JBoss Transactions 4.16

Development Guide

Development reference guide for the JBoss Transactions suite of software

Mark Little

Jonathan Halliday

Andrew Dinn

Kevin Connor

JBoss Transactions 4.16 Development Guide
Development reference guide for the JBoss Transactions suite of software
Edition 0

Author	Mark Little	*mlittle@redhat.com*
Author	Jonathan Halliday	*jhallida@redhat.com*
Author	Andrew Dinn	*adinn@redhat.com*
Author	Kevin Connor	*kconnor@redhat.com*
Editor	Misty Stanley-Jones	*misty@redhat.com*

Development reference guide for the JBoss Transactions suite of software

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

> **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at *http://access.redhat.com*. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).

- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **Fedora** and the component **jboss-jts**. The following link will take you to a pre-filled bug report for this product: https://bugzilla.redhat.com.

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

```
Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:
```

Be sure to give us your name so that you can receive full credit for reporting the issue.

About This Guide

The Programmers Guide contains information on how to use JBoss Transactions. This document provides a detailed look at the design and operation of JBoss Transactions. It describes the architecture and the interaction of components within this architecture.

1.1. Audience

This guide is most relevant to engineers who are responsible for developing using JBoss Transactions. Although this guide is specifically intended for service developers, it will be useful to anyone who would like to gain an understanding of transactions and how they function.

1.2. Prerequisites

This guide assumes a basic familiarity with Java service development and object-oriented programming. A fundamental level of understanding in the following areas will also be useful:

- General understanding of the APIs, components, and objects that are present in Java applications.
- A general understanding of the Windows and UNIX operating systems.

Transactions

A transaction is a unit of work that encapsulates multiple database actions such that that either all the encapsulated actions fail or all succeed.

Transactions ensure data integrity when an application interacts with multiple datasources.

2.1. The Java Transaction API (JTA)

The interfaces specified by the many transaction standards tend to be too low-level for most application programmers. Therefore, Sun Microsystems created the Java Transaction API (JTA), which specifies higher-level interfaces to assist in the development of distributed transactional applications.

Note, these interfaces are still low-level. You still need to implement state management and concurrency for transactional applications. The interfaces are also optimized for applications which require XA resource integration capabilities, rather than the more general resources which other transactional APIs allow.

With reference to JTA 1.1 (*http://www.oracle.com/technetwork/java/javaee/tech/jta-138684.html*), distributed transaction services typically involve a number of participants:

application server	provides the infrastructure required to support the application run-time environment which includes transaction state management, such as an EJB server.
transaction manager	provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.
resource manager	Using a *resource adapter* , provides the application with access to resources. The resource manager participates in distributed transactions by implementing a transaction resource interface used by the transaction manager to communicate transaction association, transaction completion and recovery. A resource adapter is used by an application server or client to connect to a Resource Manager. JDBC drivers which are used to connect to relational databases are examples of Resource Adapters.
communication resource manager	supports transaction context propagation and access to the transaction service for incoming and outgoing requests.

From the point of view of the transaction manager, the actual implementation of the transaction services does not need to be exposed. You only need to define high-level interfaces to allow transaction demarcation, resource enlistment, synchronization and recovery process to be driven from the users of the transaction services. The JTA is a high-level application interface that allows a

transactional application to demarcate transaction boundaries, and also contains a mapping of the X/Open XA protocol.

Compatibility

the JTA support provided by JBoss Transactions is compliant with the 1.1 specification.

2.2. Introducing the API

The Java Transaction API consists of three elements:

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

All of the JTA classes and interfaces exist within the *javax.transaction* package, and the corresponding JBoss Transactions implementations within the *com.arjuna.ats.jta* package.

Each Xid created by JBoss Transactions needs a unique node identifier encoded within it, because JBoss Transactions can only recover transactions and states that match a specified node identifier. The node identifier to use should be provided to JBoss Transactions via the `CoreEnvironmentBean.nodeIdentifier` property. This value must be unique across your JBoss Transactions instances. The identifier is alphanumeric and limited to 10 bytes in length. If you do not provide a value, then JBoss Transactions generates one and reports the value via the logging infrastructure.

2.3. UserTransaction

The `UserTransaction` interface provides applications with the ability to control transaction boundaries. It provides methods `begin` , `commit` , and `rollback` to operate on top-level transactions.

Nested transactions are not supported, and method `begin` throws the exception `NotSupportedException` if the calling thread is already associated with a transaction. `UserTransaction` automatically associates newly created transactions with the invoking thread.

To obtain a `UserTransaction` , call the static method `com.arjuna.ats.jta.UserTransaction.userTransaction()` .

Procedure 2.1. Selecting the local JTA Implementation

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

2.4. TransactionManager

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

To obtain a `TransactionManager` , invoke the static method `com.arjuna.ats.jta.TransactionManager.transactionManager` .

The `TransactionManager` maintains the transaction context association with threads as part of its internal data structure. A thread's transaction context may be **`null`** or it may refer to a specific global transaction. Multiple threads may be associated with the same global transaction. As noted in *Section 2.3, "UserTransaction"* , nested transactions are not supported.

Each transaction context is encapsulated by a Transaction object, which can be used to perform operations which are specific to the target transaction, regardless of the calling thread's transaction context.

Table 2.1. `TransactionManager` Methods

`begin`	Starts a new top-level transaction and associates the transaction context with the calling thread. If the calling thread is already associated with a transaction, exception `NotSupportedException` is thrown.
`getTransaction`	Returns the Transaction object representing the transaction context which is currently associated with the calling thread. You can use this object to perform various operations on the target transaction.
`commit`	Completes the transaction currently associated with the calling thread. After it returns, the calling thread is associated with no transaction. If `commit` is called when the thread is not associated with any transaction context, an exception is thrown. In some implementations, the `commit` operation is restricted to the transaction originator only. If the calling thread is not allowed to commit the transaction, an exception is thrown. JBoss Transactions does not currently impose any restriction on the ability of threads to terminate transactions.
`rollback`	Rolls back the transaction associated with the current thread. After the `rollback` method completes, the thread is associated with no transaction.

In a multi-threaded environment, multiple threads may be active within the same transaction. If checked transaction semantics have been disabled, or the transaction times out, a transaction may terminated by a thread other than the one that created it. In this case, the creator usually needs to be notified. JBoss Transactions notifies the creator during operations `commit` or `rollback` by throwing exception `IllegalStateException` .

2.5. Suspend and resuming a transaction

The JTA supports the concept of a thread temporarily suspending and resuming transactions in order to perform non-transactional work. Call the `suspend` method to temporarily suspend the current transaction that is associated with the calling thread. The thread then operates outside of the scope of the transaction. If the thread is not associated with any transaction, a null object reference is returned.

Otherwise, a valid Transaction object is returned. Pass the Transaction object to the `resume` method to reinstate the transaction context.

The `resume` method associates the specified transaction context with the calling thread. If the transaction specified is not a valid transaction, , the thread is associated with no transaction. if `resume` is invoked when the calling thread is already associated with another transaction, the `IllegalStateException` exception is thrown.

Example 2.1. Using the `suspend` method

```
Transaction tobj = TransactionManager.suspend();
..
TransactionManager.resume(tobj);
```

Note

JBoss Transactions allows a suspended transaction to be resumed by a different thread. This feature is not required by JTA, but is an important feature.

When a transaction is suspended, the application server must ensure that the resources in use by the application are no longer registered with the suspended transaction. When a resource is de-listed this triggers the Transaction Manager to inform the resource manager to disassociate the transaction from the specified resource object. When the application's transaction context is resumed, the application server must ensure that the resources in use by the application are again enlisted with the transaction. Enlisting a resource as a result of resuming a transaction triggers the Transaction Manager to inform the resource manager to re-associate the resource object with the resumed transaction.

2.6. The Transaction interface

The `Transaction` interface allows you to perform operations on the transaction associated with the target object. Every top-level transaction is associated with one Transaction object when the transaction is created.

Uses of the Transaction object

- enlist the transactional resources in use by the application.
- register for transaction synchronization call backs.
- commit or rollback the transaction.
- obtain the status of the transaction.

The `commit` and `rollback` methods allow the target object to be committed or rolled back. The calling thread does not need to have the same transaction associated with the thread. If the calling thread is not allowed to commit the transaction, the transaction manager throws an exception. At present JBoss Transactions does not impose restrictions on threads terminating transactions.

The JTA standard does not provide a means to obtain the transaction identifier. However, JBoss Transactions provides several ways to view the transaction identifier. Call method `toString` to print full information about the transaction, including the identifier. Alternatively you can cast the javax.transaction.Transaction instance to a com.arjuna.ats.jta.transaction.Transaction , then call either

method `get_uid` , which returns an ArjunaCore Uid representation, or `getTxId` , which returns an Xid for the global identifier, i.e., no branch qualifier.

2.7. Resource enlistment

Typically, an application server manages transactional resources, such as database connections, in conjunction with some resource adapter and optionally with connection pooling optimization. For an external transaction manager to coordinate transactional work performed by the resource managers, the application server must enlist and de-list the resources used in the transaction. These resources, called *participants* , are enlisted with the transaction so that they can be informed when the transaction terminates, by being driven through the two-phase commit protocol.

As stated previously, the JTA is much more closely integrated with the XA concept of resources than the arbitrary objects. For each resource the application is using, the application server invokes the `enlistResource` method with an XAResource object which identifies the resource in use.

The enlistment request causes the transaction manager to inform the resource manager to start associating the transaction with the work performed through the corresponding resource. The transaction manager passes the appropriate flag in its `XAResource.start` method call to the resource manager.

The `delistResource` method disassociates the specified resource from the transaction context in the target object. The application server invokes the method with the two parameters: the XAResource object that represents the resource, and a flag to indicate whether the operation is due to the transaction being suspended (**`TMSUSPEND`**), a portion of the work has failed (**`TMFAIL`**), or a normal resource release by the application (**`TMSUCCESS`**).

The de-list request causes the transaction manager to inform the resource manager to end the association of the transaction with the target XAResource . The flag value allows the application server to indicate whether it intends to come back to the same resource whereby the resource states must be kept intact. The transaction manager passes the appropriate flag value in its `XAResource.end` method call to the underlying resource manager.

2.8. Transaction synchronization

Transaction synchronization allows the application server to be notified before and after the transaction completes. For each transaction started, the application server may optionally register a Synchronization call-back object to be invoked by the transaction manager, which will be one of the following:

`beforeCompletion`	Called before the start of the two-phase transaction complete process. This call is executed in the same transaction context of the caller who initiates the `TransactionManager.commit` or the call is executed with no transaction context if `Transaction.commit` is used.
`afterCompletion`	Called after the transaction completes. The status of the transaction is supplied in the parameter. This method is executed without a transaction context.

2.9. Transaction equality

The transaction manager implements the Transaction object's `equals` method to allow comparison between the target object and another Transaction object. The `equals` method returns **`true`** if the target object and the parameter object both refer to the same global transaction.

Example 2.2. Method `equals`

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

boolean isSame = txObj.equals(someOtherTxObj);
```

2.10. TransactionSynchronizationRegistry

The `javax.transaction.TransactionSynchronizationRegistry` interface, added to the JTA API in version 1.1, provides for registering Synchronizations with special ordering behavior, and for storing key-value pairs in a per-transaction Map. Full details are available from the JTA 1.1 API specification and javadoc. Here we focus on implementation specific behavior.

Example 2.3. Accessing the TransactionSynchronizationRegistry in standalone environments

```
javax.transaction.TransactionSynchronizationRegistry tsr = new
 com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionSynchronizationRegistryImple();
```

This is a stateless object and hence is cheap to instantiate.

Accessing the TransactionSynchronizationRegistry via JNDI

In application server environments, the standard JNDI name binding is **`java:comp/TransactionSynchronizationRegistry`** .

Ordering of interposed Synchronizations is relative to other local Synchronizations only. In cases where the transaction is distributed over multiple JVMs, global ordering is not guaranteed.

The per-transaction data storage provided by the `TransactionSynchronizationRegistry` methods `getResource` and `putResource` are non-persistent and thus not available in `Transactions` during crash recovery. When running integrated with an application server or other container, this storage may be used for system purposes. To avoid collisions, use an application-specific prefix on map keys, such as **`put("myapp_"+key, value)`** . The behavior of the Map on Thread s that have status **`NO_TRANSACTION`** or where the transaction they are associated with has been rolled back by another Thread , such as in the case of a timeout, is undefined. A Transaction can be associated with multiple Thread s. For such cases the Map is synchronized to provide thread safety.

The Resource Manager

3.1. The XAResource interface

Some transaction specifications and systems define a generic resource which can be used to register arbitrary resources with a transaction, the JTA is much more XA-specific. Interface `javax.transaction.xa.XAResource` is a Java mapping of the XA interface. The `XAResource` interface defines the contract between a `ResourceManager` and a `TransactionManager` in a distributed transaction processing environment. A resource adapter for a `ResourceManager` implements the `XAResource` interface to support association of a top-level transaction to a resource such as a relational database.

The `XAResource` interface can be supported by any transactional resource adapter designed to be used in an environment where transactions are controlled by an external transaction manager, such a database management system. An application may access data through multiple database connections. Each database connection is associated with an `XAResource` object that serves as a proxy object to the underlying `ResourceManager` instance. The transaction manager obtains an `XAResource` for each `ResourceManager` participating in a top-level transaction. The `start` method associates the transaction with the resource, and the `end` method disassociates the transaction from the resource.

The `ResourceManager` associates the transaction with all work performed on its data between invocation of `start` and `end` methods. At transaction commit time, these transactional `ResourceManagers` are informed by the transaction manager to prepare, commit, or roll back the transaction according to the two-phase commit protocol.

For better Java integration, the `XAResource` differs from the standard XA interface in the following ways:

- The resource adapter implicitly initializes the `ResourceManager` when the resource (the connection) is acquired. There is no equivalent to the `xa_open` method of the interface XA.
- `Rmid` is not passed as an argument. Each `Rmid` is represented by a separate `XAResource` object.
- Asynchronous operations are not supported, because Java supports multi-threaded processing and most databases do not support asynchronous operations.
- Error return values caused by the transaction manager's improper handling of the `XAResource` object are mapped to Java exceptions via the **`XAException`** class.
- The DTP concept of Thread of Control maps to all Java threads that are given access to the `XAResource` and `Connection` objects. For example, it is legal for two different threads to perform the `start` and `end` operations on the same `XAResource` object.

3.1.1. Extended XAResource control

By default, whenever an `XAResource` object is registered with a JTA-compliant transaction service, there is no way to manipulate the order in which it is invoked during the two-phase commit protocol, with respect to other `XAResource` objects. JBoss Transactions, however, provides support for controlling the order via the two interfaces `com.arjuna.ats.jta.resources.StartXAResource` and `com.arjuna.ats.jta.resources.EndXAResource`. By inheriting your `XAResource` instance from either of these interfaces, you control whether an instance of your class is invoked first or last, respectively.

Note

Only one instance of each interface type may be registered with a specific transaction.

The *ArjunaCore Development Guide* discusses the *Last Resource Commit optimization (LRCO)*, whereby a single resource that is only one-phase aware, and does not support the `prepare` phase, can be enlisted with a transaction that is manipulating two-phase aware participants. This optimization is also supported within the JBoss Transactions.

In order to use the LRCO, your XA`Resource` implementation must extend the `com.arjuna.ats.jta.resources.LastResourceCommitOptimisation` marker interface. A marker interface is an interface which provides no methods. When enlisting the resource via method `Transaction.enlistResource`, JBoss Transactions ensures that only a single instance of this type of participant is used within each transaction. Your resource is driven last in the commit protocol, and no invocation of method `prepare` occurs.

By default an attempt to enlist more than one instance of a LastResourceCommitOptimisation class will fail and false will be returned from Transaction.enlistResource. This behavior can be overridden by setting the com.arjuna.ats.jta.allowMultipleLastResources to true. However, before doing so you should read the section on enlisting multiple one-phase aware resources.

Important

You need to disable interposition support to use the LCRO in a distributed environment. You can still use implicit context propagation.

3.1.1.1. Enlisting multiple one-phase-aware resources

One-phase commit is used to process a single one-phase aware resource, which does not conform to the two-phase commit protocol. You can still achieve an atomic outcome across resources, by using the LRCO, as explained earlier.

Multiple one-phase-aware resources may be enlisted in the same transaction. One example is when a legacy database runs within the same transaction as a legacy JMS implementation. In such a situation, you cannot achieve atomicity of transaction outcome across multiple resources, because none of them enter the `prepare` state. They commit or roll back immediately when instructed by the transaction coordinator, without knowledge of other resource states and without a way to undo if subsequent resources make a different choice. This can result in data corruption or heuristic outcomes.

You can approach these situations in two different ways:

- Wrap the resources in compensating transactions. See the *XTS Transactions Development Guide* for details.
- Migrate the legacy implementations to two-phase aware equivalents.

If neither of these options is viable, JBoss Transactions support enlisting multiple one-phase aware resources within the same transaction, using LRCO, which is discussed in the *ArjunaCore Development Guide* in detail.

Warning

Even when this support is enabled, JBoss Transactions issues a warning when it detects that the option has been enabled: **`You have chosen to enable multiple last resources in the transaction manager. This is transactionally unsafe and should not be relied upon.`** Another warning is issued when multiple one-phase aware resources are enlisted within a transaction: **`This is transactionally unsafe and should not be relied on.`**

To override the above-mentioned warning at runtime, set the `CoreEnvironmentBean.disableMultipleLastResourcesWarning` property to **`true`**. You will see a warning that you have done this when JBoss Transactions starts up and see the warning about enlisting multiple one-phase resources only the first time it happens, but after that no further warnings will be output. You should obviously only consider changing the default value of this property (false) with caution.

3.2. Opening a resource manager

The X/Open XA interface requires the transaction manager to initialize a resource manager, using method `xa_open`, before invoking any other of the interface's methods. JTA requires initialization of a resource manager to be embedded within the resource adapter that represents the resource manager. The transaction manager does not need to know how to initialize a resource manager. It only informs the resource manager about when to start and end work associated with a transaction and when to complete the transaction. The resource adapter opens the resource manager when the connection to the resource manager is established.

3.3. Closing a resource manager

The resource adapter closes a resource manager as a result of destroying the transactional resource. A transaction resource at the resource adapter level is comprised of two separate objects:

- An `XAResource` object that allows the transaction manager to start and end the transaction association with the resource in use and to coordinate transaction completion process.
- A connection object that allows the application to perform operations on the underlying resource, such as JDBC operations on an RDBMS.

Once opened, the resource manager is kept open until the resource is released explicitly. When the application invokes the connection's `close` method, the resource adapter invalidates the connection object reference that was held by the application and notifies the application server about the close. The transaction manager invokes the `XAResource.end` method to disassociate the transaction from that connection.

The close notification triggers the application server to perform any necessary cleanup work and to mark the physical XA connection as free for reuse, if connection pooling is in place.

3.4. Thread of control

The X/Open XA interface specifies that the transaction-association-related `xa` calls must be invoked from the same thread context. This *thread-of-control* requirement does not apply to the object-oriented component-based application run-time environment, in which application threads are dispatched dynamically as methods are invoked.. Different threads may use the same connection resource to access the resource manager if the connection spans multiple method invocation. Depending on the implementation of the application server, different threads may be involved with the same `XAResource` object. The resource context and the transaction context operate independent of thread context. This creates the possibility of different threads invoking the `start` and `end` methods.

If the application server allows multiple threads to use a single `XAResource` object and the associated connection to the resource manager, the application server must ensure that only one transaction context is associated with the resource at any point of time. Thus the `XAResource` interface requires the resource managers to support the two-phase commit protocol from any thread context.

3.5. Transaction association

A transaction is associated with a transactional resource via the `start` method and disassociated from the resource via the `end` method. The resource adapter internally maintains an association between the resource connection object and the `XAResource` object. At any given time, a connection is associated with zero or one transaction. JTA does not support nestedtransactions, so attempting to invoke the `start` method on a thread that is already associated with a transaction is an error.

The transaction manager can Interleave multiple transaction contexts using the same resource, as long as methods `start` and `end` are invoked properly for each transaction context switch. Each time the resource is used with a different transaction, the method `end` must be invoked for the previous transaction that was associated with the resource, and method `start` must be invoked for the current transaction context.

3.6. Externally controlled connections

For a transactional application whose transaction states are managed by an application server, its resources must also be managed by the application server so that transaction association is performed properly. If an application is associated with a transaction, the application must not perform transactional work through the connection without having the connection's resource object already associated with the global transaction. The application server must ensure that the `XAResource` object in use is associated with the transaction, by invoking the `Transaction.enlistResource` method.

If a server-side transactional application retains its database connection across multiple client requests, the application server must ensure that before dispatching a client request to the application thread, the resource is enlisted with the application's current transaction context. This implies that the application server manages the connection resource usage status across multiple method invocations.

3.7. Resource sharing

When the same transactional resource is used to interleave multiple transactions, the application server must ensure that only one transaction is enlisted with the resource at any given time. To initiate the transaction commit process, the transaction manager is allowed to use any of the resource objects connected to the same resource manager instance. The resource object used for the two-phase commit protocol does not need to have been involved with the transaction being completed.

The resource adapter must be able to handle multiple threads invoking the `XAResource` methods concurrently for transaction commit processing. This is illustrated in *Example 3.1, "Resource sharing example"*.

Example 3.1. Resource sharing example

```
XAResource xares = r1.getXAResource();

xares.start(xid1); // associate xid1 to the connection

..
xares.end(xid1); // disassociate xid1 to the connection
..
xares.start(xid2); // associate xid2 to the connection
..
// While the connection is associated with xid2,
// the TM starts the commit process for xid1
status = xares.prepare(xid1);
..
xares.commit(xid1, false);
```

A transactional resource `r1`. Global transaction `xid1` is started and ended with r1. Then a different global transaction `xid2` is associated with `r1`. Meanwhile, the transaction manager may start the two phase commit process for `xid1` using `r1` or any other transactional resource connected to the same resource manager. The resource adapter needs to allow the commit process to be executed while the resource is currently associated with a different global transaction.

3.8. Local and global transactions

The resource adapter must support the usage of both local and global transactions within the same transactional connection. Local transactions are started and coordinated by the resource manager internally. The `XAResource` interface is not used for local transactions. When using the same connection to perform both local and global transactions, the following rules apply:

- The local transaction must be committed or rolled back before a global transaction is started in the connection.
- The global transaction must be disassociated from the connection before any local transaction is started.

3.9. Transaction timeouts

You can associate timeout values with transactions in order to control their lifetimes. If the timeout value elapses before a transaction terminates, by committing or rolling back, the transaction system rolls it back. The `XAResource` interface supports a `setTransactionTimeout` operation, which allows the timeout associated with the current transaction to be propagated to the resource manager and if supported, overrides any default timeout associated with the resource manager. Overriding the timeout can be useful when long-running transactions may have lifetimes that would exceed the default, and using the default timeout would cause the resource manager to roll back before the transaction terminates, and cause the transaction to roll back as well.

If You do not explicitly set a timeout value for a transaction, or you use a value of **`0`**, an implementation-specific default value may be used. In JBoss Transactions, property value `CoordinatorEnvironmentBean.defaultTimeout` represents this implementation-specific default, in seconds. The default value is 60 seconds. A value of **`0`** disables default transaction timeouts.

Unfortunately, imposing the same timeout as the transaction on a resource manager is not always appropriate. One example is that your business rules may require you to have control over the lifetimes on resource managers without allowing that control to be passed to some external entity. JBoss Transactions supports an all-or-nothing approach to whether or not method `setTransactionTimeout` is called on `XAResource` instances.

If the `JTAEnvironmentBean.xaTransactionTimeoutEnabled` property is set to **`true`**, which is the default, it is called on all instances. Otherwise, use the `setXATransactionTimeoutEnabled` method of `com.arjuna.ats.jta.common.Configuration`.

3.10. Dynamic registration

Dynamic registration is not supported in `XAResource`. There are two reasons this makes sense.

- In the Java component-based application server environment, connections to the resource manager are acquired dynamically when the application explicitly requests a connection. These resources are enlisted with the transaction manager on an as-needed basis.
- If a resource manager needs to dynamically register its work to the global transaction, you can implement this at the resource adapter level via a private interface between the resource adapter and the underlying resource manager.

General Transaction Issues

4.1. Advanced transaction issues with TxCore

Atomic actions (transactions) can be used by both application programmers and class developers. Thus entire operations (or parts of operations) can be made atomic as required by the semantics of a particular operation. This chapter will describe some of the more subtle issues involved with using transactions in general and TxCore in particular.

4.1.1. Checking transactions

In a multi-threaded application, multiple threads may be associated with a transaction during its lifetime, sharing the context. In addition, it is possible that if one thread terminates a transaction, other threads may still be active within it. In a distributed environment, it can be difficult to guarantee that all threads have finished with a transaction when it is terminated. By default, TxCore will issue a warning if a thread terminates a transaction when other threads are still active within it. However, it will allow the transaction termination to continue.

Other solutions to this problem are possible. One example would be to block the thread which is terminating the transaction until all other threads have disassociated themselves from the transaction context. Therefore, TxCore provides the **`com.arjuna.ats.arjuna.coordinator.CheckedAction`** class, which allows the thread or transaction termination policy to be overridden. Each transaction has an instance of this class associated with it, and application programmers can provide their own implementations on a per transaction basis.

Example 4.1. Class **`CheckedAction`**

```
public class CheckedAction
{
    public synchronized void check (boolean isCommit, Uid actUid,
                                    BasicList list);
};
```

When a thread attempts to terminate the transaction and there are active threads within it, the system will invoke the `check` method on the transaction's `CheckedAction` object. The parameters to the check method are:

isCommit
: Indicates whether the transaction is in the process of committing or rolling back.

actUid
: The transaction identifier.

list
: A list of all of the threads currently marked as active within this transaction.

When `check` returns, the transaction termination will continue. Obviously the state of the transaction at this point may be different from that when `check` was called, e.g., the transaction may subsequently have been committed.

A **`CheckedAction`** instance is created for each transaction. As mentioned above, the default implementation simply issues warnings in the presence of multiple threads active on the transaction

when it is terminated. However, a different instance can be provided to each transaction in one of the following ways:

- Use the `setCheckedAction` method on the **`BasicAction`** instance.
- Define an implementation of the `CheckedActionFactory` interface, which has a single method `getCheckedAction` (final Uid `txId` , final String `actionType`) that returns a **`CheckedAction`** . The factory class name can then be provided to the Transaction Service at runtime by setting the `CoordinatorEnvironmentBean.checkedActionFactory` property.

4.1.2. Gathering statistics

By default, the Transaction Service does not maintain any history information about transactions. However, by setting the `CoordinatorEnvironmentBean.enableStatistics` property variable to **YES** , the transaction service will maintain information about the number of transactions created, and their outcomes. This information can be obtained during the execution of a transactional application via the **`com.arjuna.ats.arjuna.coordinator.TxStats`** class.

Example 4.2. Class **`TxStats`**

```
public class TxStats
{
    /**
     * @return the number of transactions (top-level and nested) created so far.
     */

    public static int numberOfTransactions();

    /**
     * @return the number of nested (sub) transactions created so far.
     *

     public static int numberOfNestedTransactions();

     /**
     * @return the number of transactions which have terminated with heuristic
     *         outcomes.
     */

    public static int numberOfHeuristics();
    /**
     * @return the number of committed transactions.
     */

    public static int numberOfCommittedTransactions();

    /**
     * @return the total number of transactions which have rolled back.
     */

    public static int numberOfAbortedTransactions();

    /**
     * @return total number of inflight (active) transactions.
     */

    public static int numberOfInflightTransactions ();

    /**
     * @return total number of transactions rolled back due to timeout.
     */

    public static int numberOfTimedOutTransactions ();
```

```
    /**
     * @return the number of transactions rolled back by the application.
     */

    public static int numberOfApplicationRollbacks ();

    /**
     * @return number of transactions rolled back by participants.
     */

    public static int numberOfResourceRollbacks ();

    /**
     * Print the current information.
     */

    public static void printStatus(java.io.PrintWriter pw);
}
```

The class **`ActionManager`** gives further information about specific active transactions through the classes **`getTimeAdded`** , which returns the time (in milliseconds) when the transaction was created, and **`inflightTransactions`** , which returns the list of currently active transactions.

4.1.3. Asynchronously committing a transaction

By default, the Transaction Service executes the `commit` protocol of a top-level transaction in a synchronous manner. All registered resources will be told to prepare in order by a single thread, and then they will be told to commit or rollback. This has several possible disadvantages:

- In the case of many registered resources, the `prepare` operating can logically be invoked in parallel on each resource. The disadvantage is that if an “early” resource in the list of registered resource forces a rollback during `prepare` , possibly many prepare operations will have been made needlessly.
- In the case where heuristic reporting is not required by the application, the second phase of the commit protocol can be done asynchronously, since its success or failure is not important.

Therefore, JBoss Transactions provides runtime options to enable possible threading optimizations. By setting the `CoordinatorEnvironmentBean.asyncPrepare` environment variable to **YES** , during the `prepare` phase a separate thread will be created for each registered participant within the transaction. By setting `CoordinatorEnvironmentBean.asyncCommit` to **YES** , a separate thread will be created to complete the second phase of the transaction if knowledge about heuristics outcomes is not required.

4.1.4. Transaction Logs

JBoss Transactions supports a number of different transaction log implementations. They are outlined below.

4.1.4.1. The ActionStore

This is the original version of the transaction log as provided in prior releases. It is simple but slow. Each transaction has an instance of its own log and they are all written to the same location in the file system

4.1.4.2. The HashedActionStore

This implementation is based on the ActionStore but the individual logs are striped across a number of sub-directories to improve performance. Check the Configuration Options table for how to configure the HashedActionStore.

4.1.4.3. LogStore

This implementation is based on a traditional transaction log. All transaction states within the same process (VM instance) are written to the same log (file), which is an append-only entity. When transaction data would normally be deleted, e.g., at the end of the transaction, a delete record is added to the log instead. Therefore, the log just keeps growing. Periodically a thread runs to prune the log of entries that have been deleted.

A log is initially given a maximum capacity beyond which it cannot grow. Once this is reached the system will create a new log for transactions that could not be accommodated in the original log. The new log and the old log are pruned as usual. During the normal execution of the transaction system there may be an arbitrary number of log instances. These should be garbage collected by the system (or the recovery sub-system) eventually.

Check the Configuration Options table for how to configure the LogStore.

Tools

This chapter explains how to start and use the tools framework and what tools are available.

Note

For this version of JBoss Transactions the GUI based tools are mainly documented in the file <INSTALL_ROOT>/INSTALL

5.1. ObjectStore command-line editors

There are currently two command-line editors for manipulating the ObjectStore. These tools are used to manipulate the lists of heuristic participants maintained by a transaction log. They allow a heuristic participant to be moved from that list back to the list of prepared participants so that transaction recovery may attempt to resolve them automatically.

5.1.1. LogEditor

Started by executing `com.arjuna.ats.arjuna.tools.log.LogBrowser`, this tool supports the following options that can be provided on the command-line.

Table 5.1. LogEditor Options

Option	Description
-tx *id*	Specifies the transaction log to work on.
-type *name*	The transaction type to work on.
-dump	Print out the contents of the log identified by the other options.
-forget *index*	Move the specified target from the heuristic list to the prepared list.
-help	Print out the list of commands and options.

5.1.2. LogBrowser

The LogBrowser, invoked by calling `com.arjuna.ats.arjuna.tools.log.LogBrowser`, is similar to the LogEditor, but allows multiple log instances to be manipulated. It presents a shell-like interface, with the following options:

Table 5.2. LogBrowserOptions

Option	Description
ls [*type*]	List the logs for the specified type. If no type is specified, the editor must already be attached to the transaction type.
select [*type*]	Browse a specific transaction type. If already attached to a transaction type, you are detached from that type first.
attach *log*	Attach the console to the specified transaction log. If you are attached to another log, the command will fail.
detach	Detach the console from the current log.
forget *pid*	Move the specified heuristic participant back to the `prepared` list. The console must be attached.

Option	Description
delete *pid*	Delete the specified heuristic participant. The console must be attached.
types	List the supported transaction types.
quit	Exit the console tool.
help	Print out the supported commands.

Configuration options

6.1. Loading a configuration

Each module of the system contains a ***module*propertyManager** class., which provides static getter methods for one or more ***name*EnvironmentBean** classes. An example is **com.arjuna.ats.arjuna.commmon.arjPropertyManager**. These environment beans are standard JavaBean containing properties for each configuration option in the system. Typical usage is of the form:

```
int defaultTimeout =
    arjPropertyManager.getCoordinatorEnvironmentBean().getDefaultTimeout();
```

These beans are singletons, instantiated upon first access, using the following algorithm.

Procedure 6.1. Algorithm for environment bean instantiation

1. The properties are loaded and populated from a properties file named and located as follows:
    a. If the properties file name property is set, its value is used as the file name.
    b. If not, the default file name is used.
2. The file thus named is searched for by, in order
    1. absolute path
    2. user.dir
    3. user.home
    4. java.home
    5. directories contained on the classpath
    6. a default file embedded in the product .jar file.
3. The file is treated as being of standard java.util.Properties xml format and loaded accordingly. The entry names are of the form EnvironmentBeanClass.propertyName:**<entry key="CoordinatorEnvironmentBean.commitOnePhase">YES</entry>**. Valid values for Boolean properties are case-insensitive, and may be one of:
    - NO
    - YES
    - FALSE
    - TRUE
    - OFF
    - ON

    In the case of properties that take multiple values, they are white-space-delimited.

Example 6.1. Example Environment Bean

```
<entry key="RecoveryEnvironmentBean.recoveryModuleClassNames">
  com.arjuna.ats.internal.arjuna.recovery.AtomicActionRecoveryModule
  com.arjuna.ats.internal.txoj.recovery.TORecoveryModule
</entry>
```

4. After the file is loaded, it is cached and is not re-read until the JVM is restarted. Changes to the properties file require a restart in order to take effect.

5. After the properties are loaded, the EnvironmentBean is then inspected and, for each field, if the properties contains a matching key in the search order as follows, the `setter` method for that field is invoked with the value from the properties, or the system properties if different.

   - Fully.Qualified.NameEnvironmentBean.propertyName
   - NameEnvironmentBean.propertyName (this is the preferred form used in the properties file)
   - the old com.arjuna... properties key (deprecated, for backwards compatibility only).

6. The bean is then returned to the caller, which may further override values by calling setter methods.

The implementation reads most bean properties only once, as the consuming component or class is instantiated. This usually happens the first time a transaction is run. As a result, calling `setter` methods to change the value of bean properties while the system is running typically has no effect, unless it is done prior to any use of the transaction system. Altered bean properties are not persisted back to the properties file.

You can configure the system using a bean wiring system such as JBoss Microcontainer or Spring. Take care when instantiating beans, to obtain the singleton via the static getter (factory) method on the module property manager. Using a new bean instantiated with the default constructor is ineffective, since it is not possible to pass this configured bean back to the property management system.

6.2. ArjunaCore Options

The canonical reference for configuration options is the Javadoc of the various **`EnvironmentBean`** classes, For ArjunaCore these are:

- **`com.arjuna.common.internal.util.logging.LoggingEnvironmentBean.java`**
- **`com.arjuna.common.internal.util.logging.basic.BasicLogEnvironmentBean.java`**
- **`com.arjuna.ats.txoj.common.TxojEnvironmentBean.java`**
- **`com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.java`**
- **`com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.java`**
- **`com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean.java`**
- **`com.arjuna.ats.arjuna.common.CoreEnvironmentBean.java`**

6.3. JBossJTA Configuration options

The canonical reference for configuration options is the javadoc of the various EnvironmentBean classes. For JBossJTA, these classes are the ones provided by ArjunaCore, as well as:

- **`com.arjuna.ats.jdbc.common.JDBCEnvironmentBean.java`**
- **`com.arjuna.ats.jta.common.JTAEnvironmentBean.java`**

6.4. JBossJTS Options

The canonical reference for configuration options is the javadoc of the various **`EnvironmentBean`** classes, For ArjunaJTS these are the ones provided by ArjunaCore, as well as:

- **`com.arjuna.orbportability.common.OrbPortabilityEnvironmentBean.java`**
- **`com.arjuna.ats.jts.common.JTSEnvironmentBean.java`**

Appendix A. Revision History

Revision 1 **Thu Oct 28 2010** **Misty Stanley-Jones** *misty@redhat.com*

Initial conversion of book into Docbook

Revision 2 **Thu Apr 14 2011** **Tom Jenkinson** *tom.jenkinson@redhat.com*

Taken from JBossJTA development guide and selected others