JBossJTA 4.15.0

JBossJTA Administration Guide

Administration of the JBossJTA toolkit, which implements the JTA API in the JBossJTA Transaction Service

Mark Little

Jonathan Halliday

Andrew Dinn

Kevin Connor

JBossJTA 4.15.0 JBossJTA Administration Guide
Administration of the JBossJTA toolkit, which implements the JTA API in the JBossJTA Transaction Service
Edition 1

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*

Preface

1. Prerequisites

JBossJTA works in conjunction with JBossJTA. In addition to the documentation here, consult the JBossJTA documentation, which ships as part of JBossJTA and is also available on the JBoss Transaction Service website at *http://www.jboss.org/jbosstm*.

2. 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.

2.1. 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.

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

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 **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.

2.2. 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"));
   }
}
```

2.3. 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.

3. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the JBoss Issue Tracker: *https://jira.jboss.org/* against the product **JBossJTA.**

When submitting a bug report, be sure to mention the manual's identifier:

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction

Apart from ensuring that the run-time system is executing normally, there is little continuous administration needed for the JBossJTA software. Refer to *Important Points for Administrators* for some specific concerns.

Important Points for Administrators

- The present implementation of the JBossJTA system provides no security or protection for data. The objects stored in the JBossJTA object store are (typically) owned by the user who ran the application that created them. The Object Store and Object Manager facilities make no attempt to enforce even the limited form of protection that Unix/Windows provides. There is no checking of user or group IDs on access to objects for either reading or writing.

- Persistent objects created in the Object Store never go away unless the StateManager.destroy method is invoked on the object or some application program explicitly deletes them. This means that the Object Store gradually accumulates garbage (especially during application development and testing phases). At present we have no automated garbage collection facility. Further, we have not addressed the problem of dangling references. That is, a persistent object, A, may have stored a Uid for another persistent object, B, in its passive representation on disk. There is nothing to prevent an application from deleting B even though A still contains a reference to it. When A is next activated and attempts to access B, a run-time error will occur.

- There is presently no support for version control of objects or database reconfiguration in the event of class structure changes. This is a complex research area that we have not addressed. At present, if you change the definition of a class of persistent objects, you are entirely responsible for ensuring that existing instances of the object in the Object Store are converted to the new representation. The JBossJTA software can neither detect nor correct references to old object state by new operation versions or vice versa.

- Object store management is critically important to the transaction service.

Starting and Stopping the Transaction Manager

By default the transaction manager starts up in an active state such that new transactions can be created immediately. If you wish to have more control over this it is possible to set the **`CoordinatorEnvironmentBean.startDisabled`** configuration option to **`YES`** and in which case no transactions can be created until the transaction manager is enabled via a call to method `TxControl.enable`).

It is possible to stop the creation of new transactions at any time by calling method `TxControl.disable`. Transactions that are currently executing will not be affected. By default recovery will be allowed to continue and the transaction system will still be available to manage recovery requests from other instances in a distributed environment. (See the Failure Recovery Guide for further details). However, if you wish to disable recovery as well as remove any resources it maintains, then you can pass **`true`** to method `TxControl.disable`; the default is to use **`false`**.

If you wish to shut the system down completely then it may also be necessary to terminate the background transaction reaper (see the Programmers Guide for information about what the reaper does.) In order to do this you may want to first prevent the creation of new transactions (if you are not creating transactions with timeouts then this step is not necessary) using method `TxControl.disable`. Then you should call method `TransactionReaper.terminate`. This method takes a Boolean parameter: if **`true`** then the method will wait for the normal timeout periods associated with any transactions to expire before terminating the transactions; if `false` then transactions will be forced to terminate (rollback or have their outcome set such that they can only ever rollback) immediately.

Note

if you intent to restart the recovery manager later after having terminated it then you MUST use the `TransactionReapear.terminate` method with asynchronous behavior set to **`false`**.

ObjectStore Management

Within the transaction service installation, the object store is updated regularly whenever transactions are created, or when **Transactional Objects for Java** is used. In a failure-free environment, the only object states which should reside within the object store are those representing objects created with the **Transactional Objects for Java** API.

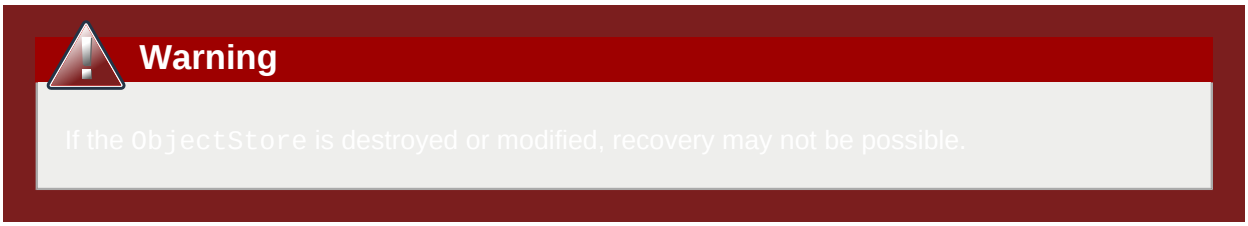
However, if failures occur, transaction logs may remain in the object store until crash recovery facilities have resolved the transactions they represent. As such it is very important that the contents of the object store are not deleted without due care and attention, as this will make it impossible to resolve in doubt transactions. In addition, if multiple users share the same object store it is important that they realize this and do not simply delete the contents of the object store assuming it is an exclusive resource.

JBossJTA Runtime Information

Compile-time configuration information is available via class **`com.arjuna.common.util.ConfigurationInfo`**. Runtime configuration is embodied in the various <name>EnvironmentBean classes, see the configuration section of the user guide. These beans have corresponding MBean interfaces and may be linked to JMX for remote inspection of the configuration if desired.

Failure Recovery Administration

The failure recovery subsystem of JBossJTA will ensure that results of a transaction are applied consistently to all resources affected by the transaction, even if any of the application processes or the machine hosting them crash or lose network connectivity. In the case of machine (system) crash or network failure, the recovery will not take place until the system or network are restored, but the original application does not need to be restarted. Recovery responsibility is delegated to *Section 5.1, "The Recovery Manager"*. Recovery after failure requires that information about the transaction and the resources involved survives the failure and is accessible afterward: this information is held in the `ActionStore`, which is part of the `ObjectStore`.



Until the recovery procedures are complete, resources affected by a transaction that was in progress at the time of the failure may be inaccessible. For database resources, this may be reported as tables or rows held by "in-doubt transactions". For **TransactionalObjects for Java** resources, an attempt to activate the `Transactional Object` (as when trying to get a lock) will fail.

5.1. The Recovery Manager

The failure recovery subsystem of JBossJTA requires that the stand-alone Recovery Manager process be running for each `ObjectStore` (typically one for each node on the network that is running JBossJTA applications). The **`RecoveryManager`** file is located in the arjunacore JAR file within the package *com.arjuna.ats.arjuna.recovery.RecoveryManager*. To start the Recovery Manager issue the following command:

```
java com.arjuna.ats.arjuna.recovery.RecoveryManager
```

If the *`-test`* flag is used with the Recovery Manager then it will display a **`Ready`** message when initialized, i.e.,

```
java com.arjuna.ats.arjuna.recovery.RecoveryManager -test
```

5.2. Configuring the Recovery Manager

The RecoveryManager reads the properties defined in the **`jbossts-properties.xml`** file.

A default version of **`jbossts-properties.xml`** is supplied with the distribution. This can be used without modification, except possibly the debug tracing fields, as shown in *Section 5.3, "Output"*.

5.3. Output

It is likely that installations will want to have some form of output from the RecoveryManager, to provide a record of what recovery activity has taken place. RecoveryManager uses the logging mechanism provided by **jboss logging**, which provides a high level interface that hides differences that exist between existing logging APIs such Jakarta log4j or JDK logging API.

The configuration of **jboss logging** depends on the underlying logging framework that is used, which is determined by the availability and ordering of alternatives on the classpath. Please consult the jboss logging documentation for details. Each log message has an associated log Level, that gives the importance and urgency of a log message. The set of possible Log Levels, in order of least severity, and highest verbosity, is:

1. **TRACE**
2. **DEBUG**
3. **INFO**
4. **WARN**
5. **ERROR**
6. **FATAL**

Messages describing the start and the periodical behavior made by the RecoveryManager are output using the **INFO** level. If other debug tracing is wanted, the finer debug or trace levels should be set appropriately.

Setting the normal recovery messages to the **INFO** level allows the RecoveryManager to produce a moderate level of reporting. If nothing is going on, it just reports the entry into each module for each periodic pass. To disable **INFO** messages produced by the Recovery Manager, the logging level could be set to the higher level of **ERROR**, which means that the RecoveryManager will only produce **ERROR**, **WARNING**, or **FATAL** messages.

5.4. Periodic Recovery

The RecoveryManager scans the ObjectStore and other locations of information, looking for transactions and resources that require, or may require recovery. The scans and recovery processing are performed by recovery modules. These recovery modules are instances of classes that implement the `com.arjuna.ats.arjuna.recovery.RecoveryModule interface`. Each module has responsibility for a particular category of transaction or resource. The set of recovery modules used is dynamically loaded, using properties found in the RecoveryManager property file.

The interface has two methods: `periodicWorkFirstPass` and `periodicWorkSecondPass`. At an interval defined by property com.arjuna.ats.arjuna.recovery.periodicRecoveryPeriod, the RecoveryManager calls the first pass method on each property, then waits for a brief period, defined by property com.arjuna.ats.arjuna.recovery.recoveryBackoffPeriod. Next, it calls the second pass of each module. Typically, in the first pass, the module scans the relevant part of the ObjectStore to find transactions or resources that are in-doubt. An in-doubt transaction may be part of the way through the commitment process, for instance. On the second pass, if any of the same items are still in-doubt, the original application process may have crashed, and the item is a candidate for recovery.

An attempt by the RecoveryManager to recover a transaction that is still progressing in the original process is likely to break the consistency. Accordingly, the recovery modules use a mechanism, implemented in the *com.arjuna.ats.arjuna.recovery.TransactionStatusManager* package, to check to see if the original process is still alive, and if the transaction is still in progress. The RecoveryManager only proceeds with recovery if the original process has gone, or, if still alive, the transaction is completed. If a server process or machine crashes, but the transaction-initiating process survives, the transaction completes, usually generating a warning. Recovery of such a transaction is the responsibility of the RecoveryManager.

It is clearly important to set the interval periods appropriately. The total iteration time will be the sum of the periodicRecoveryPeriod and recoveryBackoffPeriod properties, and the length of time it takes

to scan the stores and to attempt recovery of any in-doubt transactions found, for all the recovery modules. The recovery attempt time may include connection timeouts while trying to communicate with processes or machines that have crashed or are inaccessible. There are mechanisms in the recovery system to avoid trying to recover the same transaction indefinitely. The total iteration time affects how long a resource will remain inaccessible after a failure. – periodicRecoveryPeriod should be set accordingly. Its default is 120 seconds. The recoveryBackoffPeriod can be comparatively short, and defaults to 10 seconds. –Its purpose is mainly to reduce the number of transactions that are candidates for recovery and which thus require a call to the original process to see if they are still in progress.

Note

In previous versions of **JBossJTA**, there was no contact mechanism, and the back-off period needed to be long enough to avoid catching transactions in flight at all. From 3.0, there is no such risk.

Two recovery modules, implementations of the `com.arjuna.ats.arjuna.recovery.RecoveryModule` interface, are supplied with **JBossJTA**. These modules support various aspects of transaction recovery, including JDBC recovery. It is possible for advanced users to create their own recovery modules and register them with the Recovery Manager. The recovery modules are registered with the RecoveryManager using `RecoveryEnvironmentBean.recoveryModuleClassNames`. These will be invoked on each pass of the periodic recovery in the sort-order of the property names – it is thus possible to predict the ordering, but a failure in an application process might occur while a periodic recovery pass is in progress. The default Recovery Extension settings are:

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

5.5. Expired Entry Removal

The operation of the recovery subsystem cause some entries to be made in the ObjectStore that are not removed in normal progress. The RecoveryManager has a facility for scanning for these and removing items that are very old. Scans and removals are performed by implementations of the `com.arjuna.ats.arjuna.recovery.ExpiryScanner` interface. These implementations are loaded by giving the class names as the value of a property RecoveryEnvironmentBean.expiryScannerClassNames. The RecoveryManager calls the `scan()` method on each loaded Expiry Scanner implementation at an interval determined by the property RecoveryEnvironmentBean.expiryScanInterval. This value is given in hours, and defaults to 12hours. An expiryScanInterval value of zero suppresses any expiry scanning. If the value supplied is positive, the first scan is performed when RecoveryManager starts. If the value is negative, the first scan is delayed until after the first interval, using the absolute value.

The kinds of item that are scanned for expiry are:

TransactionStatusManager items

One TransactionStatusManager item is created by every application process that uses **JBossJTA**. It contains the information that allows the RecoveryManager to determine if the process that initiated the transaction is still alive, and its status. The expiry time for these items

is set by the property com.arjuna.ats.arjuna.recovery.transactionStatusManagerExpiryTime, expressed in hours. The default is 12, and 0 (zero) means never to expire.The expiry time should be greater than the lifetime of any single processes using **JBossJTA**.

The Expiry Scanner properties for these are:

```
 <entry key="RecoveryEnvironmentBean.expiryScannerClassNames">
    com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner
</entry>
```

Errors and Exceptions

This section covers the types and causes of errors and exceptions which may be thrown or reported during a transactional application.

Errors and Exceptions

NO_MEMORY

The application has run out of memory, and has thrown an `OutOfMemoryError` exception. **JBossJTA** has attempted to do some cleanup, by running the garbage collector, before re-throwing the exception. This is probably a transient problem and retrying the invocation should succeed.

`com.arjuna.ats.arjuna.exceptions.FatalError`

An error has occurred, and the error is of such severity that that the transaction system must shut down. Prior to this error being thrown the transaction service ensures that all running transactions have rolled back. If an application catches this error, it should tidy up and exit. If further work is attempted, application consistency may be violated.

`com.arjuna.ats.arjuna.exceptions.ObjectStoreError`

An error occurred while the transaction service attempted to use the object store. Further forward progress is not possible.

Object store warnings about access problems on states may occur during the normal execution of crash recovery. This is the result of multiple concurrent attempts to perform recovery on the same transaction. It can be safely ignored.

Selecting the JTA implementation

Two variants of the JTA implementation are accessible through the same interface. These are:

Purely local JTA	Only non-distributed JTA transactions are executed. This is the only version available with the JBossJTA distribution.
Remote, CORBA-based JTA	Executes distributed JTA transactions. This functionality is provided by the JTS distribution and requires a supported CORBA ORB. Consult the JTS Installation and Administration Guide for more information.

Both of these implementations are fully compatible with the transactional JDBC driver.

Procedure 7.1. Selecting the local JTA implementation

1. Set the property `JTAEnvironmentBean.jtaTMImplementation` to value **`com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple`** .

2. Set the property `JTAEnvironmentBean.jtaUTImplementation` to value **`com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple`** .

Note

These settings are the default values for the properties, so nothing needs to be changed to use the local implementation.

Appendix A. Revision History

Revision 0 **Wed Sep 1 2010** **Misty Stanley-Jones** *misty@redhat.com*

Conversion to Docbook

Revision 1 **Wed Apr 13 2011** **Tom Jenkinson** *tom.jenkinson@redhat.com*

Separation of installation and administration information