JBossJTA Development Guide

Development reference guide for the JBossJTA implementation of the JTA API

by Mark Red Hat Little, Jonathan Red Hat Halliday,
Andrew Red Hat Dinn, and Kevin Red Hat Connor

edited by Misty Red Hat Stanley-Jones

1. Prerequisites

ArjunaTA works in conjunction with ArjunaCore. In addition to the documentation here, consult the ArjunaCore documentation, which ships as part of ArjunaCore 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* [https://fedorahosted.org/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 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.

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!

You should over ride this by creating your own local Feedback.xml file.

About This Guide

The Stand-alone JTA Programmers Guide contains information on how to use JBossJTA outside of an application server.

1.1. Audience

This guide is most relevant to engineers who want to use JBossJTA in installations that are not covered elsewhere. It is assumed that the reader is already familiar with the core JBossJTA documentation set.

1.2. Prerequisites

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

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

JDBC and Transactions

2.1. Using the transactional JDBC driver

JBossJTA supports construction of both local and distributed transactional applications which access databases using the JDBC APIs. JDBC supports two-phase commit of transactions, and is similar to the XA X/Open standard. JBossTS provides JDBC support in package com.arjuna.ats.jdbc. A list of the tested drivers is available from the JBossTS website.

Only use the transactional JDBC support provided in package com.arjuna.ats.jdbc when you are using JBossTS outside of an application server, such as JBoss Application Server, or another container. Otherwise, use the JDBC support provided by your application server or container.

2.1.1. Managing transactions

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

2.1.2. Restrictions

Nested transactions are not supported by JDBC. If you try to use a JDBC connection within a subtransaction, JBossJTA throws a suitable exception and no work is allowed on that connection. However, if you need nested transactions, and are comfortable with straying from the JDBC standard, you can set property `com.arjuna.ats.jta.supportSubtransactions` property to `YES`.

2.2. Transactional drivers

The approach JBossJTA takes for incorporating JDBC connections within transactions is to provide transactional JDBC drivers as conduits for all interactions. These drivers intercept all invocations and ensure that they are registered with, and driven by, appropriate transactions. The driver `com.arjuna.ats.jdbc.TransactionalDriver` handles all JDBC drivers, implementing the `java.sql.Driver` interface. If the database is not transactional, ACID properties cannot be guaranteed.

2.2.1. Loading drivers

Example 2.1. Instantiating and using the driver within an application

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

Example 2.2. Registering the drivers with the JDBC driver manager using the Java system properties

```
Properties p = System.getProperties();

switch (dbType)
{
case MYSQL:
    p.put("jdbc.drivers", "com.mysql.jdbc.Driver");
    break;
case PGSQL:
    p.put("jdbc.drivers", "org.postgresql.Driver");
    break;
}

System.setProperties(p);
```

The jdbc.drivers property contains a colon-separated list of driver class names, which the JDBC driver manager loads when it is initialized. After the driver is loaded, you can use it to make a connection with a database.

Example 2.3. Using the `Class.forName` method

Calling `Class.forName()` automatically registers the driver with the JDBC driver manager. It is also possible to explicitly create an instance of the JDBC driver.

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

DriverManager.registerDriver(drv);
```

2.3. Connections

Because JBossJTA provides JDBC connectivity via its own JDBC driver, application code can support transactions with relatively small code changes. Typically, the application programmer only needs to start and terminate transactions.

2.3.1. JDBC

The JBossJTA driver accepts the following properties, all located in class `com.arjuna.ats.jdbc.TransactionalDriver`.

username	the database username
password	the database password

createDb	creates the database automatically if set to `true`. Not all JDBC implementations support this.
dynamicClass	specifies a class to instantiate to connect to the database, instead of using JNDI.

2.3.2. XADataSources

JDBC connections are created from appropriate DataSources. Connections which participate in distributed transactions are obtained from XADataSources. When using a JDBC driver, JBossJTA uses the appropriate DataSource whenever a connection to the database is made. It then obtains XAResources and registers them with the transaction via the JTA interfaces. The transaction service uses these XAResources when the transaction terminates in order to drive the database to either commit or roll back the changes made via the JDBC connection.

JBossJTA JDBC support can obtain XADataSources through the Java Naming and Directory Interface (JNDI) or dynamic class instantiation.

2.3.2.1. Java naming and directory interface (JNDI)

A JDBC driver can use arbitrary DataSources without having to know specific details about their implementations, by using JNDI. A specific DataSource or XADataSource can be created and registered with an appropriate JNDI implementation, and the application, or JDBC driver, can later bind to and use it. Since JNDI only allows the application to see the DataSource or XADataSource as an instance of the interface (e.g., javax.sql.XADataSource) rather than as an instance of the implementation class (e.g., com.mydb.myXADataSource), the application is not tied at build-time to only use a specific implementation.

For the TransactionalDriver class to use a JNDI-registered XADataSource, you need to create the XADataSource instance and store it in an appropriate JNDI implementation. Details of how to do this can be found in the JDBC tutorial available at the Java web site.

Example 2.4. Storing a datasource in a 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 the JNDI way of specifying the type of JNDI implementation to use.

The application must pass an appropriate connection URL to the JDBC driver:

```
          Properties dbProps = new Properties();

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

// the driver uses its own JNDI context info, remember to set it up:
jdbcPropertyManager.propertyManager.setProperty(
                                  "Context.INITIAL_CONTEXT_FACTORY", initialCtx);
jdbcPropertyManager.propertyManager.setProperty(
                                                    "Context.PROVIDER_URL", myUrl);

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

The JNDI URL must be pre-pended with `jdbc:arjuna:` in order for the TransactionalDriver to recognize that the DataSource must participate within transactions and be driven accordingly.

2.3.2.2. Dynamic class instantiation

If a JNDI implementation is not available. you can specify an implementation of the `DynamicClass` interface, which is used to get the XADataSource object. This is not recommended, but provides a fallback for environments where use of JNDI is not feasible.

Use the property `TransactionalDriver.dynamicClass` to specify the implementation to use. An example is `PropertyFileDynamicClass`, a DynamicClass implementation that reads the XADataSource implementation class name and configuration properties from a file, then instantiates and configures it.

Deprecated class

The oracle_8_1_6 dynamic class is deprecated and should not be used.

Example 2.5. Instantiating a dynamic class

The application code must specify which dynamic class the TransactionalDriver should instantiate when setting up the connection:

```
Properties dbProps = new Properties();
```

```
dbProps.setProperty(TransactionalDriver.userName, "user");
dbProps.setProperty(TransactionalDriver.password, "password");
dbProps.setProperty(TransactionalDriver.dynamicClass,
             "com.arjuna.ats.internal.jdbc.drivers.PropertyFileDynamicClass");

TransactionalDriver arjunaJDBC2Driver = new TransactionalDriver();
Connection   connection   =   arjunaJDBC2Driver.connect("jdbc:arjuna:/path/to/
property/file", dbProperties);
```

2.3.3. Using the connection

Once the connection is established, all operations on the connection are monitored by JBossJTA. you do not need to use the transactional connection within transactions. If a transaction is not present when the connection is used, then operations are performed directly on the database.

Important

JDBC does not support subtransactions.

You can use transaction timeouts to automatically terminate transactions if a connection is not terminated within an appropriate period.

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

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

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

Example 2.6. Creating and using a 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");
```

2.3.4. 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, rather than a new instance. You can try to close the connection, but the connection will only actually be closed when all users (including transactions) have either finished with the connection, or issued `close` calls.

2.3.5. Reusing connections

Some JDBC drivers allow the reuse of a connection for multiple different transactions once a given transaction completes. Unfortunately this is not a common feature, and other drivers require a new connection to be obtained 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, JBossJTA can reuse this connection. To turn on this feature, add option `reuseconnection=true` to the JDBC URL. For instance, `jdbc:arjuna:sequelink://host:port;databaseName=foo;reuseconnection=true`

2.3.6. Terminating the transaction

When a transaction with an associated JDBC connection terminates, because of the application or because a transaction timeout expires, JBossJTA uses the JDBC driver to drive the database to either commit or roll back any changes made to it. This happens transparently to the application.

2.3.7. AutoCommit

If property `AutoCommit` of the interface `java.sql.Connection` is set to `true` for JDBC, 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 connections before they can be used. If `AutoCommit` is later set to `true` by the application, JBossJTA throws the `java.sql.SQLException`.

2.3.8. Setting isolation levels

When you use the JBossJTA JDBC driver, you may need to set the underlying transaction isolation level on the XA connection. By default, this is set to `TRANSACTION_SERIALIZABLE`, but another value may be more appropriate for your application. To change it, set the property

`com.arjuna.ats.jdbc.isolationLevel` to the appropriate isolation level in string form. Example values are `TRANSACTION_READ_COMMITTED` or `TRANSACTION_REPEATABLE_READ`.

Note

Currently, this property applies to all XA connections created in the JVM.

Examples

3.1. JDBC example

Example 3.1. JDBC example

This simplified example assumes that you are using the transactional JDBC driver provided with JBossTS. For details about how to configure and use this driver see the previous Chapter.

```
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);
        }
}
```

3.2. Failure recovery example with BasicXARecovery

This class implements the `XAResourceRecovery` interface for XAResources. The parameter supplied in `setParameters` can contain arbitrary information necessary to initialize the class once created. In this example, it contains the name of the property file in which the database connection information is specified, as well as the number of connections that this file contains information on. Each item is separated by a semicolon.

This is only a small example of the sorts of things an XAResourceRecovery implementer could do. This implementation uses a property file that is assumed to contain sufficient information

to recreate connections used during the normal run of an application so that recovery can be performed on them. Typically, user-names and passwords should never be presented in raw text on a production system.

Example 3.2. Database parameter format for the properties file

```
DB_x_DatabaseURL=
DB_x_DatabaseUser=
DB_x_DatabasePassword=
DB_x_DatabaseDynamicClass=
```

x is the number of the connection information.

Some error-handling code is missing from this example, to make it more readable.

Example 3.3. Failure recovery example with BasicXARecovery

```
/*
 * 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 = ';'; // delimiter for parameters
```

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

Using JBossJTA in application servers

JBoss Application Server is discussed here. Refer to the documentation for your application server for differences.

4.1. Configuration

When JBossJTA runs embedded in JBoss Application Server, the transaction system is configured primarily through the `transaction-jboss-beans.xml` deployment descriptor, which overrides properties read from the default properties file embedded in the `.jar` file.

Table 4.1. Common configuration attributes

CoordinatorEnvironmentBean.defaultTimeout	The default transaction timeout to be used for new transactions. Specified as an integer in seconds.
CoordinatorEnvironmentBean.enableStatistics	This determines whether or not the transaction service should gather statistical information. This information can then be viewed using the TransactionStatistics MBean. Specified as a Boolean. The default is to not gather this information.

See the `transaction-jboss-beans.xml` file and the JBoss Application Server administration and configuration guide for further information.

4.2. Logging

To make JBossTS logging semantically consistent with JBoss Application Server, the `TransactionManagerService` modifies the level of some log messages, by overriding the value of the `LoggingEnvironmentBean.loggingFactory` property in the `jbossts-properties.xml` file. Therefore, the value of this property has no effect on the logging behavior when running embedded in JBoss Application Server. By forcing use of the `log4j_releveler` logger, the `TransactionManagerService` changes the level of all `INFO` level messages in the transaction code to `DEBUG`. Therefore, these messages do not appear in log files if the filter level is `INFO`. All other log messages behave as normal.

4.3. The services

The `TransactionManager` bean provides transaction management services to other components in JBoss Application Server. There are two different version of this bean and they requires different

configuration. Take care to select the `transaction-jboss-beans.xml` suitable for your needs (local JTA or JTS).

4.4. Ensuring transactional context is propagated to the server

You can coordinate transactions from a coordinator which is not located within the JBoss server , such as when using transactions created by an external OTS server. To ensure the transaction context is propagated via JRMP invocations to the server, the transaction propagation context factory needs to be explicitly set for the JRMP invoker proxy. This is done as follows:

```
JRMPInvokerProxy.setTPCFactory( new com.arjuna.ats.internal.jbossatx.jts.PropagationContextMana
```

Appendix A. Revision History

Revision History

Revision 0 Thu Oct 28 2010 MistyStanley-Jones<misty@redhat.com>

Initial conversion of book into Docbook

Revision 1 Thu Apr 14 2011 TomJenkinson<tom.jenkinson@redhat.com>

Moved some content to the main development guide