JBoss Enterprise Application Platform Common Criteria Certification 5

Transactions Development Quick Start Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5 Transactions Development Quick Start Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Andrew Dinn
Red Hat
adinn@redhat.com

Mark Little
Red Hat
mlittle@redhat.com

Jonathan Halliday
Red Hat
jhallida@redhat.com

Edited by

Misty Stanley-Jones
Red Hat
misty@redhat.com

Abstract

This guide is a quick introduction aimed at Java developers who want to write applications using the JBoss Transaction Service APIs.

Table of Contents

CHAPTER 1. GETTING STARTED WITH JTA

This chapter summarizes the key features required to construct a *Java Transactions API (JTA)* application. If you are not familiar with the JTA, please begin by reading the first section of the *Transactions Development Guide*, provided as part of the Enterprise Application Platform documentation suite..

1.1. PACKAGE LAYOUT

Everything you need to write basic JTA applications is included in the Enterprise Application Platform. The key packages are detailed in Packages Relating to the JTA.

Packages Relating to the JTA

com.arjuna.ats.jts

Contains the JBoss Transaction Service implementation of the JTS and JTA *APIs (Application Programming Interfaces)*.

com.arjuna.ats.jta

Contains local and remote JTA implementation support.

com.arjuna.ats.jdbc

Contains transactional JDBC 2.0 support.

1.2. SETTING PROPERTIES

You can configure JBossJTA at runtime by setting various property attributes, either at run-time on the command line, or through a properties file. The initial properties file is located at **`$JBOSS_HOME/server/default/conf/jbossts-properties.xml`**.

1.2.1. Specifying the Object Store Location

JBossJTA uses an object store to persistently record the outcomes of transactions, to be used in the event of failures. To customize the location of the object store, you need to pass the location when you are executing the application, as shown in Example 1.1, “Specifying the Object Store”.

Example 1.1. Specifying the Object Store

```
   java –
Dcom.arjuna.ats.arjuna.objectstore.objectStoreDir=/location/of/objectsto
re myprogram
```

By default, the object store is located in a a directory beneath the current execution directory.

By default, all object states are stored within the **`defaultStore`** sub-directory of the object store root. You can change the sub-directory by setting the **`com.arjuna.ats.arjuna.objectstore.localOSRoot`** property variable.

1.3. DEMARCING TRANSACTIONS

The JBossJTA API consists of three elements:

- A high-level application transaction demarcation interface
- A high-level transaction manager interface intended for application server
- and a standard Java mapping of the X/Open XA protocol intended for transactional resource manager

All of the JTA classes and interfaces are located in the javax.transaction package, and the corresponding JBossJTA implementations in the com.arjuna.ats.jta package.

1.3.1. `UserTransaction`

The **`UserTransaction`** interface allows applications to control transaction boundaries.

You can obtain **`UserTransaction`** implementations via JNDI.

Example 1.2. Controlling Transactions

```
// Initialize the context and get UserTransaction
InitialContext ic = new InitialContext();
UserTransaction utx = ic.lookup("java:comp/UserTransaction")
// start transaction work..
utx.begin();
.. do work
utx.commit();
```

1.3.2. `TransactionManager`

The **`TransactionManager`** interface allows the application server to control transaction boundaries on behalf of the application being managed.

You can obtain **`TransactionManager`** implementations via JNDI.

```
// Initialize the context and get the TransactionManager
InitialContext ic = new InitialContext();
TransactinoManager utm = ic.lookup("java:/TransactionManager")
```

1.3.3. `Transaction`

The **`Transaction`** interface allows operations to be performed on the transaction associated with the target object. Every top-level transaction is associated with one **`Transaction`** object when the transaction is created. The **`Transaction`** object has several uses, as described in **`Transaction`** Interface uses.

`Transaction` Interface uses

- Enlists the transactional resources in use by the application.
- Register for transaction synchronization call backs.

- Commit or roll back the transaction.
- Obtain the status of the transaction.

You can obtain a **`Transaction`** object by invoking the **`getTransaction`** method of the **`TransactionManager`** interface, as shown in Example 1.3, “Obtaining a Transaction”.

Example 1.3. Obtaining a Transaction

```
Transaction txObj = TransactionManager.getTransaction();
```

1.4. LOCAL VERSUS DISTRIBUTED JTA IMPLEMENTATIONS

You should rely on the JTS/OTS specifications for transaction propagation among transaction managers, to ensure interoperability between JTA applications.

Procedure 1.1. Selecting the Local JTA Implementation

1. Set the om.arjuna.ats.jta.jtaTMImplementation property to **`com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple`**.
2. Set the com.arjuna.ats.jta.jtaUTImplementation to **`com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple`**.

Procedure 1.2. Selecting the Distributed JTA Implementation

1. Set the com.arjuna.ats.jta.jtaTMImplementation property to **`com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple`**.
2. Set the com.arjuna.ats.jta.jtaUTImplementation property to **`com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImple`**.

1.5. JDBC AND TRANSACTIONS

JBossJTA supports the construction of both local and distributed transactional applications which access databases using the JDBC 2.0 APIs. JDBC 2.0 supports *two-phase commit* of transactions, and is similar to the XA X/Open standard. The JDBC 2.0 support is found in the com.arjuna.ats.jdbc package.

JBossJTA incorporates JDBC connections within transactions by providing transactional JDBC drivers through which all interactions occur. These drivers intercept all invocations and ensure that they are registered with, and driven by, appropriate transactions. There is a single type of transactional driver through which any JDBC driver can be driven. This driver is com.arjuna.ats.jdbc.TransactionalDriver, and it implements the **`java.sql.Driver`** interface.

One way to establish the connection is through the **`java.sql.DriverManager.getConnection`** method. After establishing the connection, JBossJTA monitors all operations. You can use such connections in the same way as any other JDBC driver connection.

JBossJTA connections can be used within multiple different transactions simultaneously. Different threads, with different notions of the current transaction, may use the same JDBC connection. JBossJTA performs connection pooling for each transaction within the JDBC connection. Although multiple threads may use the same instance of the JDBC connection, internally a different connection instance may be used per transaction. With the exception of the `close` method, all operations performed on the connection at the application level are only performed on this transaction-specific connection.

JBossJTA automatically registers the JDBC driver connection with the transaction via an appropriate resource. When the transaction terminates, this resource either commits or rolls back any changes made to the underlying database, through appropriate calls on the JDBC driver.

1.6. CONFIGURABLE OPTIONS

Important Configurable Options shows the most important configuration features, along with possible values and defaults. For more information, consult the *Transactions Development Guide*.

Important Configurable Options

com.arjuna.ats.jta.supportSubtransactions

Possible Values

1. Yes (default)
2. No

com.arjuna.ats.jta.jtaTMImplementation

Possible Values

1. com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple
2. com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple

com.arjuna.ats.jta.jtaUTImplementation

Possible Values

1. com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple
2. com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImple

com.arjuna.ats.jta.xaBackoffPeriod

Possible Values

1. Time in seconds

com.arjuna.ats.jdbc.isolationLevel

Possible Values

1. Any supported JDBC isolation level

CHAPTER 2. GETTING STARTED WITH JTS / OTS

This chapter discusses the key features required to construct a basic *OTS (Object Transaction Service)* application using the raw OTS interfaces defined by the *Object Management Group (OMG)* specification. This work focuses on implementation details. Refer to the *Transactions Development Guide* for a conceptual overview.

2.1. PACKAGE LAYOUT

Table 2.1. Important Packages Needed To Create OTS Applications

Package	Description
com.arjuna.orbportability	this package contains the classes which constitute the ORB portability library and other useful utility classes.
org.omg.CosTransactions	this package contains the classes which make up the CosTransactions.idl module.
com.arjuna.ats.jts	this package contains the JBoss Transaction Service implementations of the JTS and JTA.
com.arjuna.ats.arjuna	this package contains further classes necessary for the JBoss Transaction Service implementation of the JTS.
com.arjuna.ats.jta	this package contains local and remote JTA implementation support.
com.arjuna.ats.jdbc	this package contains transactional JDBC 2.0 support.

2.2. SETTING PROPERTIES

JBoss Transaction Service is configurable at runtime through the use of various property attributes, which are described in subsequent sections. You can provide these attributes at run-time on the command line. However, it often more convenient to specify them through the **`jbossts-properties.xml`**, which may be in any of the locations mentioned, in search order, in Possible Locations of the **`jbossts-properties.xml`** File. properties file.

Possible Locations of the `jbossts-properties.xml` File

1. The current working directory.
2. The home directory of the executing user.
3. The **`CLASSPATH`**, by means of the **`getResource`** method.

If the properties file is found, all entries within it added to the system properties, and override the defaults. You can also specify other properties not specific to the Transaction Service.

2.3. STARTING AND STOPPING THE ORB AND BOA/POA

BOA refers to *Basic Object Adapter*, and POA refers to *Portable Object Adepter*.

JBoss Transaction Service needs to be correctly initialized before any application object is created. To guarantee this, you must use the **`initORB`** method, and either of the **`initBOA`** or **`initPOA`** methods of the **`ORBInterface`** class, which is described in the ORB Portability Manual. Do not use the **`ORB_init`**, **`BOA_init`**, or **`create_POA`** methods provided by the underlying ORB, because they may lead to incorrectly operating applications.

Example 2.1. ORB Initialization

```
public static void main (String[] args)
{
    ORBInterface.initORB(args, null);
    ORBInterface.initOA();
    . . .
};
```

`ORBInterface` Methods

`orb`

Returns references to the ORB

`boa`

Returns references to the BOA

`poa`

Returns references to the POA

`rootPoa`

Returns references to the root POA

`shutdownOA`

Shut down the BOA. Run this before **`shutdownORB`**, and before terminating the application.

`shutdownORB`

Shut down the ORB. Use this after **`shutdownOA`**. Run this before terminating the application.

Use the **`shutdownOA`** and **`shutdownORB`** methods, in sequence, before terminating an application. This allows JBoss Transaction Service to perform necessary cleanup routines. The **`shutdownOA`** routine either shuts down the BOA or the POA, depending upon the ORB being used.

Example 2.2. Shutting Down the ORB

```
public static void main (String[] args)
{
    . . .
```

```
        ORBInterface.shutdownOA();
        ORBInterface.shutdownORB();
};
```

Do not use more CORBA objects after you call **`shutdown`**. You need to reinitialize the BOA/POA and ORB before using more CORBA objects.

NOTE

The term *Object Adapter* is used in the rest of this guide to refer to either the BOA or the POA, interchangeably. Where possible, this guide uses the ORB Portability classes to mask the differences between POA and BOA.

2.4. SPECIFYING THE OBJECT STORE LOCATION

JBoss Transaction Service uses an object store to persistently record the outcomes of transactions, for failure recovery. You can specify the location of the object store using the **`objectStoreDir`** property.

Example 2.3. Specifying the Object Store at Application Execution

```
 java
Dcom.arjuna.ats.arjuna.objectstore.objectStoreDir=/var/tmp/ObjectStore
myprogram
```

By default, the object store is located in a directory under the current execution directory. In the default configuration, all object states are stored within the **`defaultStore`**. However, this sub-directory can be changed by setting the com.arjuna.ats.arjuna.objectstore.localOSRoot property variable.

2.5. IMPLICIT TRANSACTION PROPAGATION AND INTERPOSITION

You can create transactions within one domain and use them within another. Therefore, information about a transaction, called the *transaction context*, needs to be propagated between these domains.

Propagating the Transaction Context

Explicit propagation

An application passes context objects as explicit parameters. These objects are either instances of the **`Control`** interface or the **`PropagationContext`** structure, and are defined by the Transaction Service. It is more efficient to use the **`PropagationContext`** structure, rather than the **`Control`** interface.

Implicit propagation

Requests on objects are implicitly associated with the client’s transaction, and share the client’s transaction context. The context is transmitted implicitly to the objects, without direct client intervention.

OTS objects supporting the Control interface are standard CORBA objects. When the interface is passed as a parameter in an operation call to a remote server, only an object reference is passed. Any operations that the remote object performs on the interface are performed on the real object.

This behavior can impose substantial penalties on an application which frequently uses these interfaces due to the overheads of remote invocation. To avoid this overhead, JBoss Transaction Service supports *interposition*. In interposition, the server creates a local object which acts as a proxy for the remote transaction, fielding all requests that would normally be passed back to the originator. This local object registers itself with the original transaction coordinator, so that it can correctly participate in the termination of the transaction. Interposed coordinators effectively form a tree structure with their parent coordinators, as shown in Figure 2.1, “Interposition”.

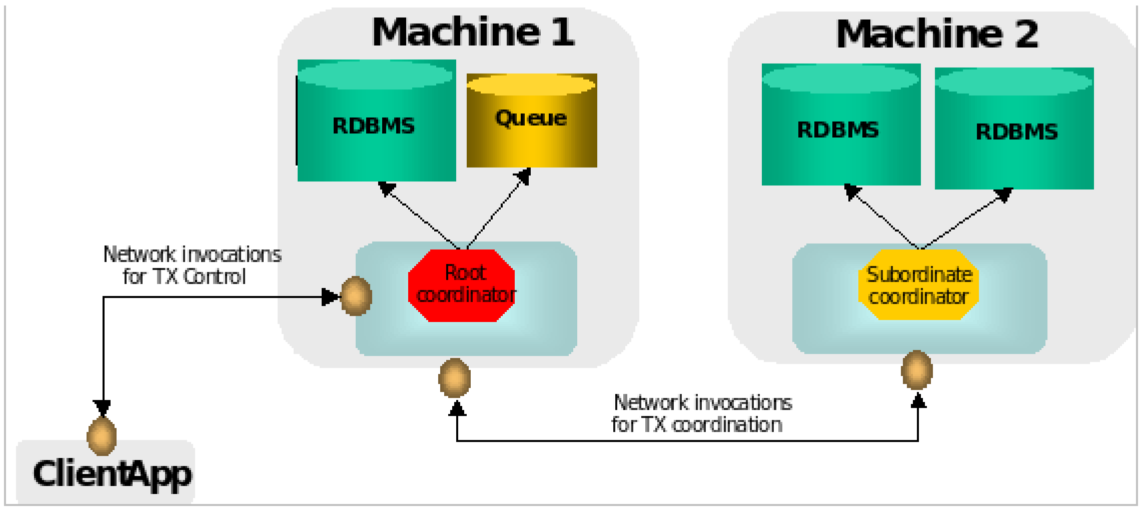

Figure 2.1. Interposition

NOTE

implicit transaction propagation does not imply that interposition is also used at the server. Instead, interposition typically requires implicit propagation.

If you require implicit context propagation and interposition, ensure that JBoss Transaction Service is correctly initialized before creating any objects. The client and server need to agree whether implicit propagation or interposition, or neither, is used. Implicit context propagation is only possible on those ORBs which support filters and interceptors, or which support the **`CosTSPortability`** interface. JacORB and the JDK miniORB both provide the required support.

Enabling Propagation

Implicit context propagation

Set the com.arjuna.ats.jts.contextPropMode property variable to **`CONTEXT`**.

Interposition

Set the com.arjuna.ats.jts.contextPropMode property variable to **`INTERPOSITION`**.

NOTE

To use the JBoss Transaction Service advanced API, you must use interposition.

2.6. OBTAINING CURRENT

You can obtain the **`Current`** pseudo-object from the **`com.arjuna.ats.jts.OTSManager`** class by using its **`get_current`** method.

2.7. TRANSACTION TERMINATION

How long a **`Control`** can access a terminated transaction is implementation-specific. In JBoss Transaction Service, if you are using the **`Current`** pseudo-object, all information about a transaction is destroyed when it terminates. For this reason, you should not use any **`Control`** references to the transaction after issuing the commit or rollback operation.

However, if you terminate the transaction explicitly, using the **`Terminator`** interface, information about the transaction is only removed when all the outstanding references to it have been destroyed. You can signal that the transaction information is no longer required, by using the **`destroyControl`** method of the OTS class, which is found in the com.arjuna.CosTransactions package. After the program indicates that the transaction information is no longer required, you should not use any **`Control`** references to the transaction.

2.8. TRANSACTION FACTORY

By default, JBoss Transaction Service does not use a separate transaction manager when creating transactions through the **`Current`** interface. Each transactional client essentially has its own transaction manager, the **`TransactionFactory`**, which is co-located with it. To override this behavior at run-time, set the com.arjuna.ats.jts.transactionManager property variable to **`YES`**. To execute the Transaction Factory, execute the **`start-transaction-service`** script, located in the ***`ATS_ROOT`*****`/bin`** directory.

`Current` typically locates the factory using the **`CosServices.cfg`** file located in the ***`$JBOSS_HOME`*****`/etc`** directory. This file is similar to the **`resolve_initial_references`** file, and is automatically created or updated when the transaction factory is started on a particular machine. This file must be copied locally to each machine which needs to share the same transaction factory.

NOTE

The information about **`CosServices.cfg`** refers to the default name and location of the configuration file. To change the name of the file, use the com.arjuna.orbportability.initialReferencesFile variable. To change its location, set the com.arjuna.orbportability.initialReferencesRoot variable.

Example 2.4. Customizing the Initial References File

```
 java –Dcom.arjuna.orbportability.initialReferencesFile=ref –
Dcom.arjuna.orbportability.initialReferencesRoot=c:\\temp prog
```

You can override the default location mechanism by setting the com.arjuna.orbportability.resolveService property variable with any of the parameters listed in ResolveService Parameters.

ResolveService Parameters

CONFIGURATION_FILE

The system uses the **`CosServices.cfg`** file. This is the default behavior.

NAME_SERVICE

JBoss Transaction Service attempts to use a name service to locate the transaction factory. If this is not supported, an exception is thrown.

BIND_CONNECT

JBoss Transaction Service uses the ORB-specific bind mechanism. If this is not supported, an exception is thrown.

If com.arjuna.orbportability.resolveService is specified when the transaction factory is run, the factory registers itself with the specified resolution mechanism.

2.9. RECOVERY MANAGER

You need to start the recovery manager subsystem to ensure that transactions are recoverable in the event of a failure. To start the recovery manager, run the **`start-recovery-manager`** script in ***`$ATS_ROOT`*****`/bin`**.

CHAPTER 3. GETTING STARTED WITH WEB SERVICES TRANSACTIONS AND XTS

3.1. CONFIGURING THE WEB SERVICES COMPONENT

For in-depth information about JBoss Transactions XTS, see the XTS section of the *Transactions Development Guide*, which ships as part of the documentation suite for the Enterprise Application Platform.

Table 3.1. Web Services Configuration

Property	Possible Values
com.arjuna.orbportability.initialReferencesFile	CosServices.cfg
com.arjuna.orbportability.initialReferencesRoot	The directory containing the file arjuna.properties.
ArjunaJTS_LicenceKey	System specific license.
com.arjuna.orbportability.resolveService	CONFIGURATION_FILE NAME_SERVICE BIND_CONNECT
com.arjuna.ats.arjuna.objectstore.objectStoreDir	Any location that the application can write to.
com.arjuna.ats.arjuna.objectstore.localOSRoot	defaultStore
PROPERTIES_FILE	arjuna.properties
com.arjuna.ats.arjuna.coordinator.asyncPrepare	YES/NO
com.arjuna.ats.arjuna.coordinator.asyncCommit	YES/NO
com.arjuna.ats.arjuna.coordinator.commitOnePhase	YES/NO
com.arjuna.ats.arjuna.coordinator.transactionSync	ON/OFF
com.arjuna.ats.arjuna.coordinator.enableStatistics	ON/OFF
com.arjuna.ats.jts.alwaysPropagateContext	YES/NO
com.arjuna.ats.jts.defaultTimeout	No timeout
com.arjuna.ats.jts.supportRollbackSync	YES/NO
com.arjuna.ats.jts.supportInterposedSynchronization	YES/NO
com.arjuna.ats.jts.supportSubtransactions	YES/NO

Property	Possible Values
com.arjuna.ats.jts.checkedTransactions	YES/NO
com.arjuna.ats.jts.transactionManager	YES/NO
com.arjuna.ats.jts.needTranContext	YES/NO
com.arjuna.ats.arjuna.coordinator.txReaperTimeout	120000000 microseconds
com.arjuna.ats.arjuna.coordinator.txReaperMode	NORMAL DYNAMIC
com.arjuna.ats.jts.contextPropMode	NONE CONTEXT INTERPOSITION

INDEX