is worth to note that the default configuration file used by HSQLDB server can be found in `$JONAS_BASE/work/hsqldb/jonas/db_jonas.properties` directory.

2.4.12. discovery service

The role of the **discovery** service is to enable domain management. It allows the management of all the servers running in the domain from the common administration point represented by the master.

The discovery service is based on IP multicast. It allows master to detect servers starting and stopping in the domain, and to discover servers there were already running in the domain.

Here is the part of `jonas.properties` related to **discovery** service:

```
###################### JOnAS Discovery service
#
#  Set the name of the implementation class of the discovery service
#  The server is not a discovery master, unless its name is
#  identical to the domain name
#jonas.service.discovery.master = true
#  If this is a master, the discovery client source port may be configured
#  with the property:
jonas.service.discovery.source.port=9888
#
#  Set the name of the implementation class and initialization parameters
jonas.service.discovery.class=org.objectweb.jonas.discovery.DiscoveryServiceImpl
jonas.service.discovery.multicast.address=224.224.224.224
jonas.service.discovery.multicast.port=9080
jonas.service.discovery.ttl=1

#  A multicast greeting message is sent out when discovery service is started.
#  The starting server listens at the port jonas.service.discovery.greeting.port
#  (default 9899) for a response for jonas.service.discovery.greeting.timeout miliseconds
#  (default 1000 ms). If a pre-existing server has the same server name as this one,
#  this server's discovery service will be terminated.

jonas.service.discovery.greeting.port=9899
jonas.service.discovery.greeting.timeout=1000
```

For the **discovery** service it is possible to :

- Indicate if the current instance is a master or a slave. Use property jonas.service.discovery.master. In the case of a master instance, the jonas.service.discovery.source.port proprty must be set. This port is used by the servers to respond to discovery notifications emitted by the master.
- Customize the IP multicast configuration that must be identical for all instances. Use properties:
  - jonas.service.discovery.multicast.address
  - jonas.service.discovery.multicast.port

  beware that multicast adresses must be consequently allocated through the network.
- Customize the time-to-live for packets: use property:
  - jonas.service.discovery.ttl

  it indicates the number of gateway hops for packets.
  - if ttl = 0 the discovery scope is the host (multicast packet aren't routed to network interfaces).
  - if ttl =1 the discovery scope is limited to the subnetworks the host is attached to (multicast packets cross the network interfaces but will be discarded by the next gateway).
  - if ttl =N>1 the discovery packets may cross N-1 gateways (provided that these gateways are configured to propagate multicast packets).
- Customize the greeting mechanism used to enforce servers name unicity in the domain. Use properties:
  - jonas.service.discovery.greeting.port
  - jonas.service.discovery.greeting.timeout

  Note that two servers on the same host must have different values in greeting.port property.

2.4.13. ha service

The **ha** (High Availability) service is required in order to replicate stateful session beans (SFSBs).

The **ha** service uses JGroups as a group communication protocol (GCP).

Here is the part of `jonas.properties` related to **ha** service:

```
###################### JOnAS HA service configuration
#
# Set the name of the implementation class of the HA service.
jonas.service.ha.class    org.objectweb.jonas.ha.HaServiceImpl

# Set the group communication framework to use
jonas.service.ha.gcl jgroups

# Set the JGroups configuration file name
jonas.service.ha.jgroups.conf jgroups-ha.xml

# Set the JGroups group name
jonas.service.ha.jgroups.groupname jonas-rep

# Set the SFSB backup info timeout. The info stored in the backup node is removed when the timer expires.
jonas.service.ha.timeout 600

# Set the datasource for the tx table
jonas.service.ha.datasource jdbc_1
```

For the **ha** service it is possible to:

- set the name of the JGroups configuration file: property jonas.service.ha.jgroups.conf
- set the name of the JGroups group: property jonas.service.ha.jgroups.groupname
- set the period of time(in seconds) the system waits before cleaning useless repliaction information: property jonas.service.ha.timeout
- set the JNDI name of the datasource corresponding to the database where is located the transaction table used by the replication mechanism: property jonas.service.ha.datasource

2.4.14. dbm service

The **dbm** service (database manager service) allow access to one or more relational databases. It 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.

Caution

the recommended way to access to databases is to use the **resource** service deploying JDBC resource adapter

The **dbm** service 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. When it is necessary to use a JDBC2-XA-compliant driver access to the databases must be done via a JDBC resource adapter XA compliant (more information can be found in Section 2.6, “Configuring JDBC Resource Adapters”

Here is the part of `jonas.properties` related to **dbm** service:

```
###################### JOnAS DBM Database service configuration
#
#  Set the name of the implementation class of the dbm service
jonas.service.dbm.class  org.objectweb.jonas.dbm.DataBaseServiceImpl

#  Set the jonas DataSources. This enables the JOnAS server to load
#  the data dources, to load related jdbc drivers, and to register the data
#  sources into JNDI.
#  This property is set with a coma-separated list of Datasource properties
#  file names (without the '.properties' suffix).
#  Ex: Oracle1,InstantDB1 (while the Datasources properties file names are
#                          Oracle1.properties and InstantDB1.properties)
jonas.service.dbm.datasources    HSQL1
```

For the **dbm** service it is possible to:

- set a list of datasource names via property jonas.service.dbm.datasources.

  for each name XX appearing in this list a `XX.properties` file must exist in `$JONAS_BASE/conf`

Access to a particular database via **dbm** service is configured in datasource.properties files that must be located in `$JONAS_BASE/conf`.

2.4.14.1. Datasource.properties files

In the JOnAS distribution several templates of datasource.properties files are provided one for Oracle, PostgreSQL, Sybase, DB2, MySQL, HSQLDB, InterBase, FirebirdSQL, Mckoi SQL, InstantDB) respectively in *Oracle1.properties*, *PostgreSQL1.properties* etc...

A complete description of the datasource.properies file can be found inSection 2.8, “Configuring JDBC DataSources”

2.4.15. jms service

jms service is a service that can be used for application that use the Java Message Service API , or that use message-driven beans (with some restrictions: it does not allow deployment of 2.1 message-driven beans).

Caution

The recommended way is to use a JMS provider through the deployment of a resource adapter and to use resource service instead (seeSection 2.7, “Configuring JMS Resource Adapters”)

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 JOnAS server itself.

Here is the part of `jonas.properties` related to **jms** service:

```
###################### JOnAS JMS service configuration
#
#  Set the name of the implementation class of the jms service
jonas.service.jms.class  org.objectweb.jonas.jms.JmsServiceImpl

#  Indicates the Jms service must be started with this class for administering the mom
jonas.service.jms.mom    org.objectweb.jonas_jms.JmsAdminForJoram

#  Set the Jms Server launching mode
#  If set to 'true'  it is launched in the same JVM as Application Server
#  If set to 'false' Jms Server is launched in a separate JVM
jonas.service.jms.collocated    true

#  Set to the url connexion when the Jms server is not collocated
#jonas.service.jms.url    joram://localhost:16010

# Set the list of administered objects topics to be created at Application Server launching time
# Note : When using resource service (default configuration), topics should go in joram-admin.cfg file
jonas.service.jms.topics    sampleTopic

# Set the list of administered objecst queuess to be created at Application Server launching time
# Note : When using resource service (default configuration), queues should go in joram-admin.cfg file
jonas.service.jms.queues    sampleQueue
```

For the **jms** service it is possible to:

- indicate if the JMS server used is collocated or not: property jonas.service.jms.collocated
  - if not collocated set the url connexion to the remote JMS server:jonas.service.jms.url
- set a list of administered objects queues or topics that must be created(if needed) at launching time properties: jonas.service.jms.topics, jonas.service.jms.queues
- indicate which class must be used to perform administrative operations: property jonas.service.jms.mom.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`

The **jms** service relies on the ejb WorkThread pool to run the MDB onMessage() methods. Here is described how to configure this thread pool.

2.4.16. thread service

The **thread** service is mandatory to use the Thread Management Framework

it must be started just after the jmx service

Here is the part of `jonas.properties` related to **jms** service:

```
###################### JOnAS Thread Service
#
# Set the name of the implementation class of the thread service
#
jonas.service.thread.class       org.objectweb.area.jonas.AreaService
jonas.service.thread.file        jonas_areas.xml
jonas.service.thread.ejbareaname EJB
```

For the **thread** service it is possible to:

- choose the main configuration file for the Thread Management Framework: property jonas.service.thread.file
- choose the name of the area for the EJB

See here [TMF.html] for a complete description of the Thread Management Framework.

2.5. Configuring Security

The **security** service is used by the **ejb**, **web**, **ws** services to provide security for J2EE components. The **ejb** service provides security in two forms: declarative security and programmatic security that is described in the EJB Programmer's Guide: Security Management [PG_Security.html#PG_Security] .

The **security** service exploits security roles and method permissions located in the J2EE deployment descriptors.

A main concept in security is *Authentication* which is the mechanism telling the container the identity of the user making the current request.

A caller is a client that may be a servlet client or a container client. Usually a client proves its identity by a couple user/password or a certificate (*credential*). Once the identification is correct JOnAS must build a security context that will be propagated with requests and be used by the container to verify that the user exists and has permissions sufficient to make the request.

JAAS is a standard framework for authenticating users. It defines configuration files (`jaas.config`) and interfaces like the `LoginModule` interface that may be used in JOnAS to perform authentication tasks.

Lightweight authentication mechanism using JACC may be used to authenticate servlet client.

In the Tomcat documentation we can find this definition: “A Realm is a "database" of usernames and passwords that identify valid users of a web application (or set of web applications), plus an enum eration of the list of roles associated with each valid user.”

In both authentication mechanisms the container use a *realm* to verify validity of users. In JOnAS the *realm* may be a database accessed via JDBC (Database realm), a LDAP directory (LDAP realm) or a flat file (Memory realm). The type of realm to use is specified in `$JONAS_BASE/conf/jonas-realm.xml`.

2.5.1. jonas-realm.xml

The file `$JONAS_BASE/conf/jonas-realm.xml` file describes:

- the content of flat file memory realm
- how to access a database realm
- how to access a LDAP realm

2.5.1.1. Memory realm

The *memoryrealm* must be named and defines users, groups and roles in the section <jonas-memoryrealm>

```
<jonas-memoryrealm>
<memoryrealm name="memrlm_1"> 1
<roles>
<role name="jonas-admin" description="JonasAdmin role" /> 2
<role name="tomcat" description="Used in examples" />
</roles>
<groups>
<group name="jonas"
roles="jonas-admin,tomcat,jaas,ws-security" description="All authorization" /> 3
</groups>
<users>
<user name="tomcat" password="tomcat" roles="tomcat,jonas-admin,manager" /> 4
<user name="jetty" password="jetty" roles="jetty" />
<!-- Example of a crypt password : password for jadmin is : jonas -->
<user name="jadmin" password="{MD5}nF3dVBB3NPfRgzWlJFwoaw==" roles="jonas-admin" /> 5
<user name="jps_admin" password="admin" roles="administrator" />
<user name="supplier" password="supplier" roles="administrator" />
<!-- Another crypt example in another format : password is jonas -->
<!-- JonasAdmin uses name="jonas" password="jonas" -->
<user name="jonas" password="SHA:NaLG+uYfgHeqth+qQBlyKr8FCTw=" groups="jonas" /> 6
<user name="principal1" password="password1" roles="role1" />
<user name="principal2" password="password2" roles="role2" />
</users>
</memoryrealm>
</jonas-memoryrealm>
```

1 memoryrealm must be named. This name will be used in the web container configuration file
2 definition of a security role
3 definition of a group of roles
4 definition of a user with non encrypted password and a list of roles
5 definition of a user with encrypted password (format MD5)
6 definition of a user with encrypted password (format SHA)

2.5.1.2. database realm

Users, groups, and roles information are stored in a database; the configuration for accessing the corresponding database is described in the section <jonas-dsrealm>

The configuration requires the name of a datasource, the tables used, and the names of the columns.

```
<jonas-dsrealm>
<dsrealm name="dsrlm_1" 1
dsName="jdbc_1" 2
userTable="realm_users" userTableUsernameCol="user_name" userTablePasswordCol="user_pass" 3
roleTable="realm_roles" roleTableUsernameCol="user_name" roleTableRolenameCol="role_name"> 4
</dsrealm>
</jonas-dsrealm>
```

1 dsrealm must be named
2 JNDI name of the `dataSource` for accessing the database via JDBC

3 defines the name of the user table and the name of the columns for username/password
4 defines the name of the role table and the name of the columns for username/rolename

to use this database a `Datasource` configuration with the right JNDI name for the **dbm** service must be set in the `jonas.properties` file.

2.5.1.3. LDAP realm

Users, groups, and roles information are stored in an LDAP directory. This is described in the section <jonas-ldaprealm>

There are some optional parameters. If they are not specified, some of the parameters are set to a default value. For example if the providerUrl element is not set, the default value is `ldap://localhost:389`. The jonas-realm_1_0.dtd DTD [http://jonas.objectweb.org/dtds/jonas-realm_1_0.dtd]file show the default values.

- minimal example:

```
<jonas-ldaprealm>
<ldaprealm name="ldaprlm_1" 1
baseDN="dc=jonas,dc=objectweb,dc=org" /> 2
</jonas-ldaprealm>
```

1 ldaprealm must be named
2 to access to LDAP server

For this sample, it is assumed that the LDAP server is on the same computer and is on the default port (389).

2.5.2. Servlet Authentication

Depending on the servlet container used, configuration differs.

2.5.2.1. Authentication with User/password and Tomcat 5.5

- Tomcat configuration:

  Tomcat embedded in the JOnAS distribution is configured in `$JONAS_BASE/conf/server.xml` to use the memory realm named `memrlm_1`

```
<Server>
[...]
<Realm className="org.objectweb.jonas.security.realm.web.catalina55.JACC" resourceName="memrlm_1" />
[...]
</Server>
```

  The authentication mechanism implemented by the class `org.objectweb.jonas.security.realm.web.catalina55.JACC` is able to work with database or LDAP realm configured in `jonas-realm.xml.` The value of resourceName attribute identifies the *realm* to be used in `jonas-realm.xml.`

- webapp configuration:

  In the `web.xml` of the web application a *basic autjentication* or a *Form based authentication* may be used

```
<web-app>
<login-config>
<auth-method>BASIC</auth-method>
```

```
<realm-name>Example Basic Authentication Area</realm-name>
</login-config>
</web-app>
```

or

```
<web-app>
<login-config>
<auth-method>FORM</auth-method>
<form-login-config>
<form-login-page>login.jsp</form-login-page>
<form-error-page>error.jsp</form-error-page>
</form-login-config>
</login-config>
</web-app>
```

Like basic authentication, form-based authentication is not secure, since the content of the user dialog is sent as plain text, and the target server is not authenticated.

To overcome this vulnerability the authentication protocol may be run over a SSL session that ensures that all message contents are protected for confidentiality.

2.5.2.2. Authentication with certificate and Tomcat 5.5

In this case, 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(DN).

- jonas-realm configuration:

  The name identifying the person to whom the certificate belongs looks like the following: CN=Someone Unknown, OU=ObjectWeb, O=JOnAS, C=ORG with:

  CN : Common Name

  OU : Organizational Unit

  O : Organization

  C : Country Name

  E : Email Address

  L : Locality

  ST :State or Province Name

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

  In the jonas-realm.xml a user with password looks like:

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

  A certificate-based user must have its DN preceded by the String: ##DN## example:

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

- Tomcat Realm configuration:

  The current Realm in $JONAS_BASE/conf/server.xml must be replaced by:

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

The class specified uses the JAAS model to authenticate the users. Thus, to choose the correct realm to be used for authentication, JAAS must be configured see in Section 2.5.4, “JAAS configuration”.

- Tomcat SSL configuration:

  The following example of <connector> element must be uncommented in $JONAS_BASE/conf/server.xml and customized (if necessary):

```
<Server>
[...]
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 9043 -->
<!--
<Connector port="9043" maxHttpHeaderSize="8192"
maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
enableLookups="false" disableUploadTimeout="true"
acceptCount="100" scheme="https" secure="true"
clientAuth="false" sslProtocol="TLS" />
-->
[...]
</Server>
```

  A complete description of SSL configuration can be found in SSL Configuration HOW-TO [http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html]

- Webapp configuration:

  In the `web.xml`of the web application a *Client Certificate Authentication Configuration* must be set, a security-constraint may be used if needed; example:

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

<security-constraint>
..
<user-data-constraint>
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
</web-app>
```

2.5.2.3. Servlet Authentication with User/password and Jetty 5.1.x

- Jetty configuration

  A web-jetty.xml file must be provided in the `WEB-INF` directory in the `.war` file in which a security interceptor `org.objectweb.jonas.security.realm.web.jetty50.Standard` form is specified instead of the default one:

```
<Configure class="org.mortbay.jetty.servlet.WebApplicationContext">
<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>
</Configure>
```

Several `web-jetty.xml` examples are located in the `$JONAS_ROOT/examples/earsample` example and `$JONAS_ROOT/examples/alarm` demo.

- webapp configuration:

  is similar to the webapp configuration with Tomcat see above.

2.5.2.4. Authentication with certificate and Jetty 5.1.x

- Jetty Realm configuration:

  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 correct *realm* to be used for authentication, JAAS must be configured, see in Section 2.5.4, “JAAS configuration”.

- Jetty SSL configuration:

  In the global deployment descriptor of Jetty (the jetty5.xml file), located in the $JONAS_BASE/conf directory, uncomment this part:

```
<!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<!-- Add a HTTPS SSL listener on port 9043                           -->
<!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<!-- UNCOMMENT TO ACTIVATE
<Call name="addListener">
<Arg>
<New class="org.mortbay.http.SunJsseListener">
<Set name="Port">9043</Set>
<Set name="MinThreads">5</Set>
<Set name="MaxThreads">100</Set>
<Set name="MaxIdleTimeMs">30000</Set>
<Set name="LowResourcePersistTimeMs">2000</Set>
<Set name="Keystore"><SystemProperty name="jetty.home" default="."/>/etc/demokeystore</Set>
<Set name="Password">OBF:1vny1zlo1x8e1vnw1vn61x8g1zlu1vn4</Set>
<Set name="KeyPassword">OBF:1u2u1wml1z7s1z7a1wnl1u2g</Set>
</New>
</Arg>
</Call>
-->
```

  A complete description of howto configure SSL for Jetty may be founf here [http://jetty.mortbay.org/jetty5/faq/faq_s_400-Security_t_ssl.html]

- webapp configuration

  is similar to the webapp configuration with Tomcat see above

- jonas-realm configuration

  is similar to the configuration with Tomcat see here [3]

2.5.3. Client container Authentication

To enable authentication mechanism in a client container it is necessary to

- choose a *callback handler*

  Callback handlers are responsible to get the user identity and to store it.

  The choice of the *callback handler* is done in the `application-xml` file, for example:

```
<application-client>
<callback-handler>org.objectweb.jonas.security.auth.callback.LoginCallbackHandler</callback-handler>
</application-client>
```

  JOnAS provides several *callback handlers*[12]*:*

  - `LoginCallbackHandler` : it is a text based handler that gets the user and password via stdin
  - `DialogCallbackHandler:` handler using a Swing dialog window to query user and password
  - `NoInputCallbackHandler`: is responsible to store a user/password
  - `CertificateCallback`: is responsible to store a certificate

- configure JASS for setting the LoginModules to be used to perform authentication see Section 2.5.4, “JAAS configuration”

In the `$JONAS_ROOT/examples/jaasclient` directory can be found three examples of container clients using JAAS authentication as well as one java client without container client that uses also JAAS.

2.5.4. JAAS configuration

The JAAS configuration is made via the *JAAS Login Configuration File*

A login configuration file consists of one or more entries, each specifying which underlying authentication technology should be used for a particular application or applications.

The contents of the JAAS configuration file has the structure below:

```
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;
```

```
};
```

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

- **required** - The LoginModule is required to succeed. If it succeeds or fails, authentication still proceeds through the LoginModule list.
- **requisite** - The LoginModule is required to succeed. If it succeeds, authentication continues through the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed through the LoginModule list).
- **sufficient** - The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed through the LoginModule list). If it fails, authentication continues through the LoginModule list.
- **optional** - The LoginModule is not required to succeed. If it succeeds or fails, authentication still proceeds through the LoginModule list.

2.5.4.1. Default JAAS configuration

JOnAS provides in `$JONAS_BASE/conf/jaas.config` a *JAAS Login Configuration File* already configured with some login configuration.

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.

The predifined entries are:

- **tomcat** used for authentication with the web container Tomcat
- **jetty** used for authentication with the web container Jetty
- **jaasclient** used when running `$JONAS_ROOT/examples/jaasclient` examples

The default configuration for the web container Tomcat is the following:

```
tomcat {
org.objectweb.jonas.security.auth.spi.JResourceLoginModule required
resourceName="memrlm_1"
;
};
```

this indicates that the `JResourceLoginModule` Login Module must be used on the memory realm named `memrlm_1.`

The default configuration for the web container Jetty is the same than the previous:

```
jetty {
org.objectweb.jonas.security.auth.spi.JResourceLoginModule required
resourceName="memrlm_1"
;
};
```

the configuration for the container clients examples :

```
jaasclient {
// Login Module to use for the example jaasclient.

org.objectweb.jonas.security.auth.spi.JResourceLoginModule required
resourceName="memrlm_1"

org.objectweb.jonas.security.auth.spi.ClientLoginModule  required
```

```
globalCtx="true"
;
};
```

Here two Login Modules are used, one for checking the identity in the memoty realm, the second for propagating a security context with the client request.

To change the location and name of the `jaas.config` file, edit the `$JONAS_BASE/bin/jonas.sh` script and modify the following line:

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

2.5.4.2. JOnAS LoginModules

JOnAS provides some predefined LoginModules:

JResourceLoginModule

This is the main LoginModule. It is highly recommended that this one be used in every authentication, as it checks the user authentication information in the specified realm database, LDAP or memory.

This LoginModule delegates the authentication to the server . Here are the possible attributes to set:

attribute name	description
resourceName	name of the realm
serverName	name of JOnAS instance (default value= jonas)
useUpperCaseUsername	if true Convert the username into uppercase for the authentication (default value=false)
certCallback	if true use certificate callback

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 an LDAP repository must exist.

attribute name	description
CRLsResourceName	specifies how the *CRLs* are stored:Two possible values "Directory" or "LDAP"
CRLsDirectoryName	The directory containing the *CRL* files (the extension for these files must be `.crl`).
address	address of the server that hosts the LDAP repository
port	port used by the LDAP repository; *CRLs* are retrieved from an LDAP directory using the LDAP schema defined in RFC 2587 [http://www.ietf.org/rfc/rfc2587.txt]

SignLoginModule — login module that signs the current Subject ,. Here are the possible attributes to set:

attribute name	description
keystoreFile	Name of the key store
keystorePass	password for the keystore
keyPass	password for the private key
alias	alias

ClientLoginModule — login module used for propagating the Principal and roles to the server, it doesn't make any authentication. This login module must be used when authentication for a client container. Here is the possible attribute to set:

attribute name	description
globalCtx	if true set the security context for all the threads of the client container instead of only on the current thread. Useful for swing client. (default value= false)

2.6. Configuring JDBC Resource Adapters

Connection of an J2EE application to databases is done through JDBC Resource Adapters (JDBC RA).

Such Resource Adapters are deployed via the **resource** service as seen in Section 2.4.5, “resource service”.

For both container-managed or bean-managed persistence, the JDBC Resource Adapter 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 [http://java.sun.com/javase/technologies/database/index.jsp] standard extensions.

An JDBC 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.

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.

2.6.1. Generic JDBC Resource Adapters

The generic JDBC RAs of JOnAS provide implementations of the `java.sql.Driver`, `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, and `javax.sql.XADataSource` interfaces. They are located in the `$JONAS_ROOT/rars/autoload`

directory and thus are deployed automatically.They consist of base (or generic) RAs facilitating the build of the user JDBC RAs.

Depending on the relational database management server and the available interface in the used JDBC-compliant driver, the user JDBC RA is linked (through the RAR link feature) to a generic RA (for example, the Driver's one). In this case, the user RA contains only a jonas-ra.xml file with some specific parameters, such as the connection url, the user/password, or the JDBC-Driver class.

Resource adapter provided with JOnAS	description	jndi name
rars/autoload/JOnAS_jdbcDS.rar	Generic JDBC RA that implements the `DataSource` interface	JOnASJDBC_DS
rars/autoload/ JOnAS_jdbcDM.rar	Generic JDBC RA that implements the `Driver` interface	JOnASJDBC_DM
rars/autoload/JOnAS_jdbcCP.rar	Generic JDBC RA that implements the `ConnectionPoolDataSource` interface	JOnASJDBC_CP
rars/autoload/JOnAS_jdbcXA.rar	Generic resource adapter that implements the `XADataSource` interface	JOnASJDBC_XA

2.6.2. Specific JDBC Resource Adapter

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 [PG_Entity.html#PG_Entity-Writing]" of the Developing Entity Bean Guide [PG_Entity.html#PG_Entity]).

An RAR file must be deployed as explained in Section 2.4.5, “resource service”.

Usually a resource Adapter contains in its `rar` file all the classes needed to access to the external resource. In the case of a specific JDBC RA it contains only a JOnAS specific deployment descriptor `jonas-ra.xml` that tell what sort of generic resource adapter to use and information related to the specific database used. The `jar` file of the actual JDBC driver mut be copied in the right place to be seen by the JOnAS classloader : `$JONAS_BASE/lib/commons.`

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

- With the RAConfig command (refer to the JOnAS Commands Reference Guide [command_guide.html] for a complete description of the command).
- Through the jonasAdmin console (refer to Administration guide for a complete description). In the jonasAdmin's tree, the Resource Adapter Module node (under the deployment node) contains a configure tab that allows editing of both the ra.xml file and the jonas-ra.xml file of the undeployed RA.

2.6.2.1. Defining the JOnAS Connector Deployment Descriptor: jonas-ra.xml

The jonas-ra.xml contains JOnAS specific information describing deployment information, logging, pooling, jdbc connections, and RAR config property values:

- *Deployment Tags*:

property name	description	possible values
jndiname	name the RAR will be registered as. This property is required This value will be used in the resource-ref section of an J2EE composant.	• Anyname (for example jdbc_1)
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`	• JONASJDBC_DM • JONASJDBC_DS • JONASJDBC_CP • JONASJDBC_XA
native-lib	defines the path where native libraries can be found.	• Any string for a path

- *Logging Tags*:

property name	description	possible values
log-enabled	determines if logging should be enabled for the RAR.	• False (default value) • True
log-topic:	defines the log topic that will be used to write log messages for this rar file.	• Any topic name • Default value is org.objectweb.jonas.jca

- *Pooling Tags*

property name	description	possible values
pool-init	Initial size of the managed connection pool	• 0 (default value) • n
pool-min	Minimum size of the managed connection pool.	• 0 (default value) • n
pool-max	Maximum size of the managed connection pool.	• n • -1 = unlimited (default value)
pool-max-age-minutes	Maximum number of minutes to keep the managed connection in the pool.	• 0 = an unlimited amount of time. • n in minutes
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	• 0 = unlimited • n (default value = 10) • -1 = cache disabled

	of 0 is unlimited and -1 disables the cache.	
pool-max-opentime	Identifies the maximum number of minutes that a managed connection can be left busy.	• 0 = an unlimited amount of time (default value). • n in minutes
pool-max-waiters:	identifies the maximum number of waiters for a managed connection. Default value is 0.	• 0 (default value) • n
pool-max-waittime	identifies the maximum number of seconds that a waiter will wait for a managed connection. Default value is 0.	• 0 (default value) • n in seconds
pool-sampling-period:	identifies the number of seconds that will occur between statistics samplings of the pool. Default is 30 seconds.	• n in seconds (default value = 30s)

- *JDBC Connection Tags*:

Note

Only valid for Connection implementation of java.sql.Connection.

property name	description	possible values
jdbc-check-level	Level of checking that will be done for the jdbc connection.	• 0 : no check (default value) • 1: check connection still open • 2 : send the test statement before reusing a connection from the pool • (keep-alive feature) send the test statement on each connection every pool-sampling-period
jdbc-test-statement	Test SQL statement sent on the connection if the jdbc-check-level is greater than 1.	• A SQL statement

- *Config Property Value Tags:*

Each entry must correspond to the config-property specified in the ra.xml of the RAR file. The default values specified in the ra.xml will be loaded first and any values set in the jonas-ra.xml will override the specified defaults. These tags differs dependiing on the generic JDBC RA used

property name	description	possible values
dsClass	Name of the class implementing `java.sql.Driver`, `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`,	• any classname representing a JDBC driver (example:org.postgresql.Driver)

	or `javax.sql.XADataSource` interfaces in the JDBC driver.	
URL	Database url of the form jdbc:<database_vendor_subprotocol>: This property may be used only for JDBC RA that implements the Driver (JDBC_DM)	• Any url valid for a database provider (example:jdbc:postgresql://localhost:5432/mydb)
user	Database user name	• any name
password:	Database password	• any string
loginTimeout	Maximum time in seconds that the driver will wait while attempting to connect to a database.	• no value = 0 (default value) • n in seconds
isolationLevel	Level of transaction isolation	• none • serializable • read_committed • read_uncommitted • repeatable_read
mapperName	Name of the JORM mapper	The possible values can be found in the List of available mappers in JORM documentation [http://jorm.objectweb.org/doc/mappers.html].
databaseName	Name of the database	• any name
description:	Informal description	• any String
portNumber	Port Number of the database server	• a number
serverName	Name of the database server.	• any name
dbSpecificMethods	allow flexibility to call setter methods on the dsClass as required by the database provider	see below the particular syntax

- dbSpecificMethods a specific property:

  The JOnAS JDBC Resource Adapter is built as a generic connector to any database provider. The limitation of this is that each database provider may have different requirements about the methods needed to configure the `dataSource` class.This dbSpecificMethods property was added to allow flexibility to call setter methods on the dsClass as required by the database provider. The specific information about what additional methods should be used is documented by the database provider. The format of the value specified is:

  [:<del_char>]<method>=<value>::<value_type>:<method>=<value>::<value_type>....with:

:	optional starting value that denotes using the next character as the delimiter instead of the default ':'

<del_char>	delimiter character to use
<method>	method to call followed by an = sign
<value>	the parameter value to pass to the method being called, followed by 2 delimiter characters.If a Properties object is being passed, then the format of this value must be (name=val, name=val, ...);
<value_type>	the parameter type used to construct the reflection call, followed by the delimiter character if additional methods are being called • Boolean or bool • Byte or byte • Character or char • Double or double • Float or float • Integer or int • Long or long • Properties or java.util.Properties • Short or short • String

2.6.2.2. Understanding pooling tags:

At JDBC RA deployment time, if **pool-init** is not null **pool-init** JDBC connection are created.

When a user requests a jdbc connection, the JDBC RA 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, it creates a new jdbc connection (if **pool-max** is not reached).

If it cannot create new connections, the user must wait (if **pool-max-waiters** is not reached) until a connection is released. After a limited time (**pool-max-waittime**), 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 **pool-sampling-period** 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 a time too long (**pool-max-age**), the pool will try to release it from the freelist. However, the JDBC RA always tries to keep open at least the number of connections specified in **pool-min**.

When the user has forgotten to close a jdbc connection, the system can automatically close it, after **pool-max-opentime** 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-check-level** . For levels >1 it tries a dummy statement on the connection before returning it to the caller. This statement is configured in **jdbc-test-statement**.

Note

this previous description is not only true for JDBC RAs but also for all types of resource adapters, except **jdbc-check-level** and **jdbc-test-statement** which are specifics for JDBC.

2.6.3. Examples of Specific JDBC Resource Adapter

2.6.3.1. Oracle JDBC resource adapter (Driver)

An RAR for Oracle named as `jdbc_1` in JNDI and using the Oracle thin Driver JDBC driver, should be described in a file (called `for example 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 database Oracle, 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.

2.6.3.2. PostgreSQL JDBC resource adapter (Driver)

To create a PostgreSQL RAR configured as jdbc_3 in JNDI, it should be described in a file (called for examplePostgreSQL3_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.postgres</jonas-config-property-value>
</jonas-config-property>
</jonas-connector>
```

2.6.3.3. Oracle JDBC resource adapter (XADataSource)

An RAR for Oracle configured as jdbc_4 in JNDI and using the Oracle `XADataSource` interface of the JDBC driver thin in order to use a JDBC2-XA-compliant driver.It may be described in a file (called for example Oracle1_XA.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_4</jndi-name>
<rarlink>JOnASJDBC_XA</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>databaseName</jonas-config-property-name>
<jonas-config-property-value>dbjonas</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
<jonas-config-property-name>portNumber</jonas-config-property-name>
<jonas-config-property-value>1521</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
<jonas-config-property-name>serverName</jonas-config-property-name>
<jonas-config-property-value>wallis</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
<jonas-config-property-name>dbSpecificMethods</jonas-config-property-name>
<jonas-config-property-value>:#setDriverType=thin##String</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.xa.client.OracleXADataSource</jonas-config-property-value>
</jonas-config-property>
</jonas-connector>
```

2.6.4. Tracing SQL Requests through P6Spy

The P6Spy [http://www.p6spy.com/]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 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-value>
</jonas-config-property>
<jonas-config-property>
<jonas-config-property-name>dsClass</jonas-config-property-name>
<jonas-config-property-value>com.p6spy.engine.spy.P6SpyDriver</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
<jonas-config-property-name>mapperName</jonas-config-property-name>
<jonas-config-property-value>rdb.postgres</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
<jonas-config-property-name>logTopic</jonas-config-property-name>
<jonas-config-property-value>org.objectweb.jonas.jdbc.DMPostgres</jonas-config-property-value>
</jonas-config-property>
</jonas-connector>
```

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

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

In `$JONAS_BASE/conf/trace.properties`:

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

2.6.5. Migration from dbm service to the JDBC RA

The migration of a `Database.properties` file to a similar Resource Adapter can be accomplished through the execution of the following RAConfig tool command. Refer to the JOnAS Commands Reference Guide [command_guide.html] for a complete description of RAConfig command.

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

Generates a `MySQL_dm.rar` file linked to `JOnAS_jdbcDM.rar`, the `jonas-ra.xml` file inserted is created with values coming from the `ra.xml` file of the `JOnAS_jdbcDM.rar` and values from the `MySQL1.properties` file

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_dm.rar` file using the following command:

```
RAConfig -path .  MySQL_dm.rar 1

RAConfig -u jonas-ra.xml MySQL_dm.rar 2
```

1 Extraction of `jonas-ra.xml` of `MySQL_dm.rar` in the working directory
2 update `MySQL_dm.rar` with `jonas-ra.xml`

2.7. Configuring JMS Resource Adapters

JMS Resource adapters can be deployed, either via the `jonas.properties` file, or via the JonasAdmin tool, or included in the autoload directory of the **resource** service.

JMS connections are obtained from a JMS RA, which is configured to identify and access a JMS server.

The JORAM resource adapter archive (`joram_for_jonas_ra.rar`) is provided with the JOnAS distribution. It is located in the `$JONAS_ROOT/rars/autoload` directory and thus is deployed automatically.

Note

jms must not appear in the list of services of the `jonas.properties` file because the JORAM's rar and the `jms` service are exclusive.

2.7.1. JORAM Resource Adapter configuration files

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 both an `a3server.xml` file located in the `$JONAS_BASE/conf` directory and the jonas-ra.xml file located within the RA.
- Creates managed JMS ConnectionFactory instances and binds them with the names **CF**, **QCF**, and **TCF**.

- Creates administered objects for this server (JMS destinations and non-managed factories) as described by the `joramAdmin.xml`, located in the `$JONAS_BASE/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.

The JORAM integration into JOnAS is composed of 3 different parts: server, RA, and administration. Each part contains its own configuration files:

- a3servers.xml is the JORAM platform configuration file, i.e. the server part. The file is located in the `$JONAS_BASE/conf` directory.
- `ra.xml` and `jonas-ra.xml` are the resource adapter configuration files. They are embedded in the resource adapter (META-INF directory).
- `joramAdmin.xml` contains the administration tasks to be performed by the JORAM server such as the JMS objects creation. It is located in the `$JONAS_BASE/conf` directory.

2.7.1.1. JORAM server configuration : a3servers.xml

The a3server.xml (`$JONAS_BASE/conf/a3server.xml`) file describes the JORAM platform, i.e., the network domain, the used transport protocol, and the reachable JORAM servers. It is used by a JORAM server at start time. By default, only one collocated JORAM server is defined (s0) based on the tcp/ip protocol. A distributed configuration example is provided in the how-to document and other examples are available in JORAM's user guide.

```
<config>
<property name="Transaction" value="fr.dyade.aaa.util.NullTransaction/> 1
<server id="0" name="S0" hostname="localhost"> 2
<service class="org.objectweb.joram.mom.proxies.ConnectionManager"
args="root root"/>
<service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
args="16010"/> 3

</server>
</config>
```

1 This property means that the non persistent mode for JMS is choosen. In order to use persistent mode, the value must be changed to "fr.dyade.aaa.util.NTransaction"
2 Here can be set the server id and the host where the server run
3 args specifies the port number the JORAM server is listening on

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

The joram_raconfig command allows to modify these parameters in all the configuration files.

If used in non-collocated mode, joram can be started with the **JmsServer** command which loads the $JONAS_BASE/conf/a3server.xml configuration file.

2.7.1.2. Resource Adapter configuration: ra.xml, jonas-ra.xml

The `ra.xml` file is the standard deployment descriptor for the JORAM adapter and the `jonas-ra.xml` file is the JOnAS-specific deployment descriptor for the JORAM adapter. These files set the central configuration of the adapter, define and set managed connection factories for outbound communication, and

define a listener for inbound communication. `jonas-ra.xml` contains specific parameters such as pool parameters or jndi names, but also may redefine the parameters of some `ra.xml` files and override their values. Globally, a good way to proceed is to keep the original `ra.xml` file with the default values and to customize the configuration only in the `jonas-ra.xml` file.

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

- With the RAConfig command to extract jonas-ra.xml, do the following:

```
RAConfig -path . joram_for_jonas_ra.rar
```

  Then, to update the archive, do the following:

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

- Through the jonasAdmin console (refer to Administration guide for a complete description).

  In the jonasAdmin's tree, the Resource Adapter Module node (under the deployment node) contains a configure tab that allows editing of both the ra.xml file and the jonas-ra.xml file of the undeployed RA.

- Through the joram_raconfig utility (refer to joram_raconfig description for a complete description).

  This tool allows easy modification to the network parameters of the JORAM server in all the configuration files.

The following properties are related to the central configuration of the adapter; they are set via some <jonas-config-property> elements:

property name	description	possible values
CollocatedServer	Running mode of the JORAM server to which the adapter gives access.	• True: when deploying, the adapter starts a collocated JORAM server. • False: when deploying, the adapter connects to a remote JORAM server. • Nothing (default True value is then set).
PlatformConfigDir	Directory where the `a3servers.xml` and `joramAdmin.xml` files are located.	• Any String describing an absolute path (ex: /myHome/myJonasRoot/conf). • Empty String, files will be searched in $JONAS_BASE/conf • Nothing (default empty string is then set).
PersistentPlatform	Persistence mode of the collocated JORAM server. - not taken into account if the JORAM server is set as non-collocated. - If true, set	• True: starts a persistent JORAM server. • False: starts a non-persistent JORAM server.

	the property 'Transaction' to 'fr.dyade.aaa.util.NTransaction' before launching the JORAM server. - If false, set the property 'Transaction' to 'fr.dyade.aaa.util.NullTransaction' before launching the JORAM server. - Warning, if the 'Transaction' property is set in the a3server.xml file, this value is ignored.	• 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. If the JORAM server is non-collocated, it must be set to the name of the already started JORAM server (this is necessary for management purpose).	• Name corresponding to the server to start (in collocated case) or to the started server as described in the a3servers.xml file (ex: s1). • Nothing (default s0 name is then set).
AdminFileXML	Name of the file describing the administration tasks to be performed by the JORAM server, i.e., JMS destinations to create, users to create, ... If the file does not exist, or is not found, no administration task is performed.	• Name of the file (ex: myAdminFile.xml). • Nothing (default joramAdmin.xml name is then set).
HostName	Name of the host where the JORAM server runs, used for accessing a remote JORAM server (non-collocated mode), and for building appropriate connection factories.	• Any host name (ex: myHost). • Nothing (default localhost name is then set).
ServerPort	Port the JORAM server is listening on, used for accessing a remote JORAM server (non-collocated mode), and for building appropriate connection factories.	• Any port value (ex: 16030). • Nothing (default 16010 value is then set).
ConnectingTimer	Duration in seconds during which connecting is attempted (connecting might take time if the server is temporarily not reachable)	• 0 : set for connecting only once and aborting if connecting failed (default value) • n : duration in seconds
CnxPendingTimer	Period in milliseconds between two ping requests sent by the client connection to the server;	• 0 means "notimer" (default value)

		• n: duration in milliseconds
TxPendingTimer	Duration in seconds during which a JMS transacted (non XA) session might be pending; above that duration the session is rolled back and closed.	• 0 value means "no timer". • n: duration in seconds
DeleteDurableSubscription	Indicates the durable Subscriptions must be deleted when the consumer is closed	• True (previous behaviour) • False (default value)

The <jonas-connection-definition> elements wrap properties related to the managed connection factories:

There are three managed connection factories:

- A Queue managed connection factory registered in JNDI with the name **QCF**
- A Topic managed connection factory registered in JNDI with the name **TCF**
- A managed connection factory registered in JNDI with the name **CF**

Here are the properties that can be configured for each managed connection factory:

property name	description	possible values
jndi-name	Name used for binding the constructed connection factory.	Any name (ex: myQueueConnectionFactory). Default values are • **QCF** for the Queue managed connection factory • **TCF** for the Topic managed connection factory • **CF** for the managed connection factory
UserName	Default user name that will be used for opening JMS connections.	• Any name (ex: myName). • Nothing (default *anonymous* name will be set).
Password	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	• True (for building local-optimized connections). • False (for building TCP connections). • Nothing (default TCP mode will be set).

The <jonas-activationspec> element 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 (by default:joramActivationSpec).

The Pooling Tags are the same than those for other RAs:

property name	description	possible values
pool-init	Initial size of the managed connection pool	• 0 (default value) • n
pool-min	Minimum size of the managed connection pool.	• 0 (default value) • n
pool-max	Maximum size of the managed connection pool.	• n • -1 = unlimited (default value)
pool-max-age-minutes	Maximum number of minutes to keep the managed connection in the pool.	• 0 = an unlimited amount of time. • n in minutes
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.	• 0 = unlimited • n (default value = 10) • -1 = cache disabled
pool-max-opentime	Identifies the maximum number of minutes that a managed connection can be left busy.	• 0 = an unlimited amount of time (default value). • n in minutes
pool-max-waiters:	identifies the maximum number of waiters for a managed connection. Default value is 0.	• 0 (default value) • n
pool-max-waittime	identifies the maximum number of seconds that a waiter will wait for a managed connection. Default value is 0.	• 0 (default value) • n in seconds
pool-sampling-period:	identifies the number of seconds that will occur between statistics samplings of the pool. Default is 30 seconds.	• n in seconds (default value = 30s)

2.7.1.3. JMS Applications Configuration

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

In earlier version the `joram-admin.cfg` was used for this same purpose but it is now deprecated.

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

Note

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

The format of this file is XML. Here are some examples:

- simple example:

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

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

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

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

```
<AdminModule>
<connect host="localhost"
```

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

The port number must be set with the server port number (defined in the `a3servers.xml` and in the JORAM's RAR configuration `ra.xml` and `jonas-ra.xml` files).

- Possible parameters for a queue definition:

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

- Possible parameters for a topic definition:

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

- Example of a dead message queue definition:

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

- Example of a scheduler queue definition:

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

- Example of a clustered queues destination:

```
<Cluster>
<Queue name="queue0"
serverId="0"
className="org.objectweb.joram.mom.dest.ClusterQueue">
<freeReader/>
<freeWriter/>
<property name="period" value="10000"/>
<property name="producThreshold" value="50"/>
```

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

2.7.1.4. joram_raconfig command

Name

joram_raconfig — Changes to the parameters (host, port, server id) in the JORAM configuration files.

Synopsis

`joram_raconfig` [[-p <port>] [-h <host>] [-s <serverid>]]

Description

The joram_raconfig tool aims to facilitate changes to the parameters (host, port, server id) in the JORAM configuration files.

JORAM relies on several configuration files: `a3servers.xml`, `joramAdmin.xml`, `ra.xml.` With joram_raconfig, all these configuration files are updated and thus the consistency is ensured.

Files modified:

- `$JONAS_BASE/conf/a3servers.xml`
- `$JONAS_BASE/conf/joramAdmin.xml`
- `$JONAS_BASE/rars/autoload/joram_for_jonas_ra.rar` in which the file `META_INF/ra.xml` is updated.

Options

-p port	port : listening port of the JORAM server
-h host	host : IP address of the JORAM server
-s serverid	serverid : server id of the JORAM server

2.7.2. JORAM's Resource Adapter tuning

2.7.2.1. ManagedConnection Pool

A pool of ManagedConnection is defined for each factory (connection definition) specified in the jonas-ra.xml file. See the pool parameters in the Section 2.7.1.2, “Resource Adapter configuration: ra.xml, jonas-ra.xml” [53].

2.7.2.2. Session/Thread pool in the JORAM RA

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

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

The maxNumberOfWorks property can be set in the message-driven bean standard deployment descriptor. For example, the code below can be added to limit the number of parallel sessions to 100 (default value is 10).

```
<activation-config-property>
<activation-config-property-name>maxNumberOfWorks</activation-config-property-name>
<activation-config-property-value>100</activation-config-property-value>
</activation-config-property>
```

As this parameter set the max number of messages that can be treated simultaneously, the max-cache-size must be set accordingly in the specific deployment descriptor.

2.7.3. Undeploying and Redeploying a JORAM Adapter

Undeploying a JORAM adapter either stops the collocated JORAM server or disconnects from a remote JORAM server. It is then possible to deploy the same adapter again. If set for running a collocated server, it will re-start it. If the running mode is persistent, then 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").

2.8. Configuring JDBC DataSources

This section describes how to configure the Datasources for connecting application to databases when the **dbm** service is used.

2.8.1. 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 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 that will be registered in JNDI registry.

This section explains how `DataSource` objects can be defined and configured in the JOnAS server.

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.

Neither the EJB specification nor the Java EE specification describe how to define `DataSource` objects so that they are available to a Java EE 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 [PG_Entity.html#PG_Entity-Writing]" of the Developing Entity Bean Guide [PG_Entity.html#PG_Entity]).

A `DataSource` object should be defined in a file called <DataSource name>.properties (for example Oracle1.properties for an Oracle DataSource or Postgres.properties for an PostgreSQL DataSource.These files must be located in `$JONAS_BASE/conf` directory.

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

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

The property file defining a `DataSource` may contain two types of information:

- connection properties

- JDBC Connection Pool properties

2.8.1.1. connection properties

property name	Description
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
datasource.isolationLevel	Database isolation level for transactions. Possible values are: • none, • serializable, • read_committed, • read_uncommitted, • repeatable_read The default depends on the database used.
datasource.mapper	JORM database mapper (for possible values see here) [http://jorm.objectweb.org/doc/mappers.html]

2.8.1.2. Connection Pool properties

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.

All these settings have default values and are not required. All these attributes can be reconfigured when JOnAS is running, with the console JonasAdmin.

property	Description	Default value
jdbc.conncheclevel	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.conncheclevel = 2.	select 1
jdbc.connmaxage	nb of minutes a connection can be kept in the pool. After this time,	1440 mn (= 1 day)

	the connection will be closed, if minconpool limit has not been reached.	
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	Maximum of concurrent waiters for a JDBC Connection. This is valid only if maxconpool has been set.	1000
jdbc.pstmtmax	Maximum number of prepared statements cached in a Connection. Setting this to a bigger value (120 for example) will lead to better performance, but will use more memory. The recommendation is to set this value to the number of different queries that are used the most often. This is to be tuned by administrators.	12

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.conncheckleve. The maximum level is to try a dummy statement on the connection before returning it to the caller. This statement is configured in jdbc.conntestst mt

2.8.1.3. DataSource example:

Here is the template for an Oracle dataSource.properties file that can be found in $JONAS_ROOT/conf:

```
###################### Oracle DataSource configuration example
#


#####
#  DataSource configuration
#
datasource.name  jdbc_1
datasource.url  jdbc:oracle:thin:@<your-hostname>:1521:<your-db>
datasource.classname oracle.jdbc.driver.OracleDriver
datasource.username <your-username>
datasource.password <user-password>
datasource.mapper rdb.oracle


#####
#  ConnectionManager configuration
#

#  JDBC connection checking level.
#     0 = no special checking
#     1 = check physical connection is still open before reusing it
#     2 = try every connection before reusing it
jdbc.conncheckleve l 0

#  Max age for jdbc connections
#  nb of minutes a connection can be kept in the pool
jdbc.connmaxage 1440

# 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.
jdbc.maxopentime 60

#  Test statement
jdbc.conntestst mt select * from dual

# JDBC Connection Pool size.
# Limiting the max pool size avoids errors from database.
jdbc.minconpool 10
jdbc.maxconpool 30

# Sampling period for JDBC monitoring :
# nb of seconds between 2 measures.
jdbc.samplingperiod 30

# Maximum time (in seconds) to wait for a connection in case of shortage.
# This may occur only when maxconpool is reached.
jdbc.maxwaittime 5

# Maximum of concurrent waiters for a JDBC Connection
# This may occur only when maxconpool is reached.
jdbc.maxwaiters 100
```

2.8.2. Tracing SQL Requests through P6Spy

The P6Spy [http://www.p6spy.com/] 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 of dataSource properties file:

```
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
```

Chapter 3. Configuring a domain

3.1. What is a domain

A domain represents an administration perimeter which is under the control of an administration authority. It provides at least one common administration point for the elements in the domain.

A JOnAS domain may contain:

- JOnAS instances
- groups of instances called clusters
- cluster daemons, elements used for the remote control of instances and clusters

A common administration point is represented by a JOnAS instance having a particular configuration and playing the role of **master**. A master has the knowledge of the domain topology and allows executing administration operations on the rest of the servers and on the clusters. It also allows the monitoring of the domain elements.

The administered elements are identified by their names, that have to be unique within the domain, and the domain name.

3.1.1. Naming policy

Names can be defined in a static way, through the `domain.xml` configuration file, or dynamically, by starting new elements in the domain. For example, when starting a JOnAS instance, the administrator can specify the *server name* using the -n option and the *domain name* by setting the **domain.name** environment property. The uniqueness of the starting server's name is enforced by the discovery service.

3.2. What is a domain configuration

A domain configuration consists in the domain topology - the description of the elements composing the domain, and the state of the elements in the domain, as viewed from the common administration point.

Before starting the master, the administrator can define an initial domain topology using the `domain.xml` configuration file.

The domain configuration dynamically evolves by starting or stopping servers and by creating or removing clusters in the domain.

3.3. How to configure a domain

3.3.1. Choose the domain name and configure the master

The first step is to choose a name for the domain (domainName) and to choose a server to represent the common administration point. This means install a JOnAS server if needed, then configure it as a master, namely by adding the discovery service in the JOnAS services list (jonas.services property) and setting the jonas.service.discovery.master property to true.

The domain name is not a configuration property for the master (neither for any server in the domain), but it has to be specified when starting the master.

Before starting the master, the administrator can define the domain's initial topology by editing the `domain.xml` configuration file.

3.3.2. Define the domain initial topology

This step is optional. It consists in defining the domain elements using the `domain.xml` configuration file located in the master's directory. If the administrator has no specific configuration needs, it should at least check the domain name element, and set its value to the chosen domainName. That file can also be used to define a default user name and password to use when connecting to servers and cluster daemons. Moreover, the administrator can choose to remove the `domain.xml` file.

The elements that can be defined in domain.xml are:

- server elements: allow to define a server in the domain, or a server in a cluster. A server has a name, a description, a location and optionally a user name and password as well as an associated cluster daemon. The location can be represented by a list of JMX remote connector server URLs.
- cluster elements: allows to group servers in a logical cluster.
- cluster daemon elements: allows to define a cluster daemon in the domain. A cluster daemon element has a name, a description, a location and optionally a user name and password. The location can be represented by a list of JMX remote connector server URLs.

3.3.3. Domain configuration at master start-up

Start the master in the domain:

```
jonas start -n masterName -Ddomain.name=domainName
```

Note that the domain name can be specified by setting a `domain.name` environment property.

Once started, the administrator can manage and monitor the following elements in the domain through JonasAdmin, or another JMX based administration application, running on the master:

- servers declared in the domain.xml file.
- other servers already started in the domain having the discovery service enabled.
- clusters declared in the domain.xml file.
- clusters detected by the administration framework
- cluster daemons declared in the domain.xml file.

Chapter 4. Configuring a cluster

4.1. What is a cluster

A cluster is a group of JOnAS instances. The servers within a cluster are called cluster members. A server may be a member of several clusters in the domain.

A cluster is an administration target in the domain: from the common administration point, the administrator can apply to a cluster, management operations like deploy or undeploy of applications. It can also monitor the clusters.

4.2. Cluster types

There are two main cluster categories:

- Clusters containing servers that are grouped together only to facilitate management tasks.
- Clusters containing servers that are grouped together to achieve objectives like scalability, high availability or failover.

The clusters in the first category, called logical clusters, are created by the domain administrator based on its particular needs. The grouping of servers (the cluster creation) can be done even though the servers are running.

In the second case, the servers which compose a cluster must have a particular configuration that allows them to achieve the expected objectives. Once servers started, the administration framework is able to automaticaly detect that they are cluster members, based on configuration criteria. Several clusters types are supported by the JOnAS administration framework. They correspond to the different roles a cluster can play:

- TomcatCluster - allows HTTP request load balancing and failover based on the mod_jk Apache connector.
- CmiCluster - allows high availability at web level based on the Tomcat 5 session replication solution.
- HaCluster - enables JNDI clustering and allows load balancing at EJB level, based on the CMI protocol
- JoramCluster - allow transaction aware failover at EJB level and stateful session bean replication.
- JoramHa - allow JMS destinations scalability based on the JORAM clustering solution.
- Java EE - allow JMS destinations failover based on JORAM HA.

4.3. Logical clusters configuration

An administrator can create a cluster if he needs to group some servers into a single administration target. There is no predefined criteria to explicitly group servers for administration purpose.

Cluster names and topology can be defined in a static way, using the domain configuration file `domain.xml`. Here is an example allowing to create a cluster named `mycluster` in `sampleDomain` domain, in which servers `node1` and `node2` are grouped together for administration purpose.

```
<domain xsi:schemaLocation="http://www.objectweb.org/jonas/ns
http://www.objectweb.org/jonas/ns/jonas-domain_4_7.xsd">
<name>sampleDomain</name>
<description>A domain example</description>
<cluster>
<name>mycluster</name>
<description>A cluster example</description>
<server>
<name>node1</name>
<location>
<url>service:jmx:rmi://myhost/jndi/jrmp://myhost:2002/jrmpconnector_node1</url>
</location>
</server>
<server>
<name>node2</name>
<location>
<url>service:jmx:rmi://myhost/jndi/jrmp://myhost:2003/jrmpconnector_node2</url>
</location>
</server>
<cluster>
<domain>
```

Clusters can also be created dynamically via the JonasAdmin [link] management application running on the master.

4.4. JkCluster configuration

A TomcatCluster is a group of servers that provides web level scalability and failover based on the mod_jk [http://tomcat.apache.org/connectors-doc/] Apache connector.

The members of a TomcatCluster are JOnAS instances configured to play the role of mod_jk workers.

The configuration described below correspond to JOnAS servers having the Tomcat based web service activated. It allows to use the Apache Web Server in front of the JkCluster members. The connection uses the AJP protocol.

To setup a JkCluster you need:

- Configure the frontal to support mod_jk.
- Configure the cluster members.
- Configure the master server in order to allow cluster members discovery.

4.4.1. mod_jk configuration

As with other Apache modules, mod_jk should be first installed on the modules directory of the Apache Web Server and the `httpd.conf` file has to be updated. Moreover, mod_jk requires `workers.properties` file that describes the host(s) and port(s) used by the workers.

4.4.1.1. workers.properties file

Here we provide a `workers.properties` file to connect the frontal with two JkCluster members. The file defines a *load-balancing* worker named `myloadbalancer`, and the two *balanced* workers, `worker1` and `worker2`. Each cluster member will be configured to play the role of one of the balanced workers. Additionally, a status worker jkstatus is defined for managing load balancers.

```
# ----------------------
# List the workers name
# ----------------------
```

```
worker.list=myloadbalancer,jkstatus

# ----------------------
# worker1
# ----------------------
worker.worker1.port=9010
worker.worker1.host=localhost
worker.worker1.type=ajp13
# Load balance factor
worker.worker1.lbfactor=1
# Define preferred failover node for worker1
#worker.worker1.redirect=worker2
# Disable worker1 for all requests except failover
#worker.worker1.disabled=True

# ----------------------
# worker2
# ----------------------
worker.worker2.port=9011
worker.worker2.host=localhost
worker.worker2.type=ajp13
# Load balance factor
worker.worker2.lbfactor=1
# Define preferred failover node for worker2
#worker.worker2.redirect=worker2
# Disable worker2 for all requests except failover
#worker.worker2.disabled=True

# ----------------------
# Load Balancer worker
# ----------------------
worker.myloadbalancer.type=lb
worker.myloadbalancer.balanced_workers=worker1,worker2
worker.mylloadbalancer.sticky_session=false
# ----------------------
# jkstatus worker
# ----------------------
worker.jkstatus.type=status
```

For a complete documentation about `workers.properties` see the Apache Tomcat Connector guide [http://tomcat.apache.org/connectors-doc/reference/workers.html].

4.4.1.2. Apache configuration

Here is information which should be customized and set in `httpd.conf` directly or included from another file:

```
# Load mod_jk module
# Update this path to match your modules location
LoadModule jk_module modules/mod_jk.so
# Location of the workers.properties file
# Update this path to match your conf directory location (put workers.properties next to httpd.conf)
JkWorkersFile /etc/httpd/conf/workers.properties
# Location of the log file
JkLogFile /var/log/mod_jk.log
# Log level : debug, info, error or emerg
JkLogLevel info
# Select the timestamp log format
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "
# Shared Memory Filename ( Only for Unix platform ) required by loadbalancer
JkShmFile /var/log/jk.shm
# Assign specific URL to the workers
JkMount /clusteredExample myloadbalancer
JkMount /clusteredExample/* myloadbalancer
# A mount point to the status worker
JkMount /jkmanager jkstatus
JkMount /jkmanager/* jkstatus
# Enable the Jk manager access only from localhost
<Location /jkmanager/>
JkMount jkstatus
Order deny,allow
Deny from all
Allow from 127.0.0.1
</Location>
```

For a complete documentation see Apache HowTo [http://tomcat.apache.org/connectors-doc/webserver_howto/apache.html].

4.4.2. Cluster members configuration

Each member needs an AJP13 connector listening on the port defined in the workers.properties file. Moreover, the worker name (here worker1/worker2) must be used as value for the Engine's jvmRoute attribute.

Here is a chunk of server.xml configurations file for the member worker1:

```
<Server>
<!-- Define the Tomcat Stand-Alone Service -->
<Service name="Tomcat-JOnAS">
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 9000 -->
<Connector port="9000" maxHttpHeaderSize="8192" maxThreads="150"
minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
redirectPort="9043" acceptCount="100"
connectionTimeout="20000" disableUploadTimeout="true"/>
<!-- AJP 1.3 Connector on port 9010 for worker.worker1.port in workers.properties file -->
<Connector port="9010" enableLookups="false" redirectPort="9043" protocol="AJP/1.3"/>

<!-- An Engine represents the entry point
You should set jvmRoute to support load-balancing via AJP ie :
-->
<Engine name="jonas" defaultHost="localhost" jvmRoute="worker1">

</Engine>
</Service>
<Server>
```

4.4.3. Master configuration

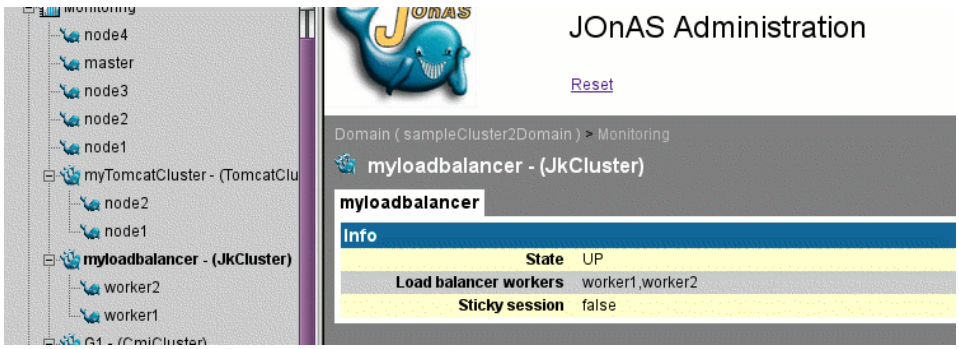
To allow cluster member discovery and dynamic cluster creation, the `workers.properties` file required by mod_jk should be copied to the master's JONAS_BASE/conf directory.

At start-up, the master reads the `workers.properties` file content. For each balanced worker it checks if there is a running server in the domain having the appropriate configuration allowing the server to play that worker role.

Suppose that the master detects a server in the domain corresponding to worker1. Then, it constructs a JkCluster named `myloadbalancer`. This name is given by the load balancer worker's name. At this moment, the cluster is composed of one member named `worker1`. The member name is given by the balanced worker's name.

After a while, a new JOnAS server is started in the domain having the configuration corresponding to the `worker2`. The master detects the new member named worker2 and updates the cluster's member list.

Here is the myloadbalancer JkCluster with workers started:



4.5. TomcatCluster configuration

Additionally to HTTP requests load balancing provided by TomcatCluster members, transparent failover for Web applications can be reached by using HTTP session replication provided by the Tomcat clustering solution.

A CmiCluster cluster members are JOnAS instances having the web service activated, using the Tomcat implementation, and having a specific configuration which allows them to be members of a Tomcat cluster.

The concerned configuration file is the `server.xml` file. Every member of the cluster must have a Cluster element defined in the default virtual host definition. The cluster name is defined by the clusterName attribute, which should be the same for all the cluster members. Another common element for the cluster members is the Membership definition.

The example below defines the configuration for a server which is a TomcatCluster member and a CmiCluster member in the same time.

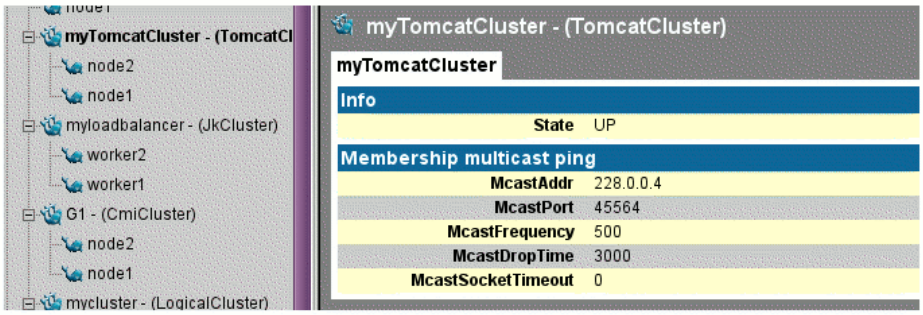
```
<Server>
<!-- Define the Tomcat Stand-Alone Service -->
<Service name="Tomcat-JOnAS">
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 9000 -->
<Connector port="9000" maxHttpHeaderSize="8192" maxThreads="150"
minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
redirectPort="9043" acceptCount="100"
connectionTimeout="20000" disableUploadTimeout="true"/>
<!-- AJP 1.3 Connector on port 9010 (value of worker.worker1.port in workers.properties file) -->
<Connector port="9010" enableLookups="false" redirectPort="9043" protocol="AJP/1.3"/>

<!-- Define the Engine -->
<Engine name="jonas" defaultHost="localhost" jvmRoute="worker1">
</Engine>

<!-- Define the default virtual host -->
<Host name="localhost" debug="0"
appBase="webapps" unpackWARs="false"
autoDeploy="false" deployOnStartup="false" deployXML="false">
<!-- Define a Cluster element -->
<Cluster className="org.apache.catalina.cluster.tcp.SimpleTcpCluster"
clusterName="myTomcatCluster"
managerClassName="org.apache.catalina.cluster.session.DeltaManager"
expireSessionsOnShutdown="false" useDirtyFlag="true"
notifyListenersOnReplication="true">
<Membership className="org.apache.catalina.cluster.mcast.McastService"
mcastAddr="228.0.0.4" mcastPort="45564"
mcastFrequency="500" mcastDropTime="3000"/>
<Receiver className="org.apache.catalina.cluster.tcp.ReplicationListener"
tcpListenAddress="auto" tcpListenPort="4003"
tcpSelectorTimeout="100" tcpThreadCount="6"/>
<Sender className="org.apache.catalina.cluster.tcp.ReplicationTransmitter"
replicationMode="pooled" ackTimeout="15000"/>
<Valve className="org.apache.catalina.cluster.tcp.ReplicationValve"
filter=".*\.gif;.*\.js;.*\.jpg;.*\.png;.*\.htm;.*\.html;.*\.css;.*\.txt;"/>
</Cluster>
</Host>
</Service>
<Server>
```

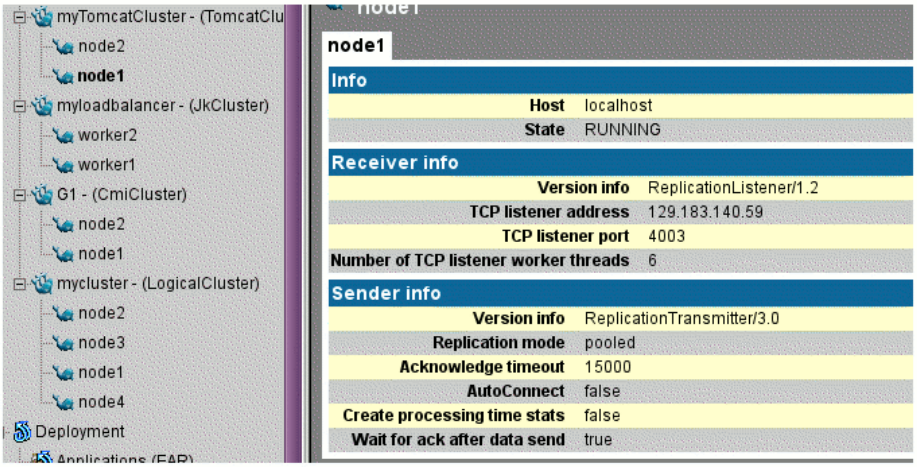
Note

the clusterName attribute is mandatory (and not set by default in `server.xml` file).

Lets consider the two JOnAS servers which play the role of `worker1` and `worker2` in the `myloadbalancer` TomcatCluster. Suppose that these servers, named `node1` and `node2` are configured as members of the `myTomcatCluster` CmiCluster. The master detects automatically the Tomcat cluster membership and creates a CmiCluster named `myTomcatCluster`. It adds `node1` and `node2` to the cluster's member list.

Here is myTomcatCluster cluster with node1 and node2 members running





4.6. CmiCluster configuration

The HaCluster members use the CMI cluster protocol which provides:

- a replicated registry, allowing JNDI high availability for clients doing lookup operations
- CMI cluster stubs, allowing EJB load balancing and failover.

The members of a HaCluster is a JOnAS instance having a particular configuration related to the CMI protocol usage. Moreover, the CMI protocol implementation being based on JGroups [http://www.jgroups.org/javagroupsnew/docs/index.html], a JGroups configuration file is need. Indeed, a Cmi-Cluster members are JGroups group members. The cluster name is given by the JGroups group name.

4.6.1. CMI configuration

The concerned configuration file is the `carol.properties` file.

Common configuration values:

- Have **cmi** in the carol.protocols list.
- Set the cluster name as JGroups group name using the carol.cmi.multicast.groupname property.

Specific configuration values:

- Define the the registry URL using the carol.cmi.url property.
- Specify the JGroups configuration file name. By default, this file is named jgroups-cmi.xml.

4.6.2. JGroup configuration

The configuration file defines the JGroups protocol stack configuration. The default jgroups-cmi.xml file contains a UDP protocol stack based on IP multicast. Each member has to use the same multicast address and multicast port number.

Chapter 5. Glossary

Glossary

Axis [http://ws.apache.org/axis/]	Java platform for creating and deploying web services applications
CAROL [http://carol.objectweb.org/]	Library allowing to use different RMI implementations.
CMI	(Clustered Method Invocation) is the JOnAS protocol cluster for high availability load-balancing and fail-over
EasyBeans [http://www.easybeans.net/xwiki/bin/view/Main/]	An Open source and lightweight EJB3 container that can be embedded in JOnAS and other application servers. It is an ObjectWeb project.
EIS	Enterprise Information Systems
EJB	(Enterprise JavaBeans) technology is the server-side component architecture for Java Platform, Enterprise Edition (Java EE). EJB technology enables rapid development of distributed, transactional, secure and portable applications based on Java technology.
Hibernate	A Java-based object-relational mapping/persistence framework.
IIOP	(Inter-operable Internet Object Protocol) CORBA RPC standard protocol on TCP/IP.
JAAS	(Java Authentication and Authorization Service) is a set of APIs that enable services to authenticate and enforce access controls upon users.
jakarta commons login [http://jakarta.apache.org/commons/logging/]	Wrapper around a variety of logging API implementations.
Java EE	(Java Platform, Enterprise Edition) standard for developing portable, robust, scalable and secure server-side Java applications.
J2CA	(J2EE Connector Architecture) standard for facilitating the integration of application servers with heterogeneous Enterprise Information Systems (EISs).
J2EE	(Java 2 Platform, Enterprise Edition) standard for developing portable, robust, scalable and secure server-side Java applications.up to version 1.5.
JDBC	(Java Database Connectivity) JDBC API provides a call-level API for SQL-based database access.
JDK	(Java Development Kit) A set a Java tools (compiler, jvm, library ...) for Java programs development.
JDO	(Java Data Objects) API is a standard interface-based Java model abstraction of persistence.

Jetty [http://www.mortbay.org/]	is a pure java open-source, standards-based, web server implemented.
JGroups [http://www.jgroups.org/javagroupsnew/docs/index.html]	a toolkit for reliable multicast communication.
JMS	(Java Message Service) is a Java Message Oriented Middleware (MOM) API.
JMX	(Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications.
JNDI	(Java Naming Directory Interface) Standard API/SPI for Java EE naming interface.
JORAM [http://joram.objectweb.org/]	(Java Open Reliable Asynchronous Messaging) is an open source implementation of the JMS API built on top of the ScalAgent [http://www.scalagent.com/] distributed agent technology and hosted by ObjectWeb
JORM [http://jorm.objectweb.org/]	(Java Object Repository Mapping) is an ObjectWeb project that provide an adaptable persistence service.
JOTM [http://jotm.objectweb.org/]	(Java Open reliable Transaction Manager) is an open source implementation of the JTA APIs hosted by ObjectWeb.
JSP	(JavaServer Pages) is a technology that provides a simplified, fast way to create dynamic web content.
JTA	(Java Transaction API) standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system : the resource manager, the application server, and the transactional applications.
JRE	(Java Runtime Environment).
JRMP	(Java Remote Method Protocol) Java RMI standard protocol.
JVM	(Java Virtual Machine) The Java virtual machine.
Log4j [http://logging.apache.org/log4j/docs/index.html]	is a Java-based logging utility (from the Apache Software Foundation). It is used primarily as a debugging tool.
Monolog [http://monolog.objectweb.org/index.html]	is the ObjectWeb solution for logging.
MX4J [http://mx4j.sourceforge.net/]	is an Open Source implementation of the Java Management Extensions (JMX) and of the JMX Remote API (JSR 160) specifications.
P6Spy [http://www.p6spy.com/]	An open source Java tool that intercepts and logs all database statements that use JDBC.
RMI	(Remote Method Invocation) This is the standard specifications of the Java RPC.

RPC		(Remote Procedure Call) all remote method call protocol is a RPC.
Speedo	[http://speedo.objectweb.org/]	is an open source implementation of the JDO 1.0.1 specification hosted by ObjectWeb.
Struts	[http://struts.apache.org/]	Apache Struts is an open-source framework for developing Java EE web applications. It uses and extends the Java Servlet API to encourage developers to adopt a model-view-controller.
Tomcat	[http://tomcat.apache.org/]	Apache Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages.
Velocity	[http://velocity.apache.org/engine/index.html]	The Apache Velocity Engine is a free open-source templating engine.