Atomikos
TransactionsEssentials™ Guide

Atomikos TransactionsEssentials Guide

Table of Contents

List of Examples

Chapter 1. Introduction

1.1. Who Should Read This Guide

You should read this guide if you fall into one of the following categories:

- You want to use Atomikos TransactionsEssentials.
- You want to add transaction support to your J2SE application.
- You want to understand a bit more about JTA.

1.2. Preface

This user guide explains how to use Sun's Java Transaction API (JTA)™ version 1.0.1 and the Atomikos TransactionsEssentials™ embedded transaction manager. It is not meant as a general discussion of JTA. However, an overview of JTA is included, and wherever appropriate there are questions at the end of each chapter that allow you to test your understanding. You are encouraged to actively try to answer these questions, since they will allow you to get more out if this manual. For more information on JTA, you are referred to the Sun site (http://java.sun.com), where detailed JTA specifications can be downloaded for free. Although examples based on JDBC™ and JMS™ are used to illustrate the concepts, we consider those two technologies to fall outside the scope of this manual. Again, the Sun website has more information for the interested reader. If you don't like to read manuals, then you can take a shortcut and go to the examples included in the installation folder. These illustrate the concepts that are explained in the text with source code of example programs.

1.3. System Requirements

This guide has been written for Atomikos TransactionsEssentials release 3.0 or higher. In order to use the Atomikos system we recommend that you install and run a Java VM of at least version 1.4. Memory requirements are likely to depend more on your code than on our software because of the compact nature of the kernel; most modern systems should have more than enough memory. The libraries that come with Transactions include most of what you need in order to compile transactional applications: the API definitions for javax.transaction.*, javax.sql.* and javax.jms.* are included so that you do not need to get those separately.

> NOT included in this release are vendor-specific JDBC or JMS implementation libraries. For instance, if you want to use our transaction service to manage transactions that access your Oracle™ database, then you need to make sure that you have the Oracle JDBC classes installed in your classpath, in addition to the Transactions distribution classes. Likewise, if you want to use IBM MQSeries™ for JMS then you need to make sure that the MQSeries libraries are in your classpath.

Chapter 2. JTA Overview

> Unless explicitly mentioned, the discussion in this chapter is limited to pure JTA: the content of this chapter should apply for ANY JTA implementation, not just Atomikos'. Atomikos-specific information is provided in the later chapters of this manual.

This chapter is a generic JTA overview: it quickly reviews the most important things about JTA that you need to know in order to use Transactions™. The organization is as follows:

- Transactions
- What is JTA?
- Two-Phase Commit
- JTA Components
- JTA Interactions
- Questions

2.1. Transactions

A transaction is a logical unit of work which effects can either be made permanent in their entirety (committed) or cancelled in their entirety (rolled back). Before a transaction is committed or rolled back, it is active. Active transactions' effects are typically invisible to other, concurrent transactions. Consequently, only committed transactions' effects are visible (can you see why?).

Example 2.1. A typical transaction use case

Imagine that you want to publish a customer-related order message through the Java Message Service (JMS) and at the same time mark the customer's order data in the database as being processed. The message should not be sent unless the database can be updated and vice versa.

The concept of transactions requires system-level software support to make these properties hold. A piece of software that takes care of this is called a transaction manager or transaction service.

2.1.1. Nested Transactions

The nested transaction model is a variant of the normal ('flat') transactional model. Nested transactions differ in that a subtransaction can be created within an existing transaction (which becomes the parent transaction). The subtransaction is again a transaction that can be committed or rolled back. The major differences with normal transactions are in visibility and termination:

- Visibility: an active (sub)transaction's effects are visible to its subtransactions (if any). This means that there is sharing of updates from parent transaction to subtransaction.
- Termination: a rolled-back subtransaction does not affect its parent transaction. On the other hand, a committed subtransaction's effects become part of the parent transaction, and become permanent only after the top-level transaction (the one without a parent) commits.

2.2. What is JTA?

JTA is short for Sun Microsystems' Java Transaction API and is Sun's (low-level) API for creating transactions in Java and making your data access operations part of those transactions.

The JTA defines how your application can request transactional functionality on the Java platform. JTA is not a product in itself, but rather a set of Java interfaces. A vendor-specific JTA implementation referred to as a transaction manager or transaction service (such as Transactions™) is needed to actually use the functionality defined in these interfaces. In other words, you can program JTA transactions in your application, but you need the implementation classes of a JTA-compliant transaction manager vendor in order to run your application.

JTA is a standard part of the Java Enterprise (J2EE) platform and every Enterprise JavaBeans (EJB) application server should also include a JTA implementation. The JTA is said to be low-level because EJB programmers typically don't access the JTA API directly or explicitly. Rather, the EJB application server makes the appropriate calls behind the scenes.

So given that an EJB server will also give you JTA functionality, why should you consider using Atomikos' TransactionsEssentials? Here are just a few reasons:

- Not all EJB servers will provide a fully functional JTA (even if they claim so). For instance, most -if not all- open source EJB servers don't even come close to what a transaction manager needs to do, and fail when they are needed most: after restart or a server crash.
- EJB servers that do provide a reasonable transaction manager are often very expensive, and an overkill for many solutions that only need a fraction of the J2EE APIs. Atomikos TransactionsEssentials provides transactions at a fraction of the cost of a full EJB server.
- Atomikos TransactionsEssentials offers more features than defined in the JTA specification.
- Atomikos TransactionsEssentials has better and more functionality than most competitors offer.
- Atomikos TransactionsEssentials was designed for very high performance (there is no extra overhead for JTA transactions; local transactions and JTA transactions can be expected to be equally fast).
- With Atomikos TransactionsEssentials you can even bring JTA functionality on the J2SE (Java 2 Standard Edition) platform.

2.3. Two-Phase Commit

In the previous section we have referred to a JTA implementation as a transaction manager or transaction service. A transaction is a logical unit of work that either happens completely (in all databases or queues that were accessed by it) or not at all. The transaction manager is the software module that is responsible for ensuring this property. It does this by executing a two-phase commit termination protocol that addresses all of the resources that a transaction has used. This two-phase commit happens behind the scenes of your application: you typically don't notice it.

Let us briefly describe two-phase commit with the previous example still in mind. Two-phase commit works in two phases: a voting phase and a decision phase.

- In the voting (or prepare) phase, the transaction manager will ask both the JMS message queue and the database whether they can agree with a successful termination or not. Each may return a negative reply, for instance if there was a time-out which caused the database's work to be rolled back. If one of them replies positively, then it should make sure it can always make the work permanent (this implies that it can no longer cancel due to an internal time-out).
- After the transaction manager has received all of the replies (also called 'votes') it will make a global decision on the outcome of the transaction. This decision will depend on the collected replies:

- If both replies were positive (meaning that both the JMS and the database can make the work permanent), then the transaction manager will instruct each to commit.
- If at least one reply is negative (or missing) then a rollback decision is sent to the remaining resource. This means that the remaining resource cancels (rolls back) the work done for the transaction.

There are two things to notice:

- Each resource must have the capability to understand two-phase commit: it needs to reply to a prepare request from the transaction manager, and be able to rollback (cancel) the work if the transaction manager decides so.
- If a resource votes positively during prepare and is then cut off from the transaction manager (for instance, if the transaction manager crashes) then it does not know what to do. It can not cancel on its own due to the two-phase commit protocol rules, so it needs to remember the transaction indefinitely. In addition, this restricts concurrent access by other transactions. In that case, the resource is said to be in-doubt.

This explains why JTA can not be used to make anything transactional: you can only have transactional properties for applications that access the proper type of resources (those resources that understand two-phase commit).

The fact that a resource can remain in-doubt and restrict concurrent access is something that has bothered many vendors. To alleviate this restriction, a practical variant of the two-phase commit protocol includes so-called *heuristic decisions*: a resource that remains in-doubt for too long may decide to unilaterally rollback (or commit) the transaction, leading to possible violation of the all-or-nothing property. We will see more on this later in this chapter.

2.3.1. JTA Components

Here we will review the main components (interfaces) of the JTA specification and briefly discuss their roles. We will not repeat the definitions for these interfaces; those can be found in the JTA specifications on Sun's site. The packages relevant to this chapter are *javax.transaction* and *javax.transaction.xa*.

- TransactionManager
- Transaction
- Xid
- XAResource
- Synchronization
- UserTransaction
- Exceptions

2.3.1.1. TransactionManager

The transaction manager is where you can create new transactions and set properties (such as the timeout value) for future transactions. It also allows your application to retrieve the current transaction, after you have created one. An interesting point is that this is thread-safe: if you have multiple threads running concurrently, then each thread can create its own transaction and will be able to retrieve only that transaction which it created. The transaction manager will behave as a 'private' manager for each thread of your application.

The following methods are provided:

- *begin*: this method creates a transaction for the application. When it returns, you will be able to retrieve the transaction object through *getTransaction* within the current thread. Atomikos' TransactionsEssentials supports

nested transactions, meaning that a transaction can be created within another one. This means that for Atomikos TransactionsEssentials, calling this method twice in the same thread (without commit/rollback in between) will create a nested transaction, whose final commit will coincide with the commit of the first transaction you created.

- *commit*: this method will try to commit the last transaction that was created for the current thread. Afterwards, the transaction can no longer be retrieved by getTransaction. For AtomikosJTA, if the last transaction was a subtransaction then this will trigger the commit of the subtransaction. According to the semantics of nested transactions, the subtransaction's updates will not be visible or permanent before the *top-level* transaction to which it belongs is committed. The commit of a subtransaction will restore the thread association for its *parent* transaction. This means that calling *getTransaction* will again return the parent transaction.
- *rollback*: this method will trigger rollback of the last transaction that was created for the current thread. For AtomikosJTA, nested semantics apply: if the current transaction is a subtransaction, then the rollback will *not* affect the parent transaction: work done within the parent is not automatically lost by rolling back the subtransaction. As with commit, this method changes the transaction-association for the thread. For a *top-level* transaction, this leaves the current thread without a transaction. For a subtransaction, this method restores the thread association for the parent transaction.
- *getTransaction*: this method returns the transaction object for the calling thread, or null if there is no active transaction. The transaction object is needed in order to add work to it: all the work that needs to be part of this transaction must be explicitly added to it (more on that below).
- *setTransactionTimeout*: this is to set the timeout of future transactions. A timeout indicates the time a transaction is allowed to be active before it is automatically rolled back by the transaction manager.
- *getStatus*: allows you to retrieve the status of the current transaction.
- *setRollbackOnly*: see Transaction.
- *suspend*: this method is useful if an active transaction exists, but you need to start a new transaction that is independent. By suspending the current transaction, you dissociate it from the current thread and are free to begin a new one, whose commit or rollback will not affect the current transaction. If you want to have another thread continue the current transaction then this method can be used (in combination with resume) to 'pass on' the transaction to another thread.
- *resume*: this method (re-)associates the calling thread with an *existing* transaction (typically one that was suspended first). If you continue a transaction in a different thread, then that thread should call this method with the transaction as an argument. If you have done some intermediate work in a *different* transaction, then this method can be called to resume the original transaction.

Note: `setTransactionTimeout` will ignore values that exceed the maximum specified by configuration parameter `com.atomikos.icatch.max_timeout` (see the configuration chapter later in this guide).

2.3.1.2. Transaction

The transaction interface allows manipulation of an active transaction. The most important role of this interface is to add work to the scope of the transaction, thereby making the outcome of the work depend on the outcome of the transaction. The functionality of the transaction interface is discussed below.

- *enlistResource*: this method adds work to the transaction. The required argument is of type XAResource, which is an interface for resources that understand two-phase commit. By enlisting an XAResource, the work that it represents will undergo the same outcome as the transaction. If different resources are enlisted, then their

outcome will be consistent with the transaction's outcome, meaning that either all will commit or all with rollback.

- *delistResource*: this method indicates that the application stops using the XAResource for this transaction. The XAResource is essentially a connection to the underlying data source, and this method notifies the transaction manager that the connection becomes available for two-phase commit processing. There are two special cases: if a flag value of TMSUSPEND is given as a parameter, then the method call merely indicates that the application is temporarily done and intends to come back to this work. This merely serves for internal optimizations inside the data source. You should call this method if the transaction is being suspended. Coming back to such a suspended work's context is done by calling enlistResource again, with the same XAResource. The second special case is when TMFAIL is supplied as argument. This can be done to indicate that a failure has happened and that the application is *uncertain* about the work that was done. In this case, commit should not be allowed, because there is uncertainty about the contents of the transaction. For instance, if a SQLException occurs during a SQL update, then the application can not know if the update was done or not. In that case, it should delist the resource with the TMFAIL flag, because committing the transaction would lead to unknown effects on the data; this could lead to corrupt data.
- *getStatus*: this method returns the status of the transaction.
- *commit*: same as TransactionManager.commit(). This method should not be called randomly: first, every XAResource that was enlisted should also be properly delisted. Otherwise, XA-level protocol errors can occur.
- *rollback*: same as TransactionManager.rollback(). As with commit, this method should not be called randomly: first, every resource that was enlisted should also be delisted. Otherwise, XA-level protocol errors can occur.
- *setRollbackOnly*: mark the transaction so that it can not commit. This method is provided to allow application code to prevent the transaction from committing, without the requirement to call rollback directly. There are good reasons for this: the rollback should happen after proper delisting of all resources and therefore is not something that happens randomly. This method, however, can be called at any time when the transaction is active.
- *registerSynchronization*: this method adds a callback for third-party notifications about two-phase commit outcome. This is useful if you are caching updates until the end of the transaction, and need a notification about when that end is going to be.

2.3.1.3. Xid

This interface is important for the communication between the transaction manager and the system behind the XAResource. The XAResource is essentially a connection to that system, and *many different* transactions can use the same connection. Therefore each time the transaction manager wants to begin or end a transaction, it needs to use an identifier that the system behind understands and that identifies the work of the transaction in question. To this end, one JTA transaction can have one or even multiple Xid instances associated to it. It is not necessary to completely understand this mechanism in order to use Atomikos TransactionsEssentials, so it will not be discussed in more detail here.

2.3.1.4. XAResource

The XAResource is the transaction manager's connection to the data source. For each application-level connection, an XAResource is needed to make the application's work through that connection part of a JTA transaction. The details of the XAResource are not important for TransactionJTA, so we will not discuss them any further.

2.3.1.5. Synchronization

This interface is a means to register an application-level callback; it allows the application to be notified upon two-phase commit events. You can use this functionality by *implementing this interface in your application*.

Note: synchronizations are not persistent; after a crash, any recovered transactions' synchronizations will be lost.

- *beforeCompletion*: this method is called before the transaction will start its commit. A typical usage of this method is to write pending updates to the database.
- *afterCompletion*: this method is called after commit or rollback completes, and indicates whether it was successful or not.

2.3.1.6. UserTransaction

This interface is a simple and restricted version of the JTA functionality. It is the typical application-level transaction service handle in EJB. You can use this interface to expose only a subset of JTA functionality to the application code.

Note: `setTransactionTimeout` will ignore values that exceed the maximum specified by configuration parameter `com.atomikos.icatch.max_timeout` (see the configuration chapter later in this guide).

2.3.1.7. Exceptions

There are some specific exceptions in JTA that are worth mentioning: those that concern the heuristic terminations. Since they are not really made clear in the JTA specification, we will mention something about them here. Whenever a heuristic error happens the transaction manager should keep a log entry for the transaction involved, so that a human administrator can resolve any conflicts. Part of Atomikos' patent applications concern precisely the kind of information that is available in the logs in these cases.

- *HeuristicCommitException*: if all resources have been in-doubt for too long, they may have committed the transaction although all replied positively during the prepare of two-phase commit. If the transaction manager later re-establishes contact and instructs the resources to rollback then this exception will be thrown to the application. It indicates an anomaly in the transaction's outcome, where *all resources involved* have chosen to commit heuristically, because all were left in-doubt. If you get this exception, it means that the entire transaction has been committed, although rollback was desired.
- *HeuristicRollbackException*: all resources have decided to rollback although the final decision of the transaction manager was to commit. This is similar to the previous case; this time it means that the entire transaction has in fact been rolled back whereas the desired outcome was commit.
- *HeuristicMixedException*: this is the most complex error, where some of the resources may have committed and others have rolled back. It hints that the transaction's effects are only partial; this is a clear violation of transactional semantics. Remember, more information should be in the logs.

2.3.1.8. JTA Interactions

This section hightlights some typical JTA interactions for JDBC data sources. For other resources such as JMS queues, most things are the same except for the way the XAResources are to be retrieved.

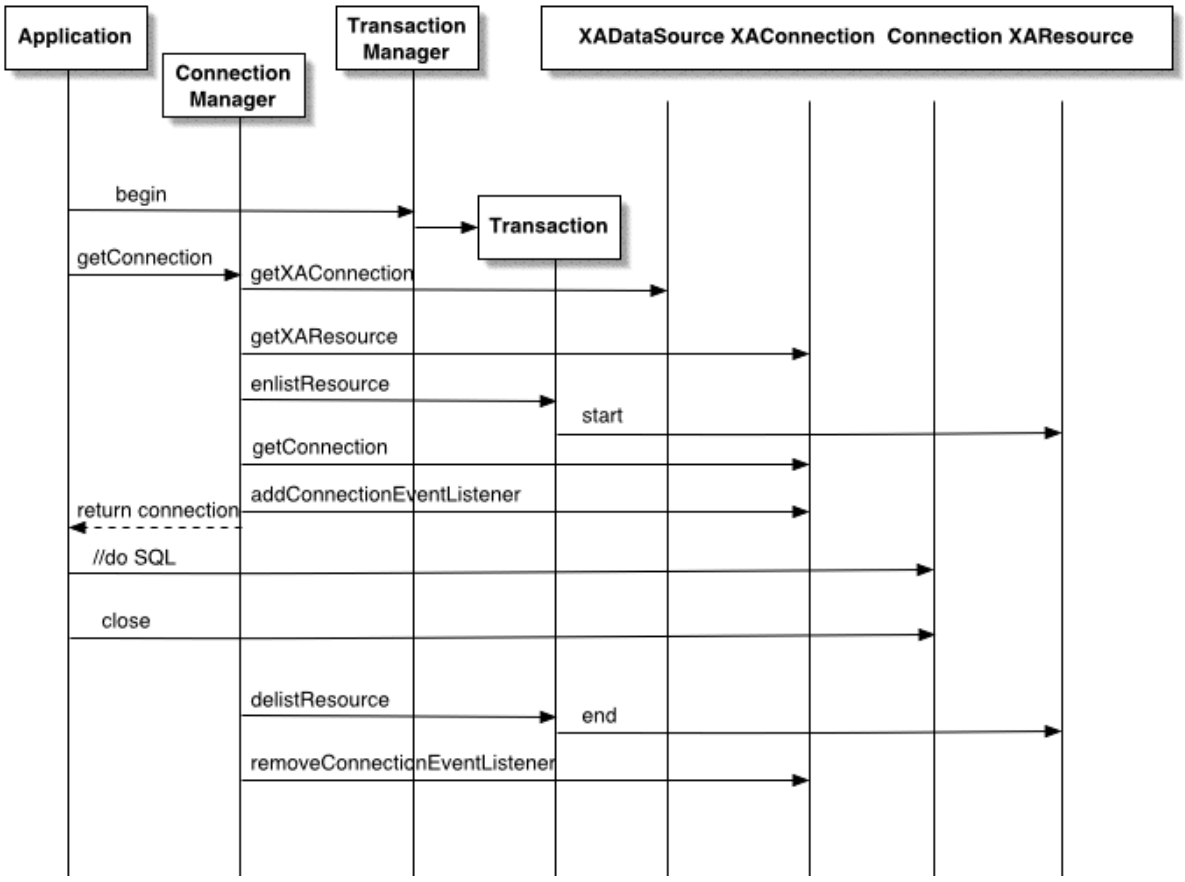
- Active Transaction
- Transaction Commit

- Transaction Rollback
- Transaction Termination with Errors

2.3.1.8.1. Active Transaction

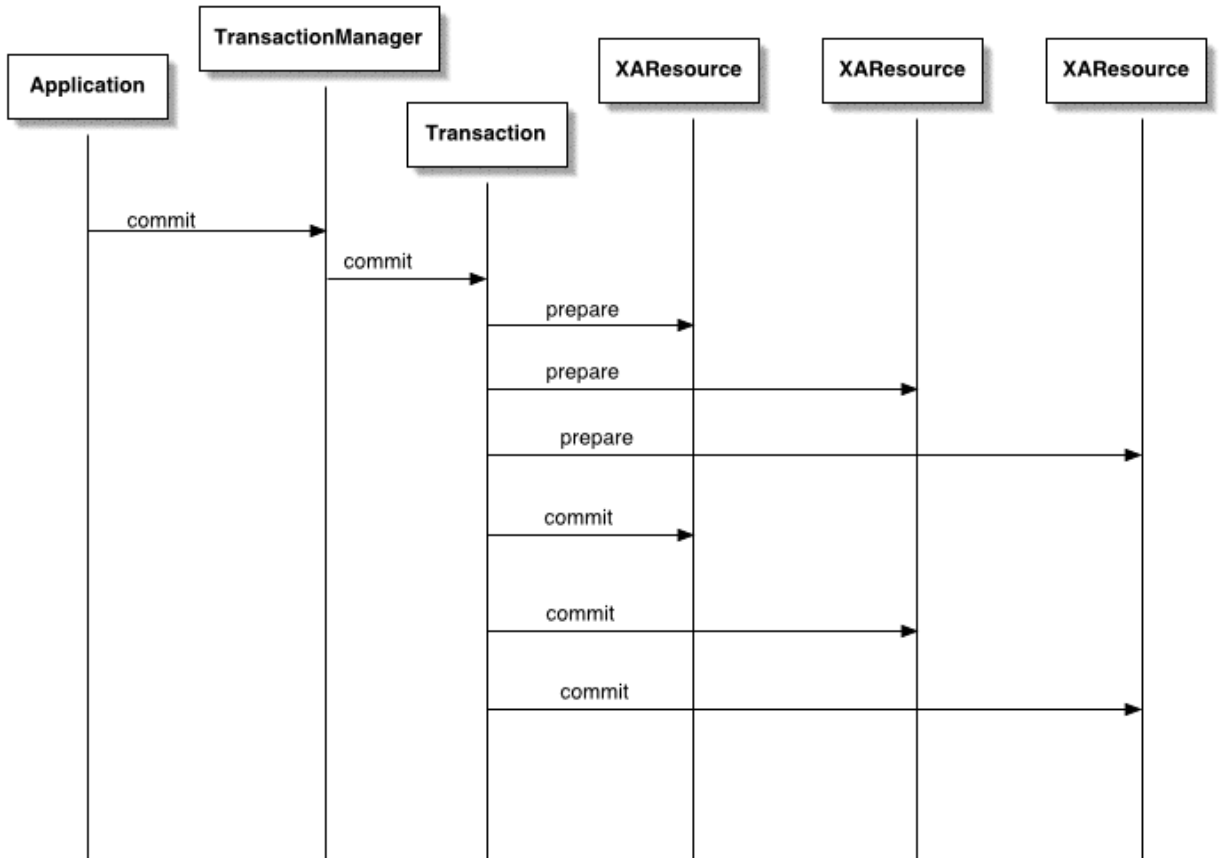
The typical interactions for an active JTA transaction are shown below. Note that the only thing you have to provide is the Application, and the Connection Manager if you don't use the Atomikos connection pools.

> Please note a very important point when using connection pools: the connection manager will only be able to delist a resource when it is informed about the application-level close operation on the JDBC connection. This means that you should *always* properly close the connections from the pool; this should be done in the finally-part of a *try{...}finally{...}* block. Opening the connection belongs in the try-part.



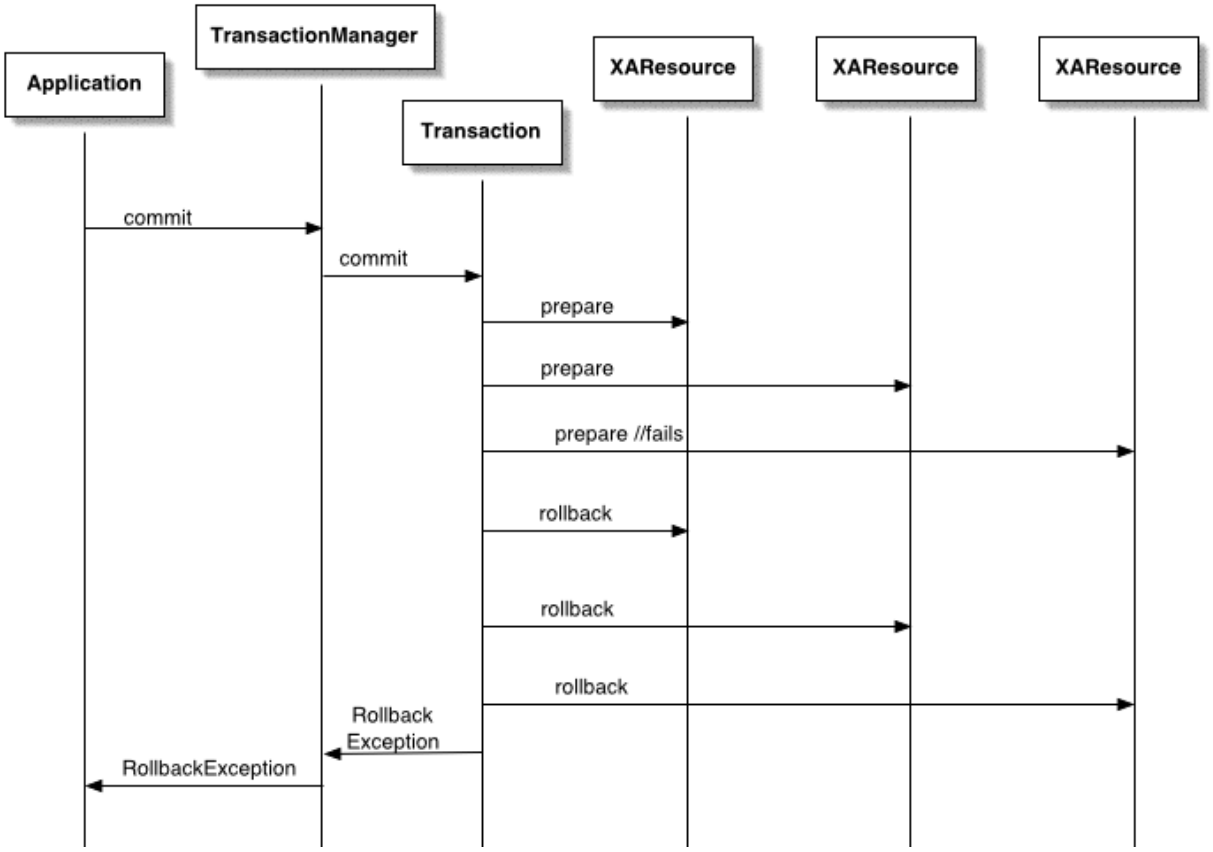
2.3.1.8.2. Transaction Commit

The typical commit scenario is shown below.



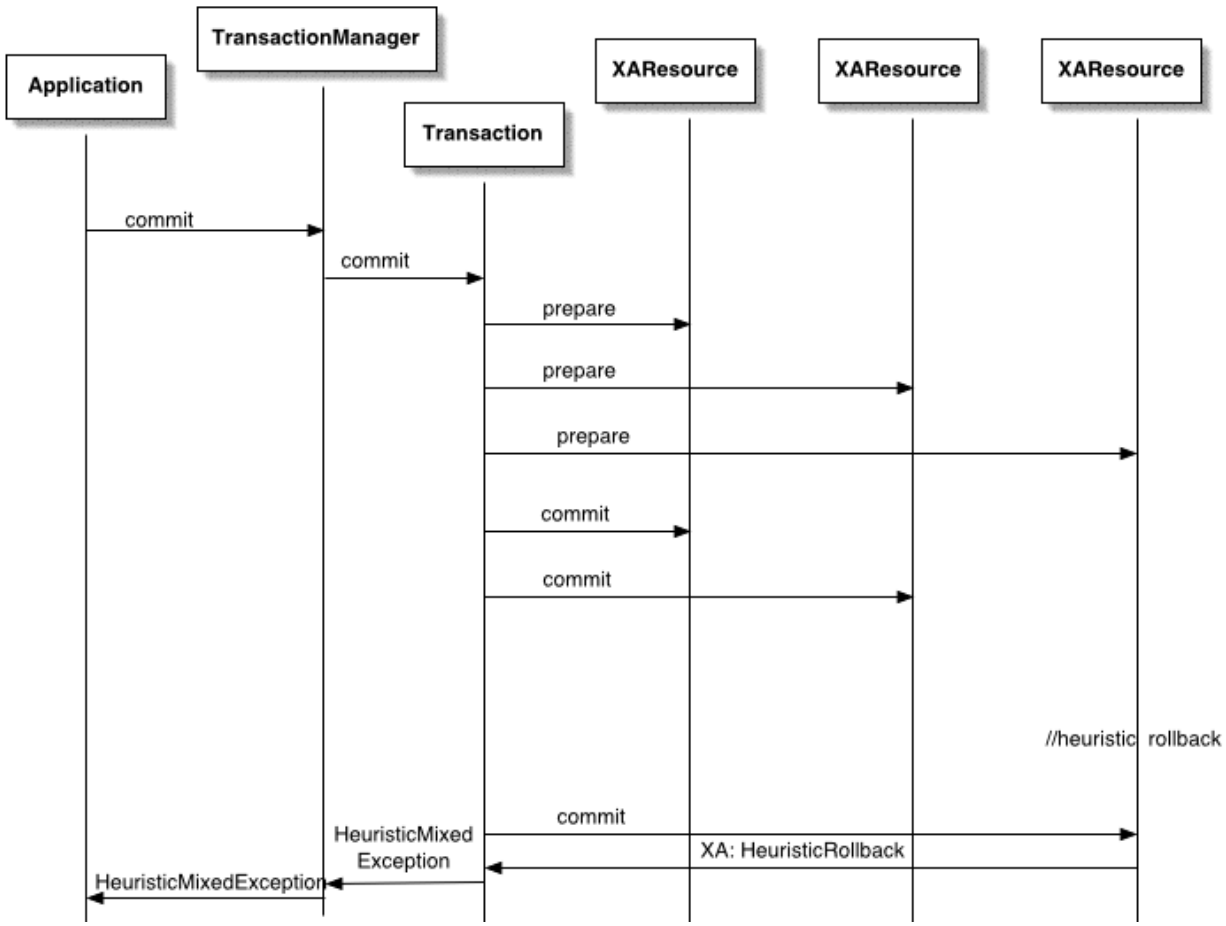
2.3.1.8.3. Transaction Rollback

A possible rollback scenario is shown below: the application requests commit, but one of the XAResources has timed out and rolled back *before* it is asked to prepare. The result is rollback, and an application-level exception (since commit was requested).



2.3.1.8.4. Transaction Termination with Errors

A possible heuristic scenario is shown below: the application requests commit, but one of the XAResources becomes unreachable *after* it is asked to prepare. The result is heuristic rollback, and by the time the transaction manager re-establishes contact commit fails with a heuristic error. An application-level heuristic mixed exception is thrown (since the other two XAResources did commit, parts have been committed and other parts have not).



2.3.1.9. Questions

2.3.1.9.1. Question 1

Consider the two-phase commit protocol and the example in the text: an update in a database and a message being published in JMS, as part of one transaction. Can't you solve the problem of reaching the same outcome for both parts by controlling the order in which you execute each one? For instance, why bother controlling the outcome of a database update if your application knows that it succeeded already?

2.3.1.9.2. Question 2

Imagine the following scenario: an application is using a JTA implementation to manage transactions that access two JDBC databases, say, DBa and DBb. The application has updated both of them and is in the course of committing the transaction. As part of that commit processing, the transaction manager has received positive replies from both databases when it asked them to prepare and hence it decides to commit. However, it can only notify DBa of this decision: before DBb can be told about commit, a system crash happens and causes DBb to go down. The rest of the system is not affected because DBb is running on its own private machine. The transaction manager repeatedly retries to connect to DBb, but after a while it gives up and throws an exception to the application. What is the exact type of this exception?

2.3.1.9.3. Question 3

Which of these methods can be called at any time when a transaction is active: *setRollbackOnly* or *rollback*? Why?

2.3.1.9.4. Question 4

An application has started a transaction and is in the middle of updating a JDBC database when a SQLException happens. Should the transaction be committed or rolled back?

2.3.1.9.5. Question 5

An application is listening on incoming remote method invocation (RMI) request. An incoming requests is executed in some Java thread according to the virtual machine's internal rules. This thread could be the same one as for a previous request. The application-level logic for an invocation involves creating a JTA transaction and doing some JDBC work as well as publishing a JMS message. If you are using a JTA implementation that supports nested transactions, why should you *always* make sure that the transaction is terminated (by commit or rollback) *before* the invocation returns?

Chapter 3. Configuring Atomikos TransactionsEssentials

Whereas the previous chapter was generic JTA information, this chapter is specific to Atomikos TransactionsEssentials. It concerns the setup (configuration) of Atomikos TransactionsEssentials in your application.

> Atomikos TransactionsEssentials is an *embedded* transaction service, meaning that it runs *inside the same VM as your application*. This optimizes speed and availability of your application.

Configuration is done in the configuration file, a properties file with *property=value* combinations of important transaction service settings. The settings you use determine general transaction-related information such as where logfiles are to be kept and what default timeout values are.

3.1. The Configuration File

The configuration file contains the parameters for initialization and operation of the transaction service. If this file can not be found then default values will be used. To instruct Atomikos TransactionsEssentials to use a custom configuration file, there are several possibilities:

- Name the file *transactions.properties* and put it in your classpath.
- Give your file any name and location you like, and specify this as a system property at startup: *java -Dcom.atomikos.icatch.file=path_to_your_file ...* Note that setting this system property overrides any transactions.properties configuration data that you might have according to the first approach.
- To avoid using a configuration file, you can also use run-time values for each of the parameter settings. You can indicate that this is the case by supplying the following system property at startup: *java -Dcom.atomikos.icatch.no_file ...* In that case, the properties need to be set programmatically before initialization of the transaction service. This is explained in the last part of this guide.

> A sample file is included in the installation folder. The configuration file can contain the parameters shown in the example below. Note that the format should be a valid Java property-file format.

Example 3.1. Sample configuration file for the transaction service

```
#SAMPLE PROPERTIES FILE FOR THE TRANSACTION SERVICE
#THIS FILE ILLUSTRATES THE DIFFERENT SETTINGS FOR THE TRANSACTION MANAGER
#UNCOMMENT THE ASSIGNMENTS TO OVERRIDE DEFAULT VALUES;

#Required: factory class name for the transaction service core.
#
com.atomikos.icatch.service=com.atomikos.icatch.standalone.UserTransactionServiceFactory
#
```

```
#Set the number of log writes between checkpoints
#
#com.atomikos.icatch.checkpoint_interval=500

#Set output directory where console file and other files are to be put
#make sure this directory exists!
#
#com.atomikos.icatch.output_dir = ./

#Set directory of log files; make sure this directory exists!
#
#com.atomikos.icatch.log_base_dir = ./

#Set base name of log file
#this name will be  used as the first part of
#the system-generated log file name
#
#com.atomikos.icatch.log_base_name = tmlog

#Set the max number of active local transactions
#or -1 for unlimited.
#
#com.atomikos.icatch.max_actives = 50

#Set the max timeout (in milliseconds) for local transactions
#
#com.atomikos.icatch.max_timeout = 300000

#The globally unique name of this transaction manager process
#override this value with a globally unique name
#
#com.atomikos.icatch.tm_unique_name = tm


#Do we want to use parallel subtransactions? JTA's default
#is NO for J2EE compatibility.
#
#com.atomikos.icatch.serial_jta_transactions=true


#If you want to do explicit resource registration then
#you need to set this value to false. See later in
#this manual for what explicit resource registration means.
#
#com.atomikos.icatch.automatic_resource_registration=true


#Set this to WARN, INFO or DEBUG to control the granularity
#of output to the console file.
#
#com.atomikos.icatch.console_log_level=WARN
```

```
#Do you want transaction logging to be enabled or not?
#If set to false, then no logging overhead will be done
#at the risk of losing data after restart or crash.
#
#com.atomikos.icatch.enable_logging=true

#Should two-phase commit be done in (multi-)threaded mode or not?
#
#com.atomikos.icatch.threaded_2pc=true

#Should exit of the VM force shutdown of the transaction core?
#
#com.atomikos.icatch.force_shutdown_on_vm_exit=false

#Should the logs be protected by a .lck file on startup?
#
#com.atomikos.icatch.lock_logs=true
```

3.2. Default Values You Should Override

Although reasonable defaults are provided, you probably should override the following configuration parameters to suit your application's needs:

- `com.atomikos.icatch.tm_unique_name`: set this value to a unique name for each application.
- `com.atomikos.icatch.max_timeout`: this value limits the timeout that can be set for any transaction. More precisely: you cannot specify a timeout that exceeds the limit specified here. Set this value according to the needs of your application. If you have long queries or updates then the default may not be sufficient.
- `com.atomikos.icatch.force_shutdown_on_vm_exit`: set this to true if you want shutdown behaviour (VM exit) to show backward compatibility with pre-3.3 releases. Note: this is NOT recommended for Spring use, since it is the Spring container that should shutdown the transaction core.

3.3. Ant-Style References in the Properties File

As of release 3.2, a value in the properties file can reference another property - see the wiki at http://wiki.atomikos.org for more information and examples.

3.4. Questions

3.4.1. Question 1

If you don't have a configuration file and don't set system properties related to the configuration, where will the transaction service write log files?

1. In the current directory.
2. In the directory where you start your application.
3. In a new directory, called *tmlog*.

3.4.2. Question 2

In the configuration file, there is a parameter called *com.atomikos.icatch.checkpoint_interval*. If you increase this parameter's value, then what happens?

1. The average log file size will be larger.

2. The average log file size will be smaller.

3.4.3. Question 3

What parameter has no default value and MUST be specified in any configuration file?

Chapter 4. Programming Transactional Applications

For any J2SE application based on Atomikos TransactionsEssentials, we can distinguish the following main programming styles. Which one is best for you depends on the way you want to use Atomikos TransactionsEssentials, and what exactly your application needs. Each option is discussed in more detail in the rest of this chapter.

1. *Regular Applications: Atomikos JDBC/JMS and the UserTransaction* Use this approach if you want to use our JDBC DataSource or JMS QueueConnectionFactory adapters to perform JDBC or JMS within the scope of a transaction.

2. *For XA-Level Integration: The JTA TransactionManager* Use this approach if you don't want to use our JDBC or JMS adapters and still want to do JTA/XA transactions with minimal effort.

3. *For Sophisticated Needs: The Atomikos UserTransactionService* This approach allows you to initiate the startup and shutdown of the transaction service, and gives you full control over how resources are configured. You will also need this approach if you want to extend transactions across RMI or JMS communication links.

4.1. Regular Applications: Atomikos JDBC/JMS and the UserTransaction

This is the easiest and most straightforward way of using Atomikos TransactionsEssentials. Your application uses the built-in Atomikos resource adapters to connect to the back-end systems, and delimits transactions through our UserTransaction implementation. No other steps are required (in particular, transaction service startup, recovery and shutdown are done automatically).

- Getting the UserTransaction
- JDBC: Using an Atomikos DataSource
- JDBC: Using an Atomikos Non-XA DataSource
- JMS: Using an Atomikos QueueConnectionFactory
- JMS: Message-Driven Functionality
- JMS: Pooled Receiver Sessions
- JMS: Pooled Sender Sessions
- JMS: Bridging Different JMS Domains

4.1.1. Getting the UserTransaction

We have a built-in implementation of *javax.transaction.UserTransaction* that you can use for your application's transaction. To do this, you merely need to construct an instance of class *com.atomikos.icatch.jta.UserTransactionImp* (use the default, no-argument constructor):

Example 4.1. Getting the UserTransaction

```
com.atomikos.icatch.jta.UserTransactionImp utx =
   new com.atomikos.icatch.jta.UserTransactionImp();

//now we are ready to do transactions!
//startup and recovery of the transaction manager
//will happen automatically upon first use of utx
```

This is all you need: startup and recovery of the transaction service will happen automatically as soon as you start using the UserTransaction. Shutdown of the transaction engine is triggered automatically as well, and happens when your application's VM exits. Also, it is worth pointing out that the UserTransactionImp class implements both *java.io.Serializable* and *javax.naming.Referenceable*, meaning it can be stored in JNDI where available. All instances of the class com.atomikos.icatch.jta.UserTransactionImp are equivalent to your application (if you have many, you can use any of them when you like).

4.1.2. JDBC: Using an Atomikos DataSource

Atomikos provides two main categories of javax.sql.DataSource implementations: one that is aware of an underlying (vendor-specific) javax.sql.XADataSource, and another one that uses any regular (non-XA) JDBC driver class. This section discusses the first category, while the next section focuses on the second.

> Our DataSource implementation is called *com.atomikos.jdbc.AtomikosDataSourceBean*. As its name suggests, this class is a JavaBean class, meaning it has a default no-argument constructor and get/set methods for setup properties. These properties indicate preferences such as connection pool settings, and also how to construct and access an underlying RDBMS vendor-specific instance of javax.sql.XADataSource. If your RDBMS vendor does not support XADataSource, then see the next section on what to do.

In addition, our DataSource class implements both *java.io.Serializable* and *javax.naming.Referenceable* so an instance can be configured and then stored in JNDI where available. In order to use this DataSource for your application's JDBC, you need to get hold of a configured instance:

Example 4.2. Configuring an Atomikos DataSource

```
com.atomikos.jdbc.AtomikosDataSourceBean ds =
      new com.atomikos.jdbc.AtomikosDataSourceBean();

//set the necessary properties
//see the javadoc documentation of the AtomikosDataSourceBean to
//get more information on which properties to set and how
//and see the sample program in examples/j2se/simple/jdbc/xadatasource
//for the complete example code
```

Using the DataSource for transactions is equally simple: just begin a new transaction, get a connection from the DataSource, and do any SQL you want. When the transaction is committed/rolledback, all SQL is committed/rolledback as well. The typical code pattern for doing this is shown below.

Example 4.3. Using an Atomikos datasource

```
boolean rollback = false;
try {
     //begin a transaction
     utx.begin();

     //access the datasource and do any JDBC you like
     Connection conn = ds.getConnection();
     ...

     //always close the connection for reuse in the
     //DataSource-internal connection pool
     conn.close();

}
catch ( Exception e ) {
     //an exception means we should not commit
     rollback = true;
}
finally {
   if ( !rollback ) utx.commit();
   else utx.rollback();
}
```

4.1.3. JDBC: Using an Atomikos Non-XA DataSource

For JDBC vendors that don't support XADataSource, we have a DataSource implementation that allows integration with Atomikos TransactionsEssentials nevertheless.

> It should be clear that this has limitations with respect to recovery: if there is no XA functionality, then pending transactions can't be recovered after restart or crash of your application. This is no problem if you only use one database, but it can be a serious data integrity risk if you use two or more databases/systems within the scope of the same transaction.

Example 4.4. Using an Atomikos Non-XA DataSource

```
com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean ds =
      new com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean();

//set the necessary properties
//see the javadoc documentation of the AtomikosNonXADataSourceBean to
//get more information on which properties to set and how
//and see the sample program in examples/j2se/simple/jdbc/drivermanager
//for the complete example code
```

The typical code pattern for doing transactions that include work in a AtomikosNonXADataSourceBean is the same as the one in the previous case.

4.1.4. JMS: Using an Atomikos JMS ConnectionFactory

For JMS queues and topics, Atomikos also has a built-in connector represented by the class *com.atomikos.jms.AtomikosConnectionFactoryBean*. Similar to our DataSource, instances of this class need a JMS vendor-specific XAConnectionFactory to work with. Please refer to the javadoc of this class for more information. A typical usage pattern is show below for queues (topics are very similar).

Example 4.5. Using an AtomikosConnectionFactoryBean

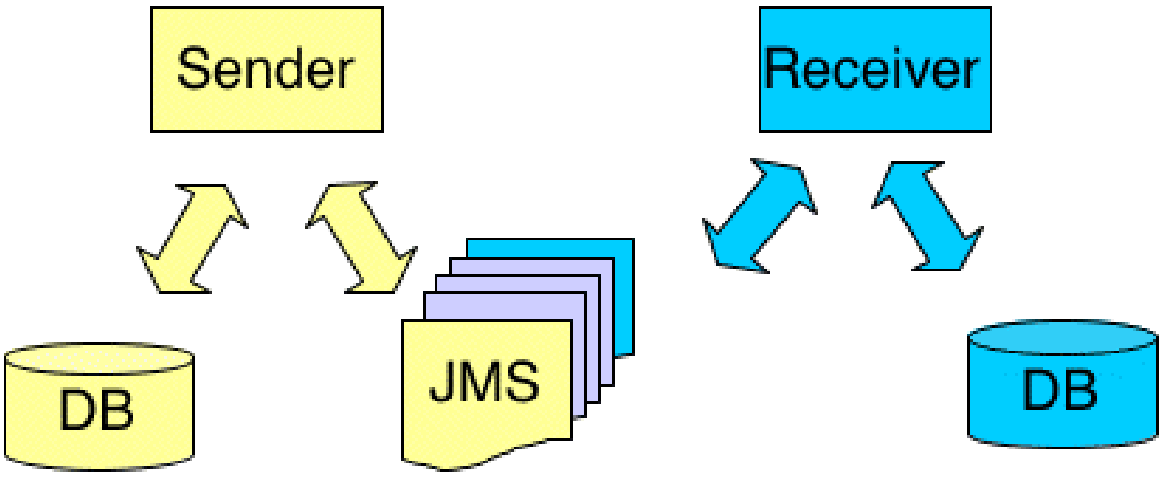
See the examples in the download folder for how to use this class.

JMS behaves differently in combination with JTA/XA transactions. In particular:

- Sending a message in a JTA transaction has no effect until commit.
- Messages that are received in a JTA transaction will only be removed from the queue at transaction commit time.

The consequences of this behaviour are also interesting:

1. The sender and receiver processes of a message always execute in a different transaction (see the figure below): the sending transaction has to commit before the message is actually transported to the receiver.
2. Because of 1, it is impossible to receive a reply for a message *sent in the same transaction*.
3. Also because of 1, there is no way to rollback the sending process when the receiver has fatal errors in processing a request. The sender's transaction has already committed before the receiver even gets the message with the request.



Always keep these restrictions in mind when using JMS in a JTA/XA transaction. These restrictions are characteristic of any standard J2EE application that combines JMS and transactions. It is possible to use Atomikos TransactionsEssentials without being bound by these restrictions, provided that you use

the propagation mechanism outlined later in this manual. In that case, you use a regular (non-XA, non-Atomikos) JMS QueueConnectionFactory and add the transaction propagation to each message that you send. This way, sending a message is not delayed until commit, the transaction context can be imported at the receiver, and both ends of the JMS communication link can execute in the same transaction.

4.1.5. JMS: Message-Driven Functionality

Atomikos TransactionsEssentials also contains a feature that is similar to message-driven beans, allowing your application to process JMS queue messages in a reliable and transactional way. In particular, your application can register implementations of *javax.jms.MessageListener* to receive messages in a transaction. There is no need for developers to know EJB in order to do this.

For more information, see the javadoc about *com.atomikos.jms.extra.MessageDrivenContainer* as well as the example programs included in the release.

4.1.6. JMS: Managed Sender Sessions

Sending messages with the benefit of pooled and managed sessions can also be done, by using the *com.atomikos.jms.extra.SingleThreadedJmsSenderTemplate*. This class allows you to reuse the same session for sending multiple messages, and refreshes the session (if necessary) to simplify application-level code. For more information, please check the javadoc as well as the example programs included in the release.

The *com.atomikos.jms.extra.SingleThreadedJmsSenderTemplate* is not thread-safe. If you have threaded code then use *com.atomikos.jms.extra.ConcurrentJmsSenderTemplate* instead.

4.2. For XA-Level Integration: The JTA TransactionManager

If you don't want to use the Atomikos connectors for JDBC or JMS, then you can still use our transaction manager, but you will have to integrate at the level of JTA/XA. This means that you will have to explicitly enlist/delist XAResource instances with the transaction service (and within each transaction). This section explains how to do this.

This approach for using Atomikos TransactionsEssentials will only work if the configuration parameter *com.atomikos.icatch.automatic_resource_registration* is set to *true*.

4.2.1. Getting the TransactionManager

Our implementation of *javax.transaction.TransactionManager* is represented by the class *com.atomikos.icatch.jta.UserTransactionManager*. Like in the case of the UserTransaction, you don't need to do anything special besides constructing an instance of this class. All the rest (transaction service startup, recovery and shutdown) is handled behind the scenes. If you have multiple instances of this TransactionManager class then you can use any you like: they are all equivalent to your application.

Like UserTransactionImp, the class UserTransactionManager implements both *java.io.Serializable* and *javax.naming.Referenceable*, meaning it can be stored in JNDI where available.

> For J2EE applications, the automatic startup mechanism is undesirable since multiple deployed applications could end up with different transaction engines. This is not recommended, and therefore J2EE application programmers should use the class *com.atomikos.icatch.jta.J2eeTransactionManager* instead: this class has the same functionality except that it doesn't trigger automatic startup of the transaction engine. In that case, you should also use the Atomikos control panel web-application (included in the installation) to automatically initiate transaction service startup and shutdown when the application server starts/stops. Like its sibling class, the J2eeTransactionManager implementation can also be bound in JNDI where available.

4.2.2. Typical Code Pattern for JTA/XA

For JTA/XA integration, the recommended code pattern is shown below. The case is illustrated for an XADataSource, but other XA-capable resources work the same way. The essence of the example is that you need to start a transaction, enlist/delist one or more XAResource instances, and then commit or rollback. This is more complicated than if you were using our JDBC or JMS adapters (as in the previous case), because our adapters do most of this for you. Please see the demo application in examples/j2se/simple/xa for complete and working source code.

Example 4.6. Typical pattern of JTA/XA usage

```
//GENERIC: get the transaction manager
com.atomikos.icatch.jta.UserTransactionManager tm =
    new com.atomikos.icatch.jta.UserTransactionManager();

//the transaction service will startup and recover whenever the
//tm is used for the first time

//SPECIFIC FOR JDBC: get the XADataSource in a vendor-specific way
//this is normally done inside a connection pool
XADataSource xads = ...//vendor-specific, see your JDBC vendor docs for info
XAConnection xaconn = xads.getXAConnection();

boolean rollback = false;
try {
       //GENERIC: begin and retrieve tx
       tm.begin();
       Transaction tx = tm.getTransaction();

       //SPECIFIC FOR JDBC: get the XAResourc from the JDBC connection
       XAResource xares = xaconn.getXAResource();

       //GENERIC: enlist the resource with the transaction
       //NOTE: this will only work if you set the configuration parameter:
       //com.atomikos.icatch.automatic_resource_registration=true
       //or, alternatively, if you use the UserTransactionService
       //integration mode explained later
       tx.enlistResource ( xares );
```

```
            //SPECIFIC FOR JDBC: access the database, the work will be
            //subject to the outcome of the current transaction
            ...

            //GENERIC: delist the resource
            tx.delistResource ( xares , XAResource.TMSUCCESS );

        }
        catch ( Exception e ) {
            rollback = true;
            throw e;
        }
        finally {
            //GENERIC: ALWAYS terminate the tx
            if ( rollback ) tm.rollback();
            else tm.commit();

            //SPECIFIC FOR JDBC: only now close the connection
            //i.e., not until AFTER commit or rollback!
            xaconn.close();
        }
```

Note that the XAConnection in the code fragment above was not closed until AFTER the transaction committed/rolledback. This is necessary because the transaction manager needs the connection to stay open until after two-phase commit is done. Otherwise, there is no way that the transaction manager can talk to the database any more (notice that the transaction manager is using the XAResource instance - which in turn relies on the XAConnection). If this scenario is unrealistic for your application, then we recommend that you use the UserTransactionService approach outlined in the next section.

4.3. For Sophisticated Needs: The Atomikos UserTransactionService

This is the most sophisticated and flexible approach for using Atomikos TransactionsEssentials: it gives you full control over almost any aspect of the transaction service. In this guide, we only explain the basics to get you started. More information can be found in the Atomikos API Guide. This section is outlined as follows:

- Getting a UserTransactionService Instance
- Overriding static propertie
- Explicit Resource Registration and Recovery
- Registering a LogAdministrator
- Explicit Startup and Shutdown
- Getting the UserTransaction
- Getting the TransactionManager

4.3.1. Getting a UserTransactionService Instance

The interface *com.atomikos.icatch.config.UserTransactionService* is a proprietary interface of Atomikos. The reason for this is simply that no current standard defines how to setup and initialize a transaction service. This interface is the key to doing the things that are outlined in this section; it is essential for using some of the more sophisticated features of Atomikos TransactionsEssentials. Getting an instance that implements this interface is done by constructing an object of class *com.atomikos.icatch.config.UserTransactionServiceImp*:

Example 4.7. Constructing the UserTransactionService object

```
com.atomikos.icatch.config.UserTransactionService uts =
    new com.atomikos.icatch.config.UserTransactionServiceImp();
```

Like all the transaction manager objects we have discussed so far, you can have as many instances of this class as you like. They are all equivalent - except for doing explicit startup of the transaction service (which can depend on instance-specific properties, as we will see shortly). Therefore, Atomikos recommends that you limit your number of instances to *one*.

4.3.2. Overriding static properties

Overriding static properties is done via an object of type *com.atomikos.icatch.config.TSInitInfo*. You can create such an object by calling the method *createTSInitInfo* on the UserTransactionService object:

Example 4.8. Creating a TSInitInfo object

```
com.atomikos.icatch.config.TSInitInfo info = uts.createTSInitInfo();

//use the info object to supplement or override the static configuration file
info.setProperty ( "com.atomikos.icatch.checkpoint_interval" , "2000" );
```

You will also need an instance of TSInitInfo for initializing (starting) the transaction service (see later in this section).

4.3.3. Explicit Resource Registration and Recovery

The procedures outlined in this subsection only work correctly if the transaction service is initialized with parameter *com.atomikos.icatch.automatic_resource_registration* set to *false*!!!

One of the most interesting features of the UserTransactionService is that you can explicitly register resources for recovery and online transaction processing. In other words, this allows you to gain fine-grained control over what resources should be recovered, how they should be recovered or used, and when recovery should happen.

Explicit resource registration is especially recommended in those cases where the vendor-specific XAResource implementations are not fully JTA/XA compliant. The most common deviation from the

specification is that the XAResource.isSameRM() method does not work correctly. This can lead to warning messages in the transaction manager's console file when you use one of the previous approaches to programming with Atomikos TransactionsEssentials.

The types of resources that are supported for this approach are the following:

- JDBC/XADataSource: use class *com.atomikos.datasource.xa.jdbc.JdbcTransactionalResource*.
- JMS/XAQueueConnectionFactory: use class *com.atomikos.datasource.xa.jms.JmsTransactionalResource*.
- JCA: use class *com.atomikos.datasource.xa.jca.JcaTransactionalResource*.

Note that there is no equivalent for NonXADataSource here: using non-XA JDBC drivers is not recoverable, so explicit registration is not relevant for such drivers.

Each of these resource types is similar to the others, the only real difference is that they use a different type of underlying connector to get XAConnection objects from when needed. This is all internal to the implementations and is of no further concern here. All that matters is that you know how to construct an instance when needed, and how to register. The following shows how to do this for JDBC; the other cases are completely analogous:

Example 4.9. Explicitly registering a (JDBC) resource for recovery.

```
//Get a vendor-specific instance of an XADataSource
XADataSource xaDataSource = ... //vendor-specific for your JDBC driver

//construct a corresponding resource that uses the XADataSource
//for recovery and normal transaction processing; give it a
//unique name for recovery and identification in the transaction service
JdbcTransactionalResource jdbcResource =
    new JdbcTransactionalResource (
    "com.mycompany.unique.name" ,
    xaDataSource );

//VERY IMPORTANT: register the resource with the transaction service
//to enable recovery; must be done BEFORE you startup!
uts.registerResource ( jdbcResource );
```

Add a separate resource for each connector (DataSource, QueueFactory or JCA resource adapter) that you expect to use. While you can also register resources after startup, you can expect better and smoother workings of the transaction service when you register before startup.

This shows the essence of what you should know about resource registration. More information about these classes can be found in the javadoc included in the installation.

4.3.4. Registering a LogAdministrator

An additional feature of Atomikos TransactionsEssentials is that you can optionally register one or more instances of *com.atomikos.icatch.admin.LogAdministrator*. Instances of this interface are supplied with a

com.atomikos.icatch.admin.LogControl This is an interface towards the transaction manager that outlines administrative tasks that are available when the transaction service is running. The main purpose of these tasks is purging the logs from old transactions, and forcing problematic transactions to terminate one way or another.

A couple of standard implementations of LogAdministrator are supplied out-of-the-box by Atomikos: a JmxLogAdministrator for JMX administration environments, a SimpleLogAdministrator for general UI environments, and a LocalLogAdministrator for Swing administration of the transaction service. By registering an instance of one of these, you will be able to retrieve the LogControl after startup, and get access to the administration features of the transaction service. See the Atomikos API guide for more information.

4.3.5. Explicit Startup and Shutdown

Having come this far, we can now look at how to perform explicit startup and shutdown of the transaction service. Going back to the UserTransactionService instance gotten earlier, the following illustrates how to do this:

Example 4.10. Explicit startup and shutdown

```
UserTransactionService uts = ...//see earlier
TSInitInfo info = uts.createTSInitInfo();

//override properties, register resources, register logadministrator
...

//startup of transaction service
uts.init ( info );

//here, the transaction service is running and has recovered all resources
//registered before

//NOTE: any resource you register will still be recovered, but this is not
//recommended in all cases

//the application is ready for doing ebusiness
...

//shutdown the transaction service,
//but wait for active transactions to complete
uts.shutdown ( false )
//if you don't want to wait, use uts.shutdown ( true );
```

You can call init and shutdown as many times as you want without exiting your application. For best results, we recommend that you use the same UserTransactionService instance while doing this.

4.3.6. Getting the UserTransaction

The UserTransactionService can be used to get a *javax.transaction.UserTransaction* (which should only be used after startup):

Example 4.11. Getting the UserTransaction via the UserTransactionService

```
//startup the transaction service
//uts.init ( info );

//now, the UserTransaction is available as follows:
javax.transaction.UserTransaction utx = uts.getUserTransaction();
```

Like all our JTA objects, this UserTransaction can be bound in JNDI where available.

> Although the previously outlined approach of using an instance of *com.atomikos.icatch.jta.UserTransactionImp* will still work, this is not recommended here: doing so could lead to accidental startup of the transaction service (by the auto-initialization feature) and that could interfere with the intention of explicitly controlling startup through the UserTransactionService.

4.3.7. Getting the TransactionManager

Likewise, *javax.transaction.TransactionManager* can be gotten like this (and should also be used after startup only):

Example 4.12. Getting the TransactionManager via the UserTransactionService

```
//startup the transaction service
//uts.init ( info );

//now, the TransactionManager is available as follows:
javax.transaction.TransactionManager tm = uts.getTransactionManager();
```

Again, this can be bound in JNDI where available.

4.3.8. Questions

4.3.8.1. Question 1

If you don't need to use your own connection pooling for JDBC, then what is the *easiest* way to start using TransactionsEssentials in your application:

1. Construct an instance of com.atomikos.icatch.jta.UserTransactionImp
2. Construct an instance of com.atomikos.icatch.jta.UserTransactionManager
3. Construct an instance of com.atomikos.icatch.jta.UserTransactionServiceImp

4.3.8.2. Question 2

If you don't want to use Atomikos connection pooling, then what is the *easiest* way to start using Atomikos TransactionsEssentials in your application:

1. Construct an instance of com.atomikos.icatch.jta.UserTransactionImp
2. Construct an instance of com.atomikos.icatch.jta.UserTransactionManager
3. Construct an instance of com.atomikos.icatch.jta.UserTransactionServiceImp

4.3.8.3. Question 3

Which of the following require the use of com.atomikos.icatch.jta.UserTransactionServiceImp?

1. Explicitly registering resources for recovery and online processing
2. Explicit control over startup and shutdown of the transaction service
3. Export or import a transaction in RMI/IIOP networks
4. Enlist/delist of XAResource instances

Appendix A. Answers

A.1. Chapter 2: Answers

A.1.1. Question 1

You can not control the joint outcome of a database update and a JMS message publication by merely ordering the executions. For instance, if you do a successful database update first and then try to send a JMS message, then what do you do if the JMS part fails? You could argue that the JDBC and the JMS allow you to explicitly control commit and rollback, but that does not change the real problem: in what order should the commits of the database and the message send be done? If you commit the database first, there is no way to go back if the later JMS commit fails. A similar argument shows that the JMS can not be committed first either. Controlling the order of executions or the order of commits is not a solution; that is why two-phase commit is used.

A.1.2. Question 2

The exception is of type *HeuristicMixedException*. The transaction manager is cut off from the in-doubt database, so it does not know whether the database will make a heuristic decision or not, and if it does then it could be either commit or rollback. A violation of the requirement that all resources have the same outcome is possible.

A.1.3. Question 3

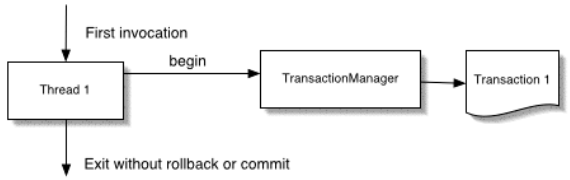
The method *setRollbackOnly* can be called almost anytime. The method *rollback* should only be called after all resources that were enlisted have also been delisted accordingly. Otherwise, XA-level errors may occur during rollback.

A.1.4. Question 4

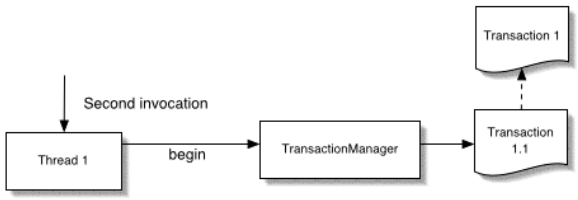
If a SQLException happens in a database update, there is a possibility that the database state for the active transaction is corrupt or does not correspond to what is expected from the point of view of the application. The transaction should be rolled back to make sure that the database is restored to the previous and correct state.

A.1.5. Question 5

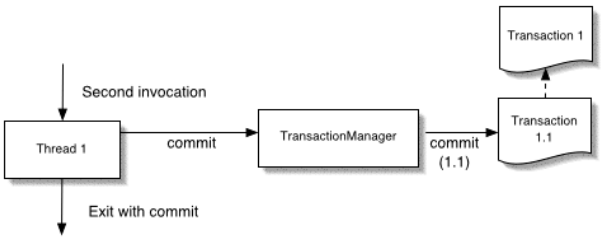
The JTA transaction manager associates the thread to the transaction you create. This means that if you call TransactionManager.begin() at the beginning of the method invocation then the executing thread has a JTA transaction. If you don't terminate this transaction then it will become a pending active transaction, subject to JTA rollback after timeout (and this will happen). *Before* the timeout, other requests may be executed in the *same* thread. If that happens then the second request's TransactionManager.begin() will create a *subtransaction* of the still pending transaction. So even if the second invocation calls commit, this will be a commit of a subtransaction, whose effects will not become permanent until the parent transaction commits. But the latter will not happen because the parent is a pending active transaction that will be rolled back if it times out. This is illustrated in the picture below.



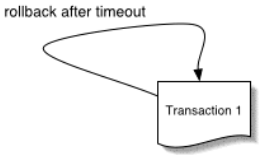
1. PENDING TRANSACTION 1



2. SECOND INVOCATION: CREATE TRANSACTION 1.1



3. SECOND INVOCATION: COMMIT TRANSACTION 1.1



4. ROLLBACK OF INVOCATIONS 1 AND 2

A.2. Chapter 3: Answers

A.2.1. Question 1

The logs will be written in the directory where you start your application. See the sample configuration file listing, which shows the default values. These are the values used if no other information is available.

A.2.2. Question 2

The average log file size will be larger. The checkpoint interval is the number of writes to the sequential log file before this file is cleaned up, so if this number is higher then the average log file size will increase as well. The cleanup encompasses purging the log by deleting information about terminated transactions.

A.2.3. Question 3

The parameter *com.atomikos.icatch.service* needs to be specified in every configuration file.

A.3. Chapter 4: Answers

A.3.1. Question 1

The easiest way is to construct an instance of com.atomikos.icatch.jta.UserTransactionImp (and use this in combination with AtomikosDataSourceBean).

A.3.2. Question 2

Construct an instance of com.atomikos.icatch.jta.UserTransactionManager and make your application call enlistResource/delistResource for each XAResource that you access.

A.3.2.1. Question 3

Which of the following require the use of com.atomikos.icatch.jta.UserTransactionServiceImp?

1. Explicitly registering resources for recovery and online processing: *yes*
2. Explicit control over startup and shutdown of the transaction service: *yes*
3. Export or import a transaction in RMI/IIOP networks: *yes*
4. Enlist/delist of XAResource instances: *no: our UserTransactionManager can do that*

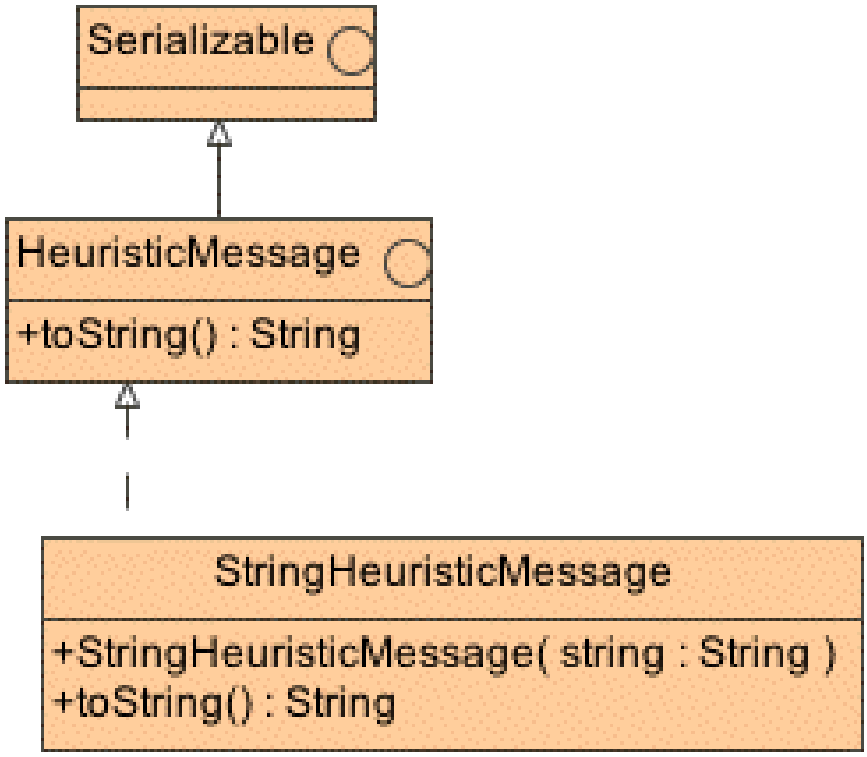
Appendix B. Getting More out of Atomikos TransactionsEssentials

There are some Atomikos-specific features that allow you to get more out of using TransactionsEssentials. Most of them are related to heuristic problem cases, and information that is available to resolve them. An explanation of these features follows next.

B.1. The HeuristicMessage Interface

One of the patent-pending features of Atomikos is the ability to include *application-level* comments in the transaction logs.

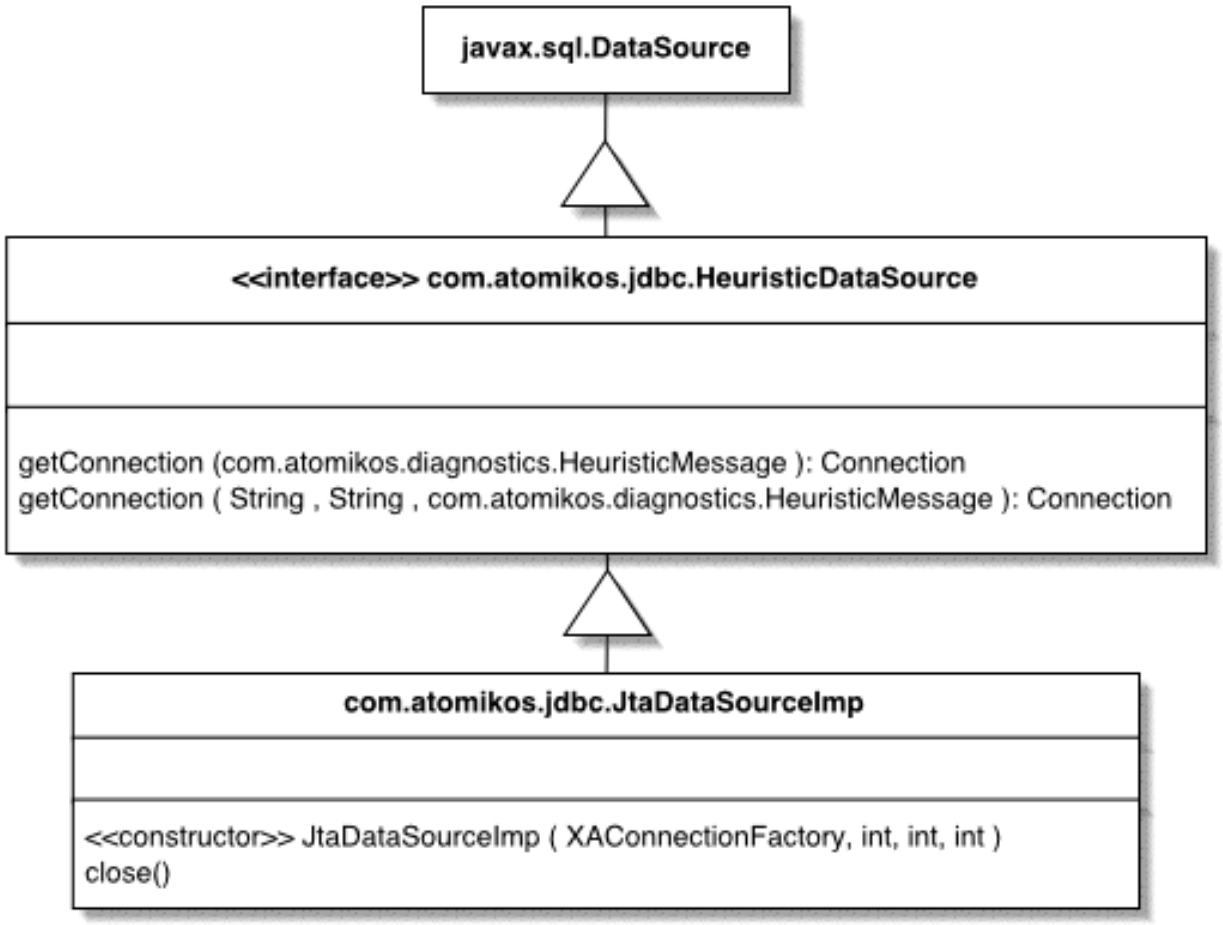
Heuristic exceptions can be well-documented with their effects on the business-level: if you add an application-level comment for each interaction with a data source or JMS queue, then the occurrence of a heuristic exception will allow the corresponding comments to be retrieved from the logs. The result is that heuristic exceptions include their application-level effects, thereby easing administration. The comment's interface type is discussed in this section; the later sections show how to add them to the interactions (and to the logs).



The figure above shows the basic interface for the kind of comments we mentioned. The interface is *com.atomikos.icatch.HeuristicMessage* and the supplied implementation class is *com.atomikos.icatch.StringHeuristicMessage*. Instances of this implementation class can be used to document interactions through JDBC or JMS. The string content will be saved in the logs along with the transaction log information.

B.2. JDBC: The HeuristicDataSource Interface

For JDBC interactions, the heuristic messages can be added at the time of getting a connection from an Atomikos DataSource. That is possible because our DataSource not only implements the interface *javax.sql.DataSource*, but also the Atomikos interface *com.atomikos.jdbc.HeuristicDataSource*, shown below.

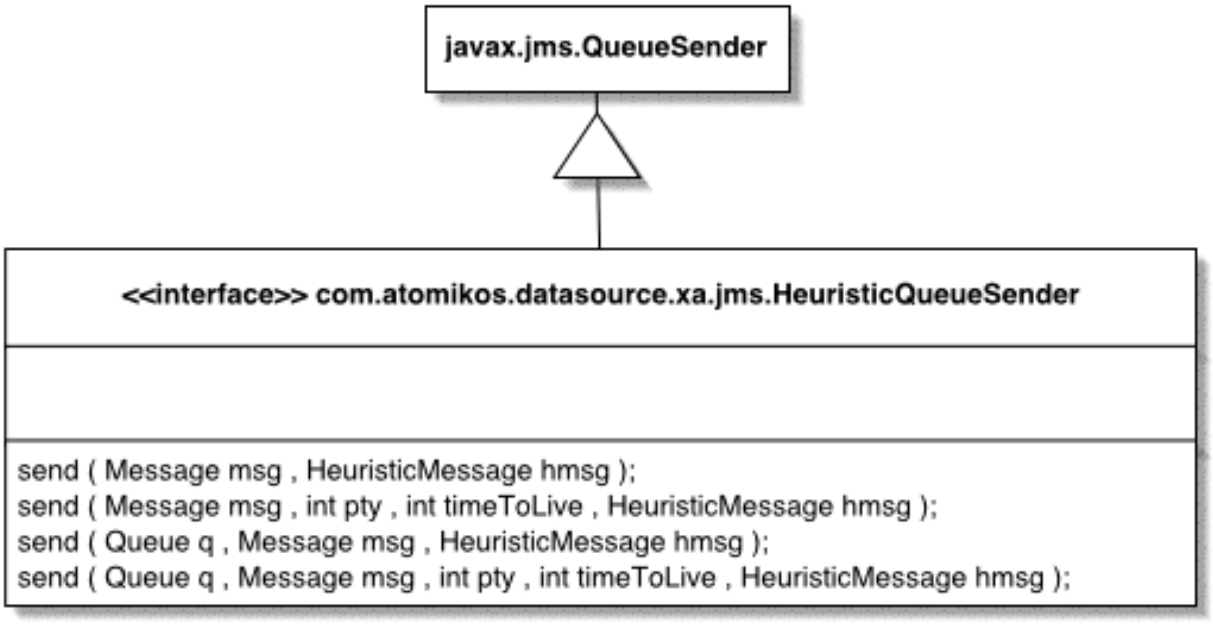


Instead of merely getting a connection from an Atomikos DataSource (as done with regular DataSource instances), you can use the getConnection method with one extra parameter that supplies a HeuristicMessage (or a String as from release 2.0). This will add the message to the logs in case of JDBC.

> You have to use the JDBC with Atomikos DataSource for this approach to work.

B.3. JMS: The HeuristicQueueSender Interface

For JMS send operations, the addition of a heuristic message is possible because the QueueSender instances that you create through the adapter classes are actually of type *com.atomikos.datasource.xa.jms.HeuristicQueueSender*.

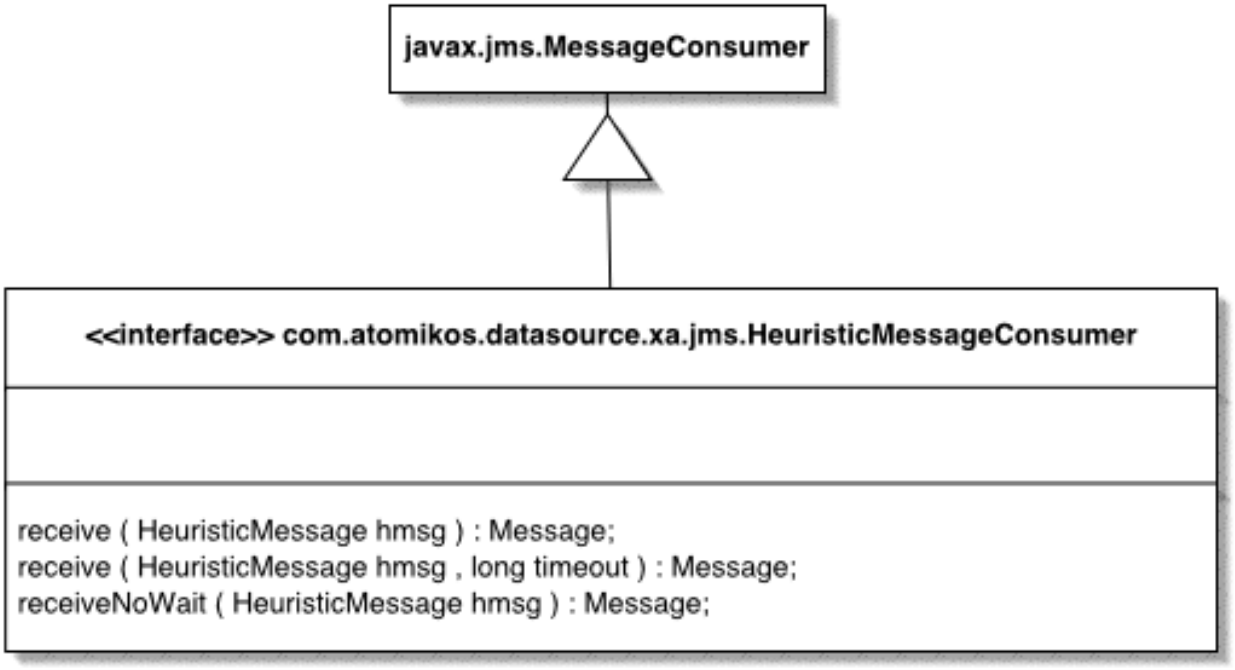


By performing a cast to this interface upon calling *QueueSession.createSender(queue)* you can gain access to the extra functionality.

> You have to use the JMS with Atomikos' JMS adapters for this approach to work.

B.4. JMS: The HeuristicQueueReceiver Interface

For JMS receive operations, the addition of a heuristic message is possible because the QueueReceiver instances that you create through the adapter classes are also of type *com.atomikos.datasource.xa.jms.HeuristicMessageConsumer*.



By performing a cast to this interface upon calling *QueueSession.createReceiver(queue)* you can gain access to the extra functionality.

> You have to use the JMS with Atomikos' JMS adapters for this approach to work.

Appendix C. Using Atomikos TransactionsEssentials in (Web) Application Servers

Atomikos TransactionsEssentials can also be used in web containers or application servers. Configuration is straightforward if you are using Spring (see the examples): you merely need to configure Spring to use Atomikos TransactionsEssentials.

Appendix D. Troubleshooting

For searching known problems and solutions, or in order to conctact technical support, please go to http://www.atomikos-support.com and login to our support forums.

Appendix E. References

- http://java.sun.com: Sun's Java website with the JTA, JDBC and JMS specifications and extra information.
- http://www.atomikos.com: Atomikos' website; please check regularly for updates and support information.
- Distributed Transaction Processing: The XA Specification (ISBN 1-872630-24-3). Published by The Open Group (http://www.opengroup.org).