



JBoss Enterprise Web Platform 5 Transactions JTA Development Guide

for use with JBoss Enterprise Web Platform 5
Edition 5.1.1

Mark Little
Kevin Connor

Andrew Dinn

Jonathan Halliday

JBoss Enterprise Web Platform 5 Transactions JTA Development Guide

for use with JBoss Enterprise Web Platform 5
Edition 5.1.1

Mark Little
mlittle@redhat.com

Andrew Dinn
adinn@redhat.com

Jonathan Halliday
jhallida@redhat.com

Kevin Connor
kconnor@redhat.com

Edited by

Misty Stanley-Jones
misty@redhat.com

Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This book provides information that Java developers need in order to write transactional applications to be deployed into the JBoss Enterprise Web Platform 5 and its patch releases.

Table of Contents

Preface	4
1. Document Conventions	4
1.1. Typographic Conventions	4
1.2. Pull-quote Conventions	5
1.3. Notes and Warnings	5
2. Getting Help and Giving Feedback	5
2.1. Do You Need Help?	5
2.2. Give us Feedback	6
Chapter 1. An Introduction to the Java Transaction API (JTA)	7
Chapter 2. The JBoss JTA Implementation	8
2.1. UserTransaction	8
2.2. TransactionManager	8
2.3. Suspending and Resuming a Transaction	9
2.4. The Transaction Interface	9
2.5. Resource Enlistment	10
2.6. Transaction Synchronization	10
2.7. Transaction Equality	10
Chapter 3. The Resource Manager	11
3.1. The XAResource Interface	11
3.1.1. Extended XAResource Control	11
3.1.2. Enlisting Multiple One-Phase Aware Resources	12
3.2. Opening a Resource Manager	12
3.3. Closing a Resource Manager	12
3.4. Threads of Control	12
3.5. Transaction Association	13
3.6. Externally-Controlled Connections	13
3.7. Resource Sharing	13
3.8. Local and Global Transactions	13
3.9. Transaction Timeouts	14
3.10. Dynamic Registration	14
Chapter 4. Transaction Recovery	15
4.1. Failure recovery	15
4.2. Recovering XAConnections	15
4.3. Alternative to XAResourceRecovery	16
Chapter 5. JDBC and Transactions	18
5.1. Using the transactional JDBC driver	18
5.1.1. Managing Transactions	18
5.1.2. Restrictions	18
5.2. Transactional drivers	18
5.2.1. Loading drivers	18
5.3. Connections	18
5.3.1. Making the connection	18
5.3.2. JBossJTA JDBC Driver Properties	18
5.3.3. XADataSources	19
5.3.3.1. Java Naming and Directory Interface (JNDI)	19
5.3.3.2. Dynamic class instantiation	19
5.3.3.3. Using the connection	19
5.3.3.4. Connection Pooling	20

5.3.3.5. Reusing Connections	20
5.3.3.6. Terminating the Transaction	20
5.3.3.7. AutoCommit	20
5.3.3.8. Setting Isolation Levels	20
Chapter 6. Examples	21
6.1. JDBC example	21
6.2. BasicXARecovery Example for Failure Recovery	24
Chapter 7. Configuring JBossJTA	29
7.1. Configuring options	29
Revision History	30

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](#) 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 include 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 keys 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 key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys 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, select 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.

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 **JBoss Enterprise Application Platform 5** and the component **doc-Transactions_JTA_Programmers_Guide**. The following link will take you to a pre-filled bug report for this product: <http://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.

Chapter 1. An Introduction to the Java Transaction API (JTA)

Transactional standards provide extremely low-level interfaces for use by application programmers. Sun Microsystems has specified higher-level interfaces to assist in the development of distributed transactional applications. These interfaces are still low-level enough to require the programmer to be concerned with state management and concurrency for transactional application. They are most useful for applications which require XA resource integration capabilities, rather than the more general resources which the other APIs allow.

With reference to [JTA99], distributed transaction services typically involve a number of participants:

Application Server

Provides the infrastructure required to support an 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

(through a resource adapter ^[1]) Provides the application with access to resources. The resource manager participates in distributed transactions by implementing a transaction resource interface. The transaction manager uses this interface to communicate transaction association, transaction completion, and recovery.

Communication Resource Manager (CRM)

Supports transaction context propagation and access to the transaction service for incoming and outgoing requests.

From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed. High-level interfaces allow transaction interface users to drive transaction demarcation, resource enlistment, synchronization, and recovery processes. The JTA is a high-level application interface that allows a transactional application to demarcate transaction boundaries, and contains also contains a mapping of the X/Open XA protocol.



Note

The JTA support provided by JBossJTA is compliant with the JTA 1.0.1 specification.

[1] 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.

Chapter 2. The JBoss JTA Implementation

The Java Transaction API (JTA) 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 transactional resource manager

All of the JTA classes and interfaces are declared within the *javax.transaction* package, and the corresponding JBossJTA implementations are defined within the *com.arjuna.ats.jta* package.



Important

Each Xid that JBoss Transaction Service creates needs a unique node identifier encoded within it. JBoss Transaction Service will only recover transactions and states that match a specified node identifier. The node identifier should be provided to JBoss transaction Service via the *com.arjuna.ats.arjuna.xa.nodeIdentifier* property. You must ensure this value is unique across your JBoss Transaction Service instances. If you do not provide a value, JBoss Transaction Service will generate one and report the value via the logging infrastructure. The node identifier should be alphanumeric.

2.1. UserTransaction

The **UserTransaction** interface allows applications to control transaction boundaries. It provides methods for beginning, committing, and rolling back top-level transactions. Nested transactions are not supported, and the **begin** method throws the **NotSupportedException** when the calling thread is already associated with a transaction. **UserTransaction** automatically associates newly created transactions with the invoking thread.



Note

You can obtain a **UserTransaction** from JNDI.

```
InitialContext ic = new InitialContext();
UserTransaction utx = ic.lookup("java:comp/UserTransaction")
```

In order to select the local JTA implementation:

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

2.2. TransactionManager

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



Note

You can obtain a **TransactionManager** from JNDI.

```
InitialContext ic = new InitialContext();
TransactionManager utm = ic.lookup("java:/TransactionManager")
```

The Transaction Manager maintains the transaction context association with threads as part of its internal data structure. A thread's transaction context is either **null** or it refers to a specific global transaction. Multiple threads can be associated with the same global transaction. Nested transactions are not supported.

Each transaction context is encapsulated within 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.

The **begin** method of **TransactionManager** begins a new top-level transaction, and associates the transaction context with the calling thread. If the calling thread is already associated with a transaction then the **begin** method throws the **NotSupportedException**.

The **getTransaction** method returns the Transaction object that represents the transaction context currently associated with the calling thread. This object can be used to perform various operations on the target transaction. These operations are described elsewhere.

The **commit** method completes the transaction currently associated with the calling thread. After it returns, the calling thread is not associated with any transaction. If **commit** is called when the thread is not associated with any transaction context, an exception is thrown. In some implementations, only the transaction originator can use the **commit** operation. If the calling thread is not permitted to commit the transaction, an exception is thrown. JBossJTA does not impose any restrictions on the ability of threads to terminate transactions.

The **rollback** method is used to roll back the transaction associated with the current thread. After the **rollback** method completes, the thread is not associated with any transaction.



Note

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, then a transaction can be terminated by a thread other than its creator. If this happens, the creator must be notified. JBoss Transaction Service does this notification during commit or rollback by throwing the **IllegalStateException** exception.

2.3. Suspending and Resuming a Transaction

The JTA supports the concept of a thread temporarily suspending and resuming transactions to enable it to perform non-transactional work. The **suspend** method is called to temporarily suspend the current transaction associated with the calling thread. If the thread is not associated with any transaction, a **null** object reference is returned; otherwise, a valid **Transaction** object is returned. The **Transaction** object can later be passed 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 valid, the transaction context is associated with the calling thread. Otherwise, the thread is not associated with any transaction.



Note

If the **resume** method is invoked when the calling thread is already associated with another transaction, the Transaction Manager throws the **IllegalStateException** exception.

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



Note

JBossJTA supports allowing a suspended transaction to be resumed by a different thread, even though this feature is not required by the JTA standards.

When a transaction is suspended, the application server de-register and free up the resources that related to 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 give the transaction back its resources. 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.

2.4. The Transaction Interface

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 can be used to:

- 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 is not required 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. JBossJTA does not impose restrictions on threads terminating transactions.

2.5. Resource Enlistment

Transactional resources, such as database connections, are typically managed by the application server in conjunction with some resource adapter, and optionally, with connection pooling optimization. In order 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 (participants) are enlisted with the transaction so that they can be informed when the transaction terminates.

As stated previously, the JTA is much more closely integrated with the XA concept of resources than the arbitrary objects. For each resource in use by the application, the application server invokes the **enlistResource** method with an **XAResource** object which identifies the resource in use. See for details on how the implementation of the **XAResource** can affect recovery in the event of a failure.

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 is responsible for passing the appropriate flag in its **XAResource.start** method call to the resource manager.

The **delistResource** method is used to dissociate the specified resource from the transaction context in the target object. The application server invokes the method with two parameters:

- An **XAResources** object, which represents the resource.
- 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, in which case 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.

2.6. 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** callback object to be invoked by the transaction manager either before or after completion:

- The **beforeCompletion** method is called prior to 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 with no transaction context if **Transaction.commit** is used.
- The **afterCompletion** method is called after the transaction has completed. The status of the transaction is supplied in the parameter. This method is executed without a transaction context.

2.7. 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.

```
Transaction txObj = TransactionManager.getTransaction();
Transaction someOtherTxObj = ..
..
boolean isSame = txObj.equals(someOtherTxObj);
```

Chapter 3. 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. The `javax.transaction.xa.XAResource` interface is a Java mapping of the **XA** interface, and defines the contract between a *Resource Manager* and a *Transaction Manager* in a distributed transaction processing environment. A *resource adapter* implements the **XAResource** interface to support association of a top-level transaction to a resource. A relational database is an example of such a resource.

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

The resource manager associates the transaction with all work performed on its data between the **start** and **end** invocations. At transaction commit time, these transactional resource managers are instructed by the transaction manager to prepare, commit, or rollback the transaction according to the two-phase commit protocol.

In order to be better integrated with Java, the **XAResource** differs from the standard **XA** interface in the following ways:

- The resource manager initialization is done implicitly by the resource adapter when the connection is acquired. There is no **xa_open** equivalent.
- **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 improper handling of the **XAResource** object by the transaction manager are mapped to Java exceptions by the **XAException** class.
- The DTP concept of *Thread of Control* maps to all Java threads with access to the **XAResource** and **Connection** objects. For example, two different threads are able 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, you have no control over the order in which it will be invoked during the two-phase commit protocol, with respect to other **XAResource** objects. However, JBoss Transaction Service supports controlling the order with 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 will be invoked at the beginning or end of the commit protocol.

Note

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

Last Resource Commit optimization (LRCO) allows a single resource that is only one-phase aware (does not support **prepare**) to be enlisted with a transaction which manipulates two-phase aware participants. JBossJTA provides LRCO support.

In order to use the LRCO feature, your **XAResource** implementation must extend the `com.arjuna.ats.jta.resources.LastResourceCommitOptimisation` marker interface. When enlisting the resource via `Transaction.enlistResource`, JBoss Transaction Service allows only a single **LastResourceCommitOptimisation** participant to be used within each transaction. Your resource is driven last in the commit protocol, and the **prepare** method is not invoked.

Note

By default, an attempt to enlist more than one instance of a **LastResourceCommitOptimisation** class will fail and **false** is returned from `Transaction.enlistResource`. You can override this behavior by setting the `com.arjuna.ats.jta.allowMultipleLastResources` property to **true**. Be sure to read the section on enlisting multiple one-phase aware resources for more information.

To use the LRCO in a distributed environment, you must disable interposition support. You are still able to use implicit context propagation.

3.1.2. Enlisting Multiple One-Phase Aware Resources

In order to guarantee consistency (atomicity) of outcome between multiple participants (resources) within the same transaction, the two-phase commit protocol is used with a durable transaction log. When possessing a single one-phase aware resource, you can still achieve an atomic (all or nothing) outcome across resources by utilizing LRCO, as explained earlier.

However, you may have enlisted multiple one-phase aware resources within the same transaction. For example, a legacy database running within the same transaction as a legacy JMS implementation. In these situations, you cannot achieve atomicity of transaction outcome across multiple resources, because none of them enter the **prepare** state. They commit or rollback immediately when instructed by the transaction coordinator, without knowledge of other resource states and without any way of undoing their actions if subsequent resources make a different choice. This can cause data corruption or heuristic outcomes.

In these situations, use either of the following approaches:

- » Wrap the resources in compensating transactions.
- » Migrate the legacy implementations to two-phase aware equivalents.

If neither of these options are viable, JBoss Transaction Service supports the enlistment of multiple one-phase aware resources within the same transaction, using LRCO. LRCO is covered earlier in this chapter.



Important

Even when LRCO support is enabled, JBoss Transaction Service issues warnings when it detects this support. The log message is **"You have chosen to enable multiple last resources in the transaction manager. This is transactionally unsafe and should not be relied upon."** or, when multiple one-phase resources are enlisted within the transaction, **"This is transactionally unsafe and should not be relied on."**

3.2. Opening a Resource Manager

The X/Open **XA** interface requires the transaction manager to initialize a resource manager using the **xa_open** prior to issuing any other **xa_** calls. JTA requires initialization of a resource manager to be embedded within the resource adapter representing the resource manager. The transaction manager does not need to know how to initialize a resource manager. It must only tell the resource manager when to start and end work associated with a transaction and when to complete the transaction. The resource adapter is responsible for opening (initializing) the resource manager when the connection to the resource manager is established.

3.3. Closing a Resource Manager

A resource manager is closed by the resource adapter 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 the transaction completion process.
- » A connection object that allows the application to perform operations on the underlying resource (for example, JDBC operations on an RDBMS).

Once opened, the resource manager is kept open until the resource is explicitly released (closed). When the application invokes the connection's **close** method, the resource adapter invalidates the connection object reference that was held by the application, notifying the application server about the **close**. The transaction manager needs to invoke the **XAResource.end** method to dissociate the transaction from that connection.

The **close** notification allows the application server to perform any necessary garbage collection and mark the physical XA connection as free for reuse, in the case of connection pooling.

3.4. Threads of Control

The X/Open **XA** interface specifies that the XA calls related to transaction associations must be invoked from the same thread context. This thread-of-control requirement is not applicable to the object-oriented component-based application run-time environment, in which application threads are dispatched dynamically at method invocation time. Different threads may use the same connection resource to access the resource manager if the connection spans multiple method invocations. 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 may operate independent of thread context. Therefore, different threads may invoke the **start** and **end** methods.

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

3.5. Transaction Association

Transactions are associated with a transactional resource via the **start** method, and dissociated 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 transactions. Because JTA does not support nested transactions, the **start** method cannot be invoked on a connection that is currently associated with a different transaction.

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

3.6. Externally-Controlled Connections

If a transactional application's 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, it is incorrect for the application to perform transactional work through the connection without having the connection's resource object already associated with the global transaction. The application server must associate the **XAResource** object in use 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 enlist the resource with the application's current transaction context. In this way, 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 is responsible for ensuring that only one transaction is enlisted with the resource at any given time. To initiate the transaction **commit** process, the transaction manager can 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 be associated with the transaction being completed.

The resource adapter must be able to handle multiple threads invoking the **XAResource** methods concurrently for transaction **commit** processing. The code below declares a transactional resource **r1**. Global transaction **xid1** is started and ended with **r1**. Then a different global transaction **xid2** is associated with **r1**. In the meantime, 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.

```
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);
```

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 starting a global transaction in the connection.
- ▶ The global transaction must be dissociated from the connection before any local transaction is started.

3.9. Transaction Timeouts

Timeout values can be associated with transactions for life cycle control. If a transaction has not terminated (committed or rolled back) before the timeout value elapses, the transaction system automatically rolls it back. The **XAResource** interface supports a operation allowing 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. This is useful when long-running transactions have lifetimes that exceed the default. If the timeout is not altered, the resource manager will rollback before the transaction terminates and subsequently cause the transaction to roll back as well.

If no timeout value is explicitly set for a transaction, or a value of 0 is specified, then an implementation-specific default value may be used. In the case of JBoss Transaction Service, how this default value is set depends upon which JTA implementation you are using.

Local JTA

Set the `com.arjuna.ats.arjuna.coordinator.defaultTimeout` property to a value expressed in seconds. The default value is 60 seconds.

JTS

Set the `com.arjuna.ats.jts.defaultTimeout` property to a value expressed in seconds. The default value is 0, meaning that transactions do not time out.

Unfortunately there are situations where imposing the same timeout as the transaction on a resource manager may not be appropriate. For example, the system administrator may need control over the lifetimes of resource managers without allowing that control to be passed to some external entity. JBoss Transaction Service supports an all-or-nothing approach to whether **setTransactionTimeout** is called on **XAResource** instances.

If the `com.arjuna.ats.jta.xaTransactionTimeoutEnabled` property is set to **true** (the default), it is called on all instances. Alternatively, the **setXATransactionTimeoutEnabled** method of `com.arjuna.ats.jta.common.Configuration` can be used.

3.10. Dynamic Registration

Dynamic registration is not supported in **XAResource** for the following reasons:

- ▶ 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, it can be done at the resource adapter level via a private interface between the resource adapter and the underlying resource manager.

Chapter 4. Transaction Recovery

4.1. Failure recovery

During recovery, the Transaction Manager needs the ability to communicate to all resource managers that are in use by the applications in the system. For each resource manager, the Transaction Manager uses the **XAResource.recover** method to retrieve the list of transactions currently in a **prepared** or **heuristically completed** state. Typically, the system administrator configures all transactional resource factories that are used by the applications deployed on the system. The JDBC **XADataSource** object, for example, is a factory for the JDBC **XAConnection** objects.

Because **XAResource** objects are not persistent across system failures, the Transaction Manager needs the ability to acquire the **XAResource** objects that represent the resource managers which might have participated in the transactions prior to a system failure. For example, a Transaction Manager might use the JNDI look-up mechanism to acquire a connection from each of the transactional resource factories, and then obtain the corresponding **XAResource** object for each connection. The Transaction Manager then invokes the **XAResource.recover** method to ask each resource manager to return the transactions that are currently in a **prepared** or **heuristically completed** state.



Note

When running XA recovery, you must tell JBoss Transaction Service which types of Xid it can recover. Each Xid that JBoss Transaction Service creates has a unique node identifier encoded within it, and JBoss Transaction Service only recovers transactions and states that match the requested node identifier. The node identifier to use should be provided to JBoss Transaction Service in a property that starts with the name `com.arjuna.ats.jta.xaRecoveryNode`. Multiple values are allowed. A value of `*` forces recovery, and possibly rollback, of all transactions, regardless of their node identifier. Use it with caution.

If the JBossJTA JDBC 2.0 driver is in use, JBossJTA manages all **XAResource** crash recovery automatically. Otherwise one, of the following recovery mechanisms is used:

- If the **XAResource** is able to be serialized, then the serialized form will be saved during transaction commitment, and used during recovery. The recreated **XAResource** is assumed to be valid and able to drive recovery on the associated database.
- The `com.arjuna.ats.jta.recovery.XAResourceRecovery`, `com.arjuna.ats.jta.recovery.XARecoveryResourceManager` and `com.arjuna.ats.jta.recovery.XARecoveryResource` interfaces are used. Refer to the JDBC chapters on failure recovery for more information.

4.2. Recovering XAConnections

When recovering from failures, JBossJTA requires the ability to reconnect to databases that were in use prior to the failures, in order to resolve outstanding transactions. Most connection information is saved by the transaction service during its normal execution, and can be used during recovery to recreate the connection. However, it is possible that some of the information is lost during the failure, if the failure occurs while it is being written. In order to recreate those connections, you must provide one implementation of the JBossJTA interface

`com.arjuna.ats.jta.recovery.XAResourceRecovery` for each database that may be used by an application.



Note

If you are using the transactional JDBC 2.0 driver provided with JBossJTA, no additional work is necessary in order to ensure that recovery occurs.

To inform the recovery system about each of the **XAResourceRecovery** instances, specify their class names through properties. Any property found in the **properties** file, or registered at run-time, starting with the name `com.arjuna.ats.jta.recovery.XAResourceRecovery` is recognized as representing one of these instances. Its value is the class name, such as:

```
com.arjuna.ats.jta.recovery.XAResourceRecoveryOracle=com.foo.barRecovery
```

Additional information to be passed to the instance at creation can be specified after a semicolon:

```
com.arjuna.ats.jta.recovery.XAResourceRecoveryOracle=com.foo.barRecovery;myData=hello
```

**Note**

These properties should be in the JTA section of the **property** file.

Any errors will be reported during recovery.

```
public interface XAResourceRecovery
{
    public XAResource getXAResource () throws SQLException;

    public boolean initialise (String p);

    public boolean hasMoreResources ();

};
```

Each method should return the following information:

initialize

After the instance is created, any additional information found after the first semicolon in the property value definition is passed to the object. The object can use this information in an implementation-specific manner.

hasMoreResources

Each **XAResourceRecovery** implementation can provide multiple **XAResource** instances. Before calling to **getXAResource**, **hasMoreResources** is called to determine whether any further connections need to be obtained. If the return value is **false**, **getXAResource** is not called again during this recovery sweep and the instance is ignored until the next recovery scan.

getXAResource

Returns an instance of the **XAResource** object. How this is created (and how the parameters to its constructors are obtained) is up to the **XAResourceRecovery** implementation. The parameters to the constructors of this class should be similar to those used when creating the initial driver or data source, and should be sufficient to create new **XAResources** instances that can be used to drive recovery.

**Note**

If you want your **XAResourceRecovery** instance to be called during each sweep of the recovery manager, ensure that once **hasMoreResources** returns **false** to indicate the end of work for the current scan, it then returns **true** for the next recovery scan.

4.3. Alternative to XAResourceRecovery

The iterator-based approach that **XAResourceRecovery** uses needs to be implemented with the ability to manage states. This leads to unnecessary complexity. In JBoss Transaction Service, you can provide an implementation of the public interface, as below:

```
com.arjuna.ats.jta.recovery.XAResourceRecoveryHelper
{
    public boolean initialise(String p) throws Exception;
    public XAResource[] getXAResources() throws Exception;
}
```

During each recovery sweep, the **getXAResources** method is called, and attempts recovery on each element of the array. For the majority of resource managers, you only need one **XAResource** in the array, since the **recover** method can return multiple Xids.

Unlike instances of **XAResourceRecovery** instances, which are configured via the XML properties file and instantiated by JBoss Transaction Service, instances of **XAResourceRecoveryHelper** are constructed by the application code and registered with JBoss Transaction Service by calling **XAResourceRecoveryModule.addXAResourceRecoveryHelper**.

The **initialize** method is not currently called by JBoss Transaction Service, but is provided to allow for the addition of further configuration options in later releases.

You can deregister **XAResourceRecoveryHelper** instances, after which they will no longer be called by the recovery manager. Deregistration may block for a while, if a recovery scan is in progress.

The ability to dynamically add and remove instances of **XAResourceRecoveryHelper** while the system is running is beneficial for environments where datasources may be deployed or undeployed, such as application servers. Be careful when classloading behavior in these cases.

Chapter 5. JDBC and Transactions

5.1. Using the transactional JDBC driver

JBossJTA supports the construction of 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.

The JDBC 2.0 support has been certified with current versions of most enterprise database vendors. See <http://www.jboss.com/products/platforms/application/supportedconfigurations/> for supported configurations.

5.1.1. Managing Transactions

JBossJTA needs to associate work performed on a JDBC connection with a specific transaction. Therefore, applications must use implicit transaction propagation and indirect transaction management. For each JDBC connection, JBossJTA must be able to determine the invoking thread's current transaction context.

5.1.2. Restrictions

Nested transactions are not supported by JDBC 2.0. If you try to use a JDBC connection within a subtransaction, JBossJTA throws an exception and no work is performed using that connection.

5.2. Transactional drivers

The JBossJTA provides JDBC drivers to incorporate JDBC connections within transactions. These drivers intercept all invocations and connect them to the appropriate transactions. A given JDBC driver can only be driven by a single type of transactional driver. If the database is not transactional, ACID (atomicity, consistency, isolation, durability) properties cannot be guaranteed. Invoke the driver using the **com.arjuna.ats.jdbc.TransactionalDriver** interface, which implements the **java.sql.Driver** interface.

5.2.1. Loading drivers

You can instantiate and use the driver from within an application. For example:

```
TransactionalDriver arjunaJDBC2Driver = new TransactionalDriver();
```

The JDBC driver manager (**java.sql.DriverManager**) to manage driver instances by adding them to the Java system properties. The `jdbc.drivers` property contains a list of driver class names, separated by colons, which the JDBC driver manager loads when it is initialized.

Alternatively, you can use the **Class.forName()** method to load the driver or drivers.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Calling the **Class.forName()** method automatically registers the driver with the JDBC driver manager. You can also explicitly create an instance of the JDBC driver.

```
sun.jdbc.odbc.JdbcOdbcDriver drv = new sun.jdbc.odbc.JdbcOdbcDriver();
DriverManager.registerDriver(drv);
```

After you load the driver, it is available for making a connection with a DBMS.

5.3. Connections

The JBossJTA provides management for transactional JDBC connections. There are some implications to using them within an application, which the developer should be aware of.

5.3.1. Making the connection

Because JBossJTA provides a new JDBC driver, application code is only lightly impacted by adding transaction support. The only thing you need to do in your code is start and end transactions.

5.3.2. JBossJTA JDBC Driver Properties

The following properties can be set and passed to the JBossJTA driver. Set them in the **com.arjuna.ats.jdbc.TransactionalDriver** class.

userName

the user name to use when attempting to connect to the database.

password

the password to use when attempting to connect to the database.

createDb

set this to **true** to cause the driver to try to create the database when it connects. This may not be supported by all JDBC 2.0 implementations.

dynamicClass

dynamicClass: this specifies a class to instantiate to connect to the database, rather than using JNDI.

5.3.3. XADataSources

JDBC 2.0 connections are created from appropriate DataSources. Those connections, which participate within distributed transactions, are obtained from **XADataSources**. JBossJTA uses the appropriate DataSource when a connection to the database is made. It then obtains **XAResources** and registers them with the transaction using the JTA interfaces. The transaction service uses these **XAResources** when the transaction terminates, triggering the database to either commit or rollback the changes made via the JDBC connection.

The JBossJTA JDBC 2.0 driver can obtain **XADataSources** in one of two ways. For simplicity, it is assumed that the JDBC 2.0 driver is instantiated directly by the application.

5.3.3.1. Java Naming and Directory Interface (JNDI)

JNDI is used so that JDBC drivers can use arbitrary DataSources without knowing implementations-specific details. You can create a specific (XA)DataSource and register it with an appropriate JNDI implementation, which allows either the application or the JDBC driver to bind to and use it. Since JNDI only allows the application to see the (XA)DataSource as an instance of the interface, rather than as an instance of the implementation class, the application is not limited to only using a specific (XA)DataSource implementation.

To make the **TransactionalDriver** class use a JNDI registered **XADataSource** you need to create the **XADataSource** instance and store it in an appropriate JNDI implementation.

```
XADataSource ds = MyXADataSource();
Hashtable env = new Hashtable();
String initialCtx = PropertyManager.getProperty("Context.INITIAL_CONTEXT_FACTORY");

env.put(Context.INITIAL_CONTEXT_FACTORY, initialCtx);

initialContext ctx = new InitialContext(env);

ctx.bind("jdbc/foo", ds);
```

The Context.INITIAL_CONTEXT_FACTORY property is how JNDI specifies the type of JNDI implementation to use.

The next step is for the application must pass an appropriate connection URL to the JDBC 2.0 driver.

```
Properties dbProps = new Properties();

dbProps.setProperty(TransactionalDriver.userName, "user");
dbProps.setProperty(TransactionalDriver.password, "password");

TransactionalDriver arjunaJDBC2Driver = new TransactionalDriver();
Connection connection = arjunaJDBC2Driver.connect("jdbc:arjuna:jdbc/foo", dbProps);
```

The JNDI URL must begin with **jdbc:arjuna:** in order for the **ArjunaJDBC2Driver** interface to recognize that the DataSource needs to participate within transactions and be driven accordingly.

5.3.3.2. Dynamic class instantiation

Dynamic class instantiation is not supported or recommended. Instead, use JNDI. Refer to [Section 5.3.3.1, "Java Naming and Directory Interface \(JNDI\)"](#) for details.

5.3.3.3. Using the connection

Once the connection has been established using the appropriate method, JBossJTA monitors all operations on the connection. If a transaction is not present when the connection is used, then operations are performed directly on the database.

You can use transaction timeouts to automatically terminate transactions if the connection has outlived its usefulness.

You can use JBossJTA connections within multiple transactions simultaneously. JBossJTA does connection pooling for each transaction within the JDBC connection. So, although multiple threads may use the same instance of the JDBC connection, internally each transaction may be using a different connection instances. With the exception of the **close** method, all operations performed on the

connection at the application level use this transaction-specific connection exclusively.

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

After the driver and connection are created, they can be used in the same way as any other JDBC driver or connection.

```
Statement stmt = conn.createStatement();

try
{
    stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b INTEGER)");
}
catch (SQLException e)
{
    // table already exists
}

stmt.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");

ResultSet res1 = stmt.executeQuery("SELECT * FROM test_table");
```

5.3.3.4. Connection Pooling

For each user name and password, JBossJTA maintains a single instance of each connection for as long as that connection is in use. Subsequent requests for the same connection get a reference to the original connection, instead of a new instance. You can explicitly close the connection, but your request will be ignored until all users, including transactions, have either finished with the connection, or issued **close** method requests.

5.3.3.5. Reusing Connections

A very few JDBC drivers allow you to reuse a connection for multiple transactions. Most drivers require a new connection for each new transaction. By default, the JBossJTA transactional driver always obtains a new connection for each new transaction. However, if an existing connection is available and is currently unused, you can use the set the reuseconnection property to **true** on the JDBC URL.

```
jdbc:arjuna:sequelink://host:port;databaseName=foo;reuseconnection=true
```

5.3.3.6. Terminating the Transaction

Whenever a transaction terminates and has a JDBC connection registered with it, the JBossJTA JDBC driver instructs the database to either commit or roll back pending changes. This happens in the background, out of the purview of the application.

5.3.3.7. AutoCommit

If the AutoCommit property of the `java.sql.Connection` property is set to **true** for JDBC 1.0, the execution of every SQL statement is a separate top-level transaction, and it is not possible to group multiple statements to be managed within a single OTS transaction. Therefore, JBossJTA disables AutoCommit on JDBC 1.0 connections before using them. If AutoCommit is subsequently set to **true** by the application, JBossJTA raises the `java.sql.SQLException` exception.

5.3.3.8. Setting Isolation Levels

When you use the JBossJTA JDBC driver, you may need to set the underlying transaction isolation level on the XA connection. Its default value is `TRANSACTION_SERIALIZABLE`, but you can set this to something more appropriate for your application by setting the `com.arjuna.ats.jdbc.isolationLevel` property to the appropriate isolation level in string form. Possible values include `TRANSACTION_READ_COMMITTED` and `TRANSACTION_REPEATABLE_READ`.



Note

At present this property applies to all XA connections created in the JVM.

Chapter 6. Examples

6.1. JDBC example

The example in [Example 6.1 “JDBCTest Example”](#) illustrates many of the points described in the JDBC chapter. Refer back to it for more information.

Example 6.1. JDBC Test Example

```

public class JDBCtest
{
    public static void main (String[] args)
    {
        /*
        */

        Connection conn = null;
        Connection conn2 = null;
        Statement stmt = null;           // non-tx statement
        Statement stmtx = null; // will be a tx-statement
        Properties dbProperties = new Properties();

        try
        {
            System.out.println("\nCreating connection to database: "+url);

            /*
            * Create conn and conn2 so that they are bound to the JBossTS
            * transactional JDBC driver. The details of how to do this will
            * depend on your environment, the database you wish to use and
            * whether or not you want to use the Direct or JNDI approach. See
            * the appropriate chapter in the JTA Programmers Guide.
            */

            stmt = conn.createStatement(); // non-tx statement

            try
            {
                stmt.executeUpdate("DROP TABLE test_table");
                stmt.executeUpdate("DROP TABLE test_table2");
            }
            catch (Exception e)
            {
                // assume not in database.
            }

            try
            {
                stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b INTEGER)");
                stmt.executeUpdate("CREATE TABLE test_table2 (a INTEGER,b INTEGER)");
            }
            catch (Exception e)
            {
            }

            try
            {
                System.out.println("Starting top-level transaction.");

                com.arjuna.ats.jta.UserTransaction.userTransaction().begin();

                stmtx = conn.createStatement(); // will be a tx-statement

                System.out.println("\nAdding entries to table 1.");

                stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");

                ResultSet res1 = null;

                System.out.println("\nInspecting table 1.");

                res1 = stmtx.executeQuery("SELECT * FROM test_table");
                while (res1.next())
                {
                    System.out.println("Column 1: "+res1.getInt(1));
                    System.out.println("Column 2: "+res1.getInt(2));
                }

                System.out.println("\nAdding entries to table 2.");

                stmtx.executeUpdate("INSERT INTO test_table2 (a, b) VALUES (3,4)");

                res1 = stmtx.executeQuery("SELECT * FROM test_table2");

                System.out.println("\nInspecting table 2.");

                while (res1.next())
                {
                    System.out.println("Column 1: "+res1.getInt(1));
                    System.out.println("Column 2: "+res1.getInt(2));
                }

                System.out.print("\nNow attempting to rollback changes.");
            }
        }
    }
}

```

```

    com.arjuna.ats.jta.UserTransaction.userTransaction().rollback();

    com.arjuna.ats.jta.UserTransaction.userTransaction().begin();

    stmtx = conn.createStatement();
    ResultSet res2 = null;

    System.out.println("\nNow checking state of table 1.");

    res2 = stmtx.executeQuery("SELECT * FROM test_table");
    while (res2.next())
    {
        System.out.println("Column 1: "+res2.getInt(1));
        System.out.println("Column 2: "+res2.getInt(2));
    }

    System.out.println("\nNow checking state of table 2.");

    stmtx = conn.createStatement();
    res2 = stmtx.executeQuery("SELECT * FROM test_table2");
    while (res2.next())
    {
        System.out.println("Column 1: "+res2.getInt(1));
        System.out.println("Column 2: "+res2.getInt(2));
    }

    com.arjuna.ats.jta.UserTransaction.userTransaction().commit(true);
}
catch (Exception ex)
{
    ex.printStackTrace();
    System.exit(0);
}
}
catch (Exception sysEx)
{
    sysEx.printStackTrace();
    System.exit(0);
}
}

```

6.2. BasicXARecovery Example for Failure Recovery

This class implements the **XAResourceRecovery** interface for **XAResources**. The parameter supplied in `setParameters` can contain arbitrary information necessary to initialize the class once created. Here, it contains the name of the property file containing database connection information, as well as the number of connections that this file knows about. Values are separated by semi-colons.

It is important to understand that this is only an example, and does not contain everything which the **XAResourceRecovery** is capable of. In real life, it is not recommended to store database connection information such as user names and passwords in a raw text file, as this example does.

The db parameters specified in the property file are assumed to be in the format:

- » DB_x_DatabaseURL=
- » DB_x_DatabaseUser=
- » DB_x_DatabasePassword=
- » DB_x_DatabaseDynamicClass=

Where x is the number of the connection information.



Note

Some error handling code has been removed from this text to make it more concise.

Example 6.2. XAResourceRecovery Example

```

/*
 * Some XAResourceRecovery implementations will do their startup work here,
 * and then do little or nothing in setDetails. Since this one needs to know
 * dynamic class name, the constructor does nothing.
 */

public BasicXARecovery () throws SQLException
{
    numberOfConnections = 1;
    connectionIndex = 0;
    props = null;
}

/*
 * The recovery module will have chopped off this class name already. The
 * parameter should specify a property file from which the url, user name,
 * password, etc. can be read.
 *
 * @message com.arjuna.ats.internal.jdbc.recovery.initexp An exception
 * occurred during initialisation.
 */

public boolean initialise (String parameter) throws SQLException
{
    if (parameter == null)
        return true;

    int breakPosition = parameter.indexOf(BREAKCHARACTER);
    String fileName = parameter;

    if (breakPosition != -1)
    {
        fileName = parameter.substring(0, breakPosition - 1);

        try
        {
            numberOfConnections = Integer.parseInt(parameter.substring(breakPosition +
1));
        }
        catch (NumberFormatException e)
        {
            return false;
        }
    }

    try
    {
        String uri = com.arjuna.common.util.FileLocator.locateFile(fileName);
        jdbcPropertyManager.propertyManager.load(XMLFilePlugin.class.getName(),
uri);

        props = jdbcPropertyManager.propertyManager.getProperties();
    }
    catch (Exception e)
    {
        return false;
    }

    return true;
}

/*
 * @message com.arjuna.ats.internal.jdbc.recovery.xarec {0} could not find
 * information for connection!
 */

public synchronized XAResource getXAResource () throws SQLException
{
    JDBC2RecoveryConnection conn = null;

    if (hasMoreResources())
    {
        connectionIndex++;

        conn = getStandardConnection();

        if (conn == null) conn = getJNDIConnection();
    }

    return conn.recoveryConnection().getConnection().getXAResource();
}

public synchronized boolean hasMoreResources ()
{

```

```

    if (connectionIndex == numberOfConnections)
        return false;
    else
        return true;
}

private final JDBC2RecoveryConnection getStandardConnection ()
    throws SQLException
{
    String number = new String("" + connectionIndex);
    String url = new String(dbTag + number + urlTag);
    String password = new String(dbTag + number + passwordTag);
    String user = new String(dbTag + number + userTag);
    String dynamicClass = new String(dbTag + number + dynamicClassTag);

    Properties dbProperties = new Properties();

    String theUser = props.getProperty(user);
    String thePassword = props.getProperty(password);

    if (theUser != null)
    {
        dbProperties.put(TransactionalDriver.userName, theUser);
        dbProperties.put(TransactionalDriver.password, thePassword);

        String dc = props.getProperty(dynamicClass);

        if (dc != null)
            dbProperties.put(TransactionalDriver.dynamicClass, dc);

        return new JDBC2RecoveryConnection(url, dbProperties);
    }
    else
        return null;
}

private final JDBC2RecoveryConnection getJNDIConnection ()
    throws SQLException
{
    String number = new String("" + connectionIndex);
    String url = new String(dbTag + jndiTag + number + urlTag);
    String password = new String(dbTag + jndiTag + number + passwordTag);
    String user = new String(dbTag + jndiTag + number + userTag);

    Properties dbProperties = new Properties();

    String theUser = props.getProperty(user);
    String thePassword = props.getProperty(password);

    if (theUser != null)
    {
        dbProperties.put(TransactionalDriver.userName, theUser);
        dbProperties.put(TransactionalDriver.password, thePassword);

        return new JDBC2RecoveryConnection(url, dbProperties);
    }
    else
        return null;
}

private int numberOfConnections;
private int connectionIndex;
private Properties props;
private static final String dbTag = "DB ";
private static final String urlTag = "_DatabaseURL";
private static final String passwordTag = "_DatabasePassword";
private static final String userTag = "_DatabaseUser";
private static final String dynamicClassTag = "_DatabaseDynamicClass";
private static final String jndiTag = "JNDI_";

/*
 * Example:
 *
 * DB2_DatabaseURL=jdbc\:arjuna\sequelink\://qa02:20001
 * DB2_DatabaseUser=tester2 DB2_DatabasePassword=tester
 * DB2_DatabaseDynamicClass=com.arjuna.ats.internal.jdbc.drivers.sequelink_5_1
 *
 * DB_JNDI_DatabaseURL=jdbc\:arjuna\:jndi DB_JNDI_DatabaseUser=tester1
 * DB_JNDI_DatabasePassword=tester DB_JNDI_DatabaseName=empay
 * DB_JNDI_Host=qa02 DB_JNDI_Port=20000
 */

private static final char BREAKCHARACTER = &#39;;&#39;; // delimiter for
parameters

```

The `com.arjuna.ats.internal.jdbc.recovery.JDBC2RecoveryConnection` class can create a new connection to the database using the same parameters used to create the initial connection.

Chapter 7. Configuring JBossJTA

7.1. Configuring options

[JTA Configuration Options and Default Values](#) shows the configuration features with default values and relevant section numbers for more detailed information.

JTA Configuration Options and Default Values

com.arjuna.ats.jta.supportSubtransactions

Default Values: Yes/No

com.arjuna.ats.jta.jtaTMIImplementation**com.arjuna.ats.jta.jtaUTImplementation**

Default Values:

com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple/com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple

com.arjuna.ats.jta.xaBackoffPeriod**com.arjuna.ats.jdbc.isolationLevel**

Default Values: Any supported JDBC isolation level.

com.arjuna.ats.jta.xaTransactionTimeoutEnabled

Default Values: true / false

Revision History

Revision 5.1.1-104.400	2013-10-31	Rüdiger Landmann
Rebuild with publican 4.0.0		
Revision 5.1.1-104	2012-07-18	Anthony Towns
Rebuild for Publican 3.0		
Revision 5.1.1-100	Mon Jul 18 2011	Jared Morgan
Incorporated changes for JBoss Enterprise Web Platform 5.1.1 GA. For information about documentation changes to this guide, refer to <i>Release Notes 5.1.1</i> .		