Design of the Service Activation Reference Implementation

OSS through Java™ Initiative

Andreas Ebbert

SA-API-RiDesign.doc

Copyright © 2001 Nokia Networks Oy All rights reserved. Use is subject to license terms.

Executive Summary

The Service Activation API offers an interface through which all kind of services can be activated and deactivated. This document describes how the Reference Implementation was designed to implement the Service Activation API.

Table of Contents

Preface

Objectives

Design description of the OSS/J Service Activation Reference Implementation

Audience

The target audience are either developers who seek information about how the OSS/J Service Activation API can be implemented or deployers who want to make use of the Service Activation Reference Implementation and extend its supported services.

Approval and Distribution

This document is reviewed and approved by the Service Activation Expert Group.

Latest Versions can be found at the OSS/J site [OSS/J 01].

Related Information

Revision History:

Date	Version	Author	State	Comments
2001-08-07	00.01	Andreas Ebbert	Initial	
2001-08-31	00.02	Andreas Ebbert	Draft	changes for early access version for spec pfd
2002-01-23	00.03	Andreas Ebbert	Draft	New Diagrams
2002-02-07	01.00	Andreas Ebbert	Approved	Last changes

1 Introduction

This document describes the design of the OSS through Java™ Initiative, Service Activation API Reference Implementation.

The Reference Implementation can either be used as a proof-of-concept for the Service Activation API, showing that it is possible to implement the API. Or it can be considered as tutorial, a tool box and a prototype.

A tutorial because the Reference Implementation might offer solutions to problems during the implementation.

A toolbox because some parts can be found reusable during a new implementation of the Service Activation API.

A prototype because you can add new service types to the Reference Implementation. Doing so you can rapidly have a prototype running which supports your very own service.

The use of the prototype is limited in a production environment though, as it probably will not scale perfectly and is not optimized for speed, memory footprint or security.

This document shows how the Reference Implementation was designed. What problems there might be and how it is to be used.

2 General

In general the Reference Implementation is designed as a set of Enterprise Java Beans on top of a RDBMS.

The Reference Implementation has been tested successfully with Weblogic Server 6.1 as application server and Cloudscape as database system. However, every effort was made to avoid any dependence on these special systems, so no reasons are known why the Reference implementation should reject to work on other application servers or database systems. Future versions of the RI will be compatible with the J2EE Reference Implementation as well as with other application servers.

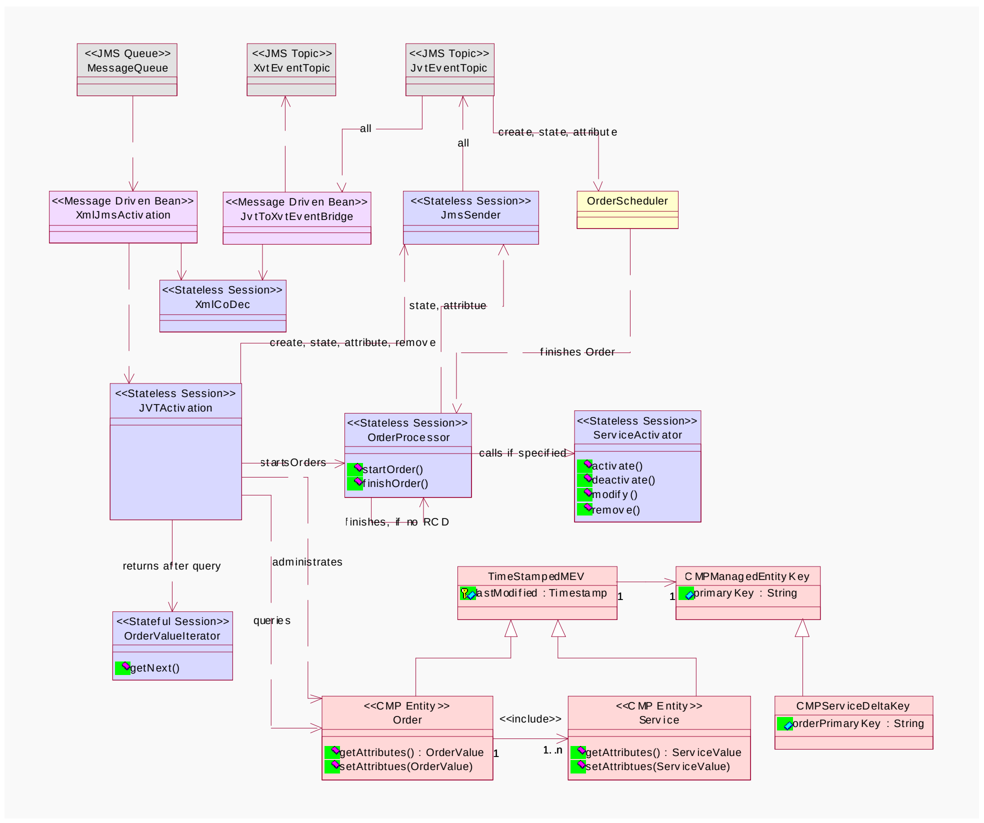
3 Component Overview

The Reference Implementation contains the following components, which are later explained in more detail and in connection with each other:

- **JVT Activation Session:** The JVT Activation Session Bean is the core part of the Reference Implementation. It implements the Java Value Type Interface as defined in the Service Activation specification
- **XmlJmsActivation:** This message driven Bean receives request as defined the in the XML Schema for Service Activation, processes them and then invoces the requested method on the JVT Activation Session Bean, encodes the result as a XML document again and sends that to the requested queue.
- **Order** and **Service:** These entities control the persistence state of all orders and services.
- **Order Processor:** The processor takes care of an order as soon as it was started.
- **Service Activator:** All services used by the reference implementation have some kind of a reference to a service activator which shall activate that peculiar service.
- **Order Scheduler:** If an order is about to be delivered at a specific date, the order is scheduled by the scheduler
- **Jms Sender:** Incorporates all functionality which is needed to send out the appropriate JMS events.
- **JvtToXvtEventBridge:** Transforms all events sent to the JVT Topic to a XML document and sends it to the XVT Topic

In Figure 1 you see an how these components work together. The color codes used mean:

- Ordinary classes or interfaces are yellow,
- Session Beans are blue,
- Entity Beans are read,
- classes or interfaces belonging to the specifications are green and
- JMS related objects are gray.



<<JMS Queue>> MessageQueue
<<JMS Topic>> XvtEventTopic
<<JMS Topic>> JvtEventTopic
all
all
create, state, attribute
<<Message Driven Bean>> XmlJmsActivation
<<Message Driven Bean>> JvtToXvtEventBridge
<<Stateless Session>> JmsSender
OrderScheduler
<<Stateless Session>> XmlCoDec
state, attribtue
create, state, attribute, remove
finishes Order
<<Stateless Session>> JVTActivation
<<Stateless Session>> OrderProcessor
startOrder()
finishOrder()
calls if specified
<<Stateless Session>> ServiceActivator
activate()
deactivate()
modify()
remove()
startsOrders
finishes, if no RCD
returns after query
administrates
TimeStampedMEV
lastModified : Timestamp
CMPManagedEntityKey
primaryKey : String
1
1
<<Stateful Session>> OrderValueIterator
getNext()
queries
<<CMP Entity>> Order
getAttributes() : OrderValue
setAttribtues(OrderValue)
<<include>>
1
1..n
<<CMP Entity>> Service
getAttributes() : ServiceValue
setAttribtues(ServiceValue)
CMPServiceDeltaKey
orderPrimaryKey : String

Figure 1 Enterprise Java Beans

3.1 JVT Activation Session Bean

The JVT Activation Session Bean is a full featured implementation of the OSS/J Service Activation API. It even offers some optional operations like order removal and order priority awareness.

The Services it supports can be adjusted at deployment time. They only have to follow a few rules, which are explained in chapter 6 (Adding New Service Types, page 17).

3.2 XML/JMS Activation

The (optional) XML over JMS interface is only partly implemented so far. It does not use the means of the common API to (de-)serialize XML documents to JVT objects, but implements it's own way of doing so. Further on, only a few methods are supported (getOrderTypes, getServiceTypes, makeOrderValue and makeServiceValue). This component can only be regarded as experimental.

3.3 Order Processor

As soon as an order is started through the JVT Activation Session Bean, the Order Processor takes care of it. The processing of the order consists of two steps. First while starting the order its state is changed and when it is being finished, the call to the Service Activator is made and the state is set to closed.

After changing the order's state to running, it either finalizes the order immediately or quits, depending on the requested delivery date set on the order.

When finalizing the order, either immediately after start or after scheduling through the Order Scheduler, the Service Activator, if specified in the contained service, is called.

3.4 Query Values

The Reference Implementation offers two additional Query Values to the one specified in the API specification. They are probably of no great use but for the Order Scheduler. Regard them as academic study how they may be implemented.

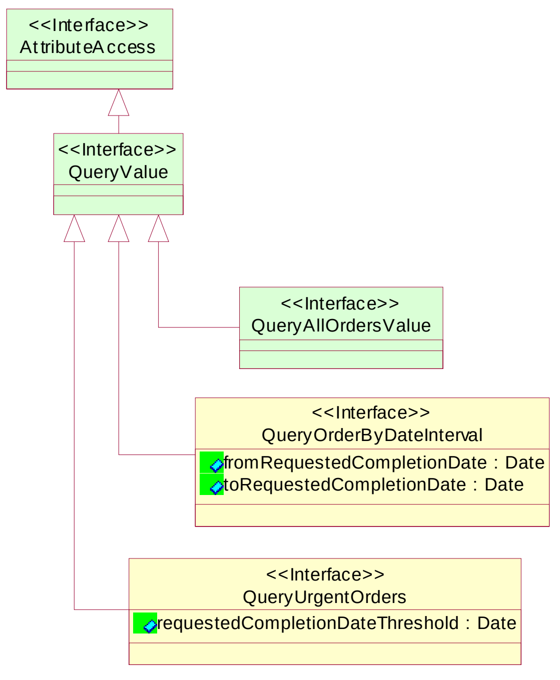


Figure 2 Query Types

3.4.1 Query Urgent Orders

This Query Value enables the client to retrieve all Orders, which requested delivery date have passed a threshold, sorted by priority and requested delivery date.

3.4.2 Query Orders By Date Interval

This Query Value enables the client to retrieve all Orders, which requested delivery date lies between a distinct interval, sorted by requested delivery date only.

3.5 Order Scheduler

Orders that have been started and have a populated requested delivery date are not finished directly be the Order Processor but are scheduled to be finished. The Scheduler assumes the order is processed in zero time and uses the Order Processor to finish the order as soon as possible after the requested delivery date has passed.

"As soon as possible" means, that there might be orders with higher priority, which get scheduled first over lower priority orders, so the actual delivery date might vary from the requested delivery date.

The state diagram below shows how the scheduler works internally. After connecting to the server it mainly performs two tasks:

1. It queries urgent orders from the JVT Activation Session Bean, i.e. all orders which requested delivery date has passed the current time, via the respective query value. It then processes these orders until all orders are finished, but only for a maximum amount of time, because while processing the urgent orders, new orders with high priority might step over the time threshold to be processed right now.
2. When there are no new orders to be processed, the scheduler queries the orders due in the next 300 seconds. A better solution would have been to get the next order due, but that is not possible, the reason being explained in chapter 3.6.1 (page 13, Querying Orders). When there is an order due in the near future, the scheduler waits until it is due or waits the full 300 seconds.

Beside that, the Scheduler listens to events originating from the JVT Activation Session Bean. If the Scheduler is in state "Wait" and receives an event that effects the order the Scheduler is waiting for right now, it notifies itself to wake up.

Of course this scheduling mechanism is plainly very basic and offers space for improvement.

Starting threads from EJBs is considered a big no-no. In order to start the Scheduler at application server startup in an application server independent fashion, i.e. Weblogic's startup classes are not an option, the thread is started within a Servlet's init method. The Servlet's whole purpose is to start that thread and can be configured to be initialized at application server startup. Through the Servlet's destroy method, the Order Processor is stopped, if the Enterprise Archive is redeployed or the server is goes down.

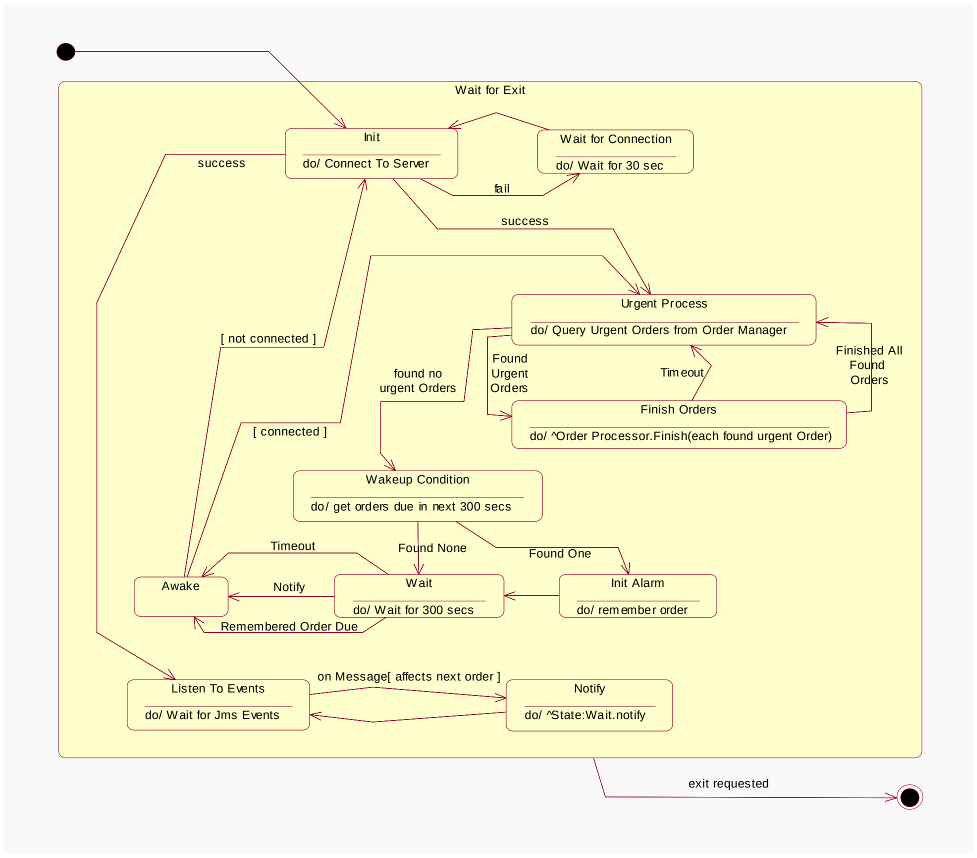


Figure 3 Order Scheduler State diagram

3.6 Order and Service

Orders and Services on the server side are the persistent version of the respective value types. They are implemented as CMP-2.0 EJBs (Container Managed Persistency Enterprise Java Beans) which means, that the Container takes care of writing the data to and reading it from the database. The relationship between an order and a service is realized as a container managed relationship.

The primary key for both EJBs is a CMPManagedEntityKey, which is taken care of in the super class of both Beans: TimestampedMEV. This class also defines an attribute which contains the last modification date.

Both Entity Beans, order and service, offer methods to set values from a java value type object and to create a JVT Object with certain attributes populated.

The Service EJB has a map which contains all attributes of new service types. This way all services can be stored in the same database table. The downside is that a template search for one of these attributes is really expensive, since **all** services have to be loaded into memory, the map has to be deserialized and then checked for that attribute.

3.6.1 Querying Orders

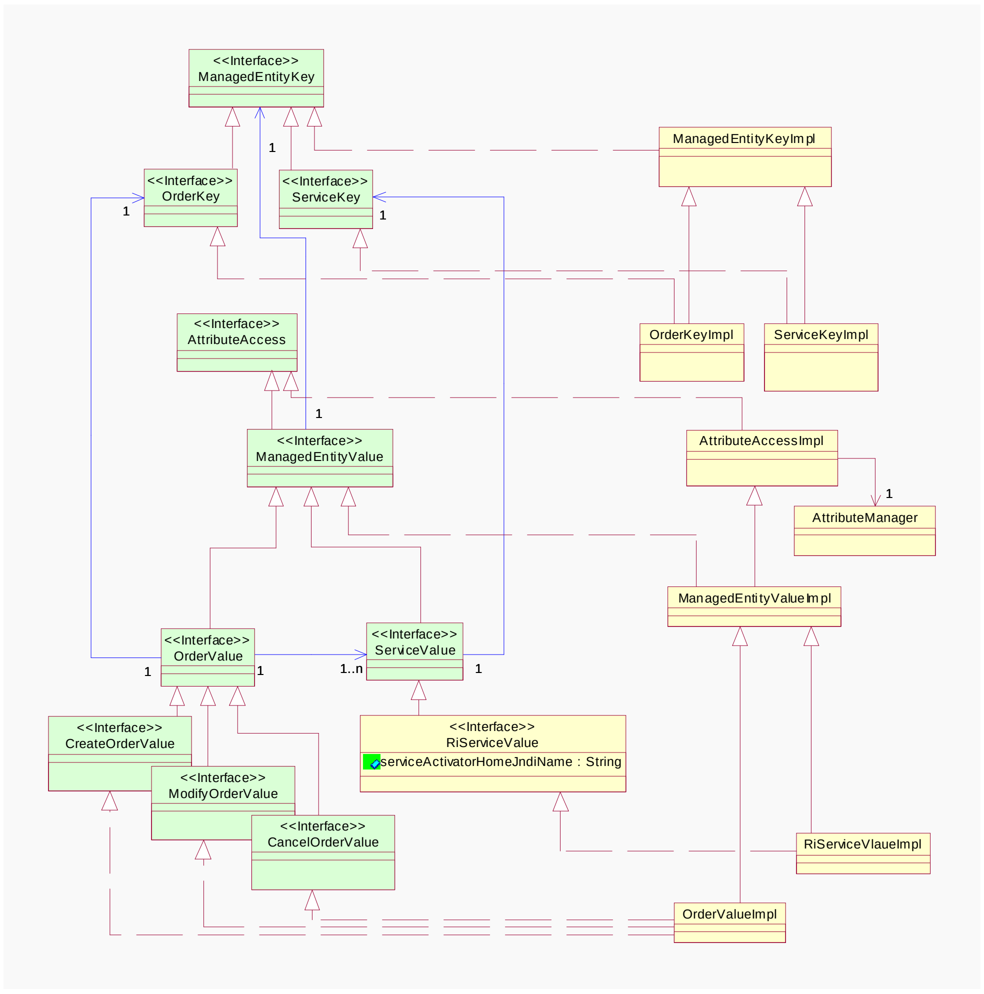
The client has multiple possibilities to query the JVT Activation Session Bean for existing orders. This chapter is going to explain how the queries are processed internally.

- **by Key:** The query by key is implemented via the findByPrimaryKey method, which is mandatory for all Entity EJBs.
- **by Query Value:** Both queries by query value are implemented as a finder method in the Order Entity EJB. Thus, the JVT Activation Session Bean plus the Query Value only are a wrapper around that.
- **by Template:** The template query is converted into a SQL statement directly in a home method of the Order Entity EJB. This could bring possible incompatibilities with it when the database schema or the database system itself changes.

Up to now the EJB-QL, which is used to define finder methods of Entity EJBs, is a very limited subset of SQL. A complete reference can be found at [EJB 01]. It does not offer options to limit the returned results to the maximum or the minimum matching row. With that limitation it was impossible to create a query which gets the next due order, as would be desired be the Order Scheduler. So there is only a query for order which are due between two dates, sorting them in ascending order, so the client then can take the first one.

4 Java Value Types

The Java Value Types are the objects, which are exchanged between the JVT Activation Session Bean and a client.



<<Interface>> ManagedEntityKey
ManagedEntityKeyImpl
<<Interface>> OrderKey
<<Interface>> ServiceKey
OrderKeyImpl
ServiceKeyImpl
<<Interface>> AttributeAccess
<<Interface>> ManagedEntityValue
AttributeAccessImpl
AttributeManager
ManagedEntityValueImpl
<<Interface>> OrderValue
<<Interface>> ServiceValue
<<Interface>> CreateOrderValue
<<Interface>> ModifyOrderValue
<<Interface>> CancelOrderValue
<<Interface>> RiServiceValue
serviceActivatorHomeJndiName : String
RiServiceVlaueImpl
OrderValueImpl
1
1..n

Figure 4 Java Value Types

The class ManagedEntityValueImpl is able to take care of all attribute related issues. It controls population state and changes in populated attributes via a dirty flag, all behaviour is either inhibited from AttributeAccessImpl, which deals with class wide attribute issues, or delegated to an AttributeManager, to handle attribute states different for each object. Thus minimizes the effort to create new JVTs.

All Services which can be handled by the Reference Implementation must implement the RiServiceValue interface. The defined attribute serviceActivatorHomeJndiName defines which Service Activator to call in order to handle this specific service. This attribute should be populated at creation time of such a service.

5 Database Schema

The Database Schema defines three tables:

5.1 Order Bean Table

The Order Bean Table stores all orders.

Column Name	Type	PK	Foreign PK
ACTUAL_COMPLETION_DATE	TIMESTAMP		
API_CLIENT_ID	VARCHAR(255)		
DESCRIPTION	VARCHAR(255)		
LAST_MODIFIED	TIMESTAMP		
MEV_PRIMARY_KEY	VARCHAR(255) NOT NULL	yes	
MEV_TYPE	VARCHAR(255) NOT NULL		
ORDER_DATE	TIMESTAMP		
PRIORITY	INTEGER NOT NULL		
PURCHASE_ORDER	VARCHAR(255)		
REQUESTED_COMPLETION_DATE	TIMESTAMP		
STATE	VARCHAR(255)		

5.2 Service Bean Table

The Service Bean Table stores all services.

Column Name	Type	PK	Foreign PK
LAST_MODIFIED	TIMESTAMP		
MEV_PRIMARY_KEY	VARCHAR(255) NOT NULL	yes	
MEV_TYPE	VARCHAR(255) NOT NULL		
ORDER_MEV_PRIMARY_KEY	VARCHAR(255)	yes	ORDERBEANTABLE. MEV_PRIMARY_KEY

Column Name	Type	PK	Foreign PK
POSITION_IN_ORDER_VALUE_ARRAY	INTEGER NOT NULL		
SUBSCRIBER_ID	VARCHAR(255)		
STATE	VARCHAR(255)		
SERVICE_ACTIVATOR_HOME_JNDI_NAME	VARCHAR(255)		
ADDITIONAL_ATTRIBUTES	LONG BIT VARYING		

5.3 Unique Key

The Unique Key table is used to generate a sequence of unique numbers. Since Cloudscape, the underlying database system, does not support sequences by default, the method as proposed in [CLOUDSCAPE 01] is used to generate unique numbers.

Column Name	Type	PK	Foreign PK
SEQUENCE	LONGINT DEFAULT 0		

6 Adding New Service Types

You can use the Service Activation Reference Implementation as a working application for your purpose: just add the services you need.

This goes as follows

1. First you have to create an Interface which is derived from the base interface for all services supported by the Reference Implementation: `com.nokia.oss.ossj.sa.ri.service.RiServiceValue`
2. Define Bean Properties in that Interface for all attributes you want to deal with
3. Create an implementation class of your service. The name of that class has to be the interface name + "Impl"! The implementing class may or may not be derived from classes already present in the Reference Implementation. Since they already do a great deal of the job concerning attribute handling you might want to do so.
4. Since the Service Activation specs require to have all attributes set on a service value when it is newly created, so that an activate order can immediately work with them, the attributes have to be populated at creation time of the value object.
5. Add the new service to the OrderManager's deployment descriptor. This can be found in `META-INF/ejb-jar.xml` in OrderManager.jar, which itself is contained in the Enterprise Archive `OssjSaRi.ear`. The name of the environment entry (`env-entry`) element is "`SupportedServices`" to which the new service has to be simply added. Whatever service is first in this list, is automatically added to a new activate order.
6. Deploy a Session Bean, which is responsible for creating, modifying and canceling that new service type. The Session Bean has to use the
    7. remote interface `com.nokia.oss.ossj.sa.ri.service.ServiceActivator` and the
    8. home interface `com.nokia.oss.ossj.sa.ri.ServiceActivatorHome`

7 Using the JVT Activation Session Bean

A client that wants to use the Service Activation API needs the following:

- The classpath has to include the following java archives:
  - **`ossjsa.jar`** – The archive containing the OSS/J service activation interfaces
  - classes, which are needed to use the services of the used application server. In case of Weblogic Server this is **`weblogic.jar`**.
- As default the JVT Activation Session Bean home remote interface is bound to the JNDI name: `Top.System.System1.ApplicationType.ServiceActivation.Application.1-0;1-0;ReferenceImplementation.Comp.JVTHome`
- As default the JMS Topic Factory is bound to the JNDI name: `Top.System.System1.ApplicationType.ServiceActivation.Application.1-0;1-0;ReferenceImplementation.Comp.TopicConnectionFactory`
- As default the JMS Event Topic is bound to the JNDI name: `Top.System.System1.ApplicationType.ServiceActivation.Application.1-0;1-0;ReferenceImplementation.Comp.JVTEventTopic`

Appendix A: Glossary and References

Glossary

References

[CLOUDSCAPE 01] "SQL-J Tips", Informix/IBM, 2001, http://www.cloudscape.com/support/doc_35/doc/html/coredocs/tricks.htm#797615

[EJB 01] "Enterprise JavaBeans™ Specification, Version 2.0", ejb-2_0-pfd2-spec.pdf, chapter 10, Sun Microsystems, 2001

[OSS/J 01] OSS through Java™ Initiative, 2001, http://java.sun.com/products/oss