JAKARTA EE

Jakarta Transactions 2.0

Jakarta Transactions Team, https://projects.eclipse.org/projects/ee4j.jta

2.0, October 08, 2020

Table of Contents

Copyright . . . 2
Eclipse Foundation Specification License . . . 2
Disclaimers . . . 2
Jakarta Transactions Specification, Version 2.0 . . . 4
1. Introduction . . . 5
1.1. Background . . . 5
1.2. Target Audience . . . 6
2. Relationship to Other Java APIs . . . 8
2.1. Java SE . . . 8
2.2. Jakarta Enterprise Beans . . . 8
2.3. JDBC . . . 8
2.4. Jakarta Messaging . . . 8
2.5. Java Transaction Service . . . 9
3. Jakarta Transactions API . . . 10
3.1. User Transaction Interface . . . 10
3.2. TransactionManager Interface . . . 11
3.2.1. Starting a Transaction . . . 12
3.2.2. Completing a Transaction . . . 12
3.2.3. Suspending and Resuming a Transaction . . . 12
3.3. Transaction Interface . . . 13
3.3.1. Resource Enlistment . . . 13
3.3.2. Transaction Synchronization . . . 15
3.3.3. Transaction Completion . . . 15
3.3.4. Transaction Equality and Hash Code . . . 15
3.4. XAResource Interface . . . 16
3.4.1. Opening a Resource Manager . . . 17
3.4.2. Closing a Resource Manager . . . 18
3.4.3. Thread of Control . . . 18
3.4.4. Transaction Association . . . 18
3.4.5. Externally Controlled connections . . . 19
3.4.6. Resource Sharing . . . 19
3.4.7. Local and Global Transactions . . . 20
3.4.8. Failure Recovery . . . 21
3.4.9. Identifying Resource Manager Instance . . . 21
3.4.10. Dynamic Registration . . . 22
3.5. Xid Interface . . . 22

3.6. TransactionSynchronizationRegistry Interface ... 23
3.7. Transactional Annotation ... 23
3.8. TransactionScoped Annotation ... 25
4. Jakarta Transactions Support in the Application Server ... 29
4.1. Connection-Based Resource Usage Scenario ... 29
4.2. Transaction Association and Connection Request Flow ... 31
4.3. Other Requirements ... 33
Appendix A: Related Documents ... 34
Appendix B: Revision History ... 35
B.1. Changes for Version 2.0 ... 35
B.2. Changes for Version 1.3 ... 35
B.3. Changes for Version 1.2 ... 35
B.4. Changes for Version 1.1 ... 35
B.5. Changes for Version 1.0.1B ... 36

```
Specification: Jakarta Transactions 2.0

Version: 2.0

Status: Final Release

Release: October 08, 2020
```

Copyright

Copyright (c) 2018, 2020 Eclipse Foundation. https://www.eclipse.org/legal/efsl.php

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation, Inc. <<url to this license>>"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) [$date-of-document] Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY

PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Jakarta Transactions Specification, Version 2.0

Chapter 1. Introduction

This document describes Jakarta Transactions. Jakarta Transactions specifies local Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the resource manager, and the application server.

The Jakarta Transactions package consists of two parts:

- A high-level application interface that allows a transactional application to demarcate transaction boundaries
- A high-level transaction manager interface that allows an application server to control transaction boundary demarcation for an application being managed by the application server

> Note – The Jakarta Transactions interfaces are presented as high-level from the transaction manager's perspective. In contrast, a low-level API for the transaction manager consists of interfaces that are used to implement the transaction manager. For example, the Java mapping of the OTS are low-level interfaces used internally by a transaction manager.

1.1. Background

Distributed transaction services in Enterprise Java middleware involves five players: the transaction manager, the application server, the resource manager, the application program, and the communication resource manager. Each of these players contributes to the distributed transaction processing system by implementing different sets of transaction APIs and functionalities.

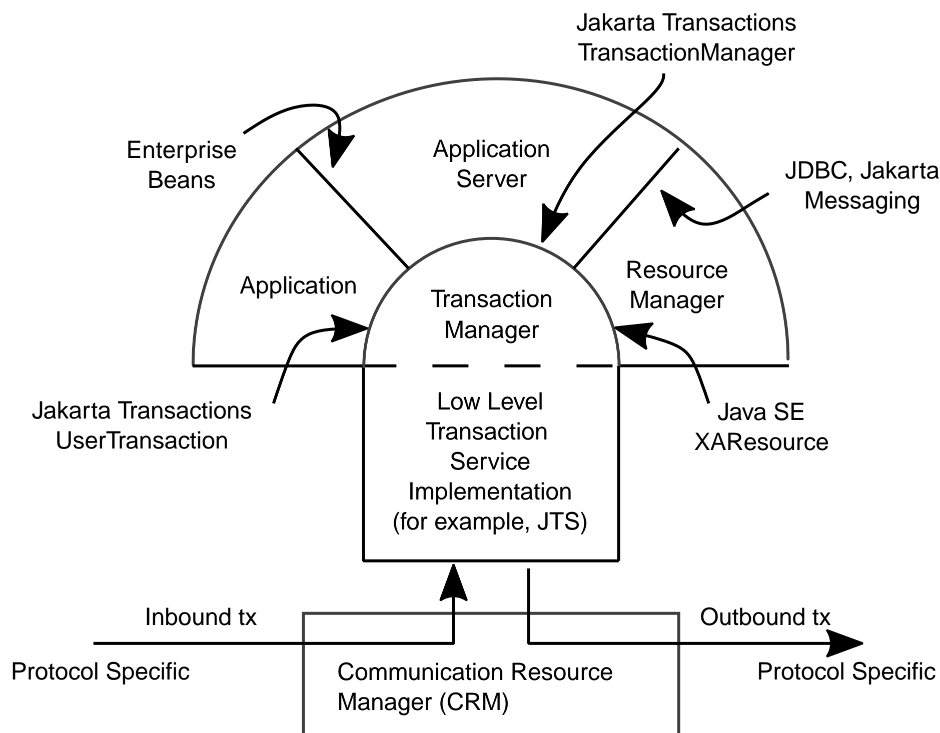
- A transaction manager provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.
- An application server (or TP monitor) provides the infrastructure required to support the application run-time environment which includes transaction state management. An example of such an application server is an Jakarta Enterprise Beans server.
- A resource manager (through a resource adapter) provides the application access to resources. The resource manager participates in distributed transactions by implementing a transaction resource interface used by the transaction manager to communicate transaction association, transaction completion and recovery work. An example of such a resource manager is a relational database server.
- A component-based transactional application that is developed to operate in a modern application server environment relies on the application server to provide transaction management support through declarative transaction attribute settings. An example of this type of applications is an application developed using the industry standard Jakarta Enterprise Beans component architecture. In addition, some other stand-alone Java client programs may wish to control their

transaction boundaries using a high-level interface provided by the application server or the transaction manager.

- A communication resource manager (CRM) supports transaction context propagation and access to the transaction service for incoming and outgoing requests. The Jakarta Transactions document does not specify requirements pertained to communication. Refer to the "*JTS Specification*" [2] for more details on interoperability between transaction managers.

From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed; only high-level interfaces need to be defined to allow transaction demarcation, resource enlistment, synchronization and recovery process to be driven from the users of the transaction services. The purpose of Jakarta Transactions is to define the local Java interfaces required for the transaction manager to support transaction management in the Java enterprise distributed computing environment.

In the diagram shown below, the small half-circle represents the Jakarta Transactions specification. Chapter 3 of the document describes each portion of the specification in details.



1.2. Target Audience

This document is intended for implementors of:

- Transaction managers, such as JTS
- Resource adapters, such as JDBC drivers and Jakarta Messaging providers

- Transactional resource managers, such as RDBMS
- Application servers, such as Jakarta EE Servers
- Advanced transactional applications written in the Java™ programming language

Chapter 2. Relationship to Other Java APIs

This chapter explores the relationship between Jakarta Transactions and other Java APIs, including Jakarta Enterprise Beans, JDBC API, Jakarta Messaging Service, and Java Transaction Service.

2.1. Java SE

Java SE provides the API that defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects in Jakarta Transactions transactions. The `javax.transaction.xa` package specifies this API consisting of the following types:

- `javax.transaction.xa.XAException`
- `javax.transaction.xa.XAResource`
- `javax.transaction.xa.Xid`

2.2. Jakarta Enterprise Beans

The Jakarta Enterprise Beans architecture requires that an Jakarta Enterprise Beans Container support application-level transaction demarcation by implementing the `jakarta.transaction.UserTransaction` interface. The `UserTransaction` interface is intended to be used by both the enterprise bean implementer (for beans with bean-managed transactions) and by the client programmer that wants to explicitly demarcate transaction boundaries within programs that are written in the Java programming language.

2.3. JDBC

A JDBC driver that supports distributed transactions implements the `javax.transaction.xa.XAResource` interface, the `javax.sql.XAConnection` interface, and the `javax.sql.XADataSource` interface. Refer to the "*JDBC 4.3 Specification*" for further details.

2.4. Jakarta Messaging

Jakarta Transactions may be used by a Jakarta Messaging provider to support distributed transactions. A Jakarta Messaging provider that supports the `XAResource` interface is able to participate as a resource manager in a distributed transaction processing system that uses a two-phase commit transaction protocol. In particular, a Jakarta Messaging provider implements the `javax.transaction.xa.XAResource` interface, the `jakarta.jms.XAConnection` interface, and the `jakarta.jms.XASession` interface. Refer to the "*Jakarta Messaging 3.0 Specification*" for further details.

2.5. Java Transaction Service

Java Transaction Service (JTS) is a specification for building a transaction manager which supports the Jakarta Transactions interfaces at the high-level and the standard Java mapping of the CORBA Object Transaction Service 1.1 specification at the low-level. JTS provides transaction interoperability using the CORBA standard IIOP protocol for transaction propagation between servers. JTS is intended for vendors that provide the transaction system infrastructure for enterprise middleware.

Chapter 3. Jakarta Transactions API

The Jakarta Transactions API consists of three elements: a high-level application transaction demarcation interface, a high-level transaction manager interface intended for an application server, and a standard Java mapping of the X/Open XA protocol intended for a transactional resource manager. This chapter specifies each of these elements in detail.

3.1. User Transaction Interface

The `jakarta.transaction.UserTransaction` interface provides the application the ability to control transaction boundaries programmatically.

The implementation of the `UserTransaction` object must be both `javax.naming.Referenceable` and `java.io.Serializable`, so that the object can be stored in all JNDI naming contexts.

The following example illustrates how an application component acquires and uses a `UserTransaction` object via injection.

```
@Resource UserTransaction userTransaction;

public void updateData() {

 // Start a transaction.
 userTransaction.begin();

 // ...

 // Perform transactional operations on data
 // Commit the transaction.
 userTransaction.commit();

}
```

The following example illustrates how an application component acquires and uses a `UserTransaction` object using a JNDI lookup.

```
public void updateData() {

 // Obtain the default initial JNDI context.
 Context context = new InitialContext();

 // Look up the UserTransaction object.
 UserTransaction userTransaction = (UserTransaction)
  context.lookup("java:comp/UserTransaction");

 // Start a transaction.
 userTransaction.begin();

 // ...

 // Perform transactional operations on data
 // Commit the transaction.
 userTransaction.commit();

}
```

The `UserTransaction.begin` method starts a global transaction and associates the transaction with the calling thread. The transaction-to-thread association is managed transparently by the transaction manager.

Support for nested transactions is not required. The `UserTransaction.begin` method throws the `NotSupportedException` when the calling thread is already associated with a transaction and the transaction manager implementation does not support nested transactions.

Transaction context propagation between application programs is provided by the underlying transaction manager implementations on the client and server machines. The transaction context format used for propagation is protocol dependent and must be negotiated between the client and server hosts. For example, if the transaction manager is an implementation of the JTS specification, it will use the transaction context propagation format as specified in the CORBA OTS specification. Transaction propagation is transparent to application programs.

3.2. TransactionManager Interface

The `jakarta.transaction.TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed. For example, the Jakarta Enterprise Beans container manages the transaction states for transactional Jakarta Enterprise Beans components; the container uses the `TransactionManager` interface mainly to demarcate transaction boundaries where operations affect the calling thread's transaction context. The transaction manager maintains the transaction context association with threads as part of its internal data structure. A thread's transaction context is either `null` or it refers to a specific global transaction. Multiple threads may concurrently be associated with the same global transaction.

Support for nested tranactions is not required.

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

3.2.1. Starting a Transaction

The `TransactionManager.begin` method starts a global transaction and associates the transaction context with the calling thread.

If the `TransactionManager` implementation does not support nested transactions, the `TransactionManager.begin` method throws the `NotSupportedException` when the calling thread is already associated with a transaction.

The `TransactionManager.getTransaction` method returns the `Transaction` object that represents the transaction context currently associated with the calling thread. This `Transaction` object can be used to perform various operations on the target transaction. Examples of `Transaction` object operations are resource enlistment and synchronization registration. The `Transaction` interface is described in "See Transaction Interface."

3.2.2. Completing a Transaction

The `TransactionManager.commit` method completes the transaction currently associated with the calling thread. After the `commit` method returns, the calling thread is not associated with a transaction. If the `commit` method is called when the thread is not associated with any transaction context, the `TransactionManager` throws an exception. In some implementations, the commit operation is restricted to the transaction originator only. If the calling thread is not allowed to commit the transaction, the `TransactionManager` throws an exception.

The `TransactionManager.rollback` method rolls back the transaction associated with the current thread. After the `rollback` method completes, the thread is associated with no transaction.

3.2.3. Suspending and Resuming a Transaction

A call to the `TransactionManager.suspend` method temporarily suspends the transaction that is currently associated with the calling thread. If the thread is not associated with any transaction, a `null` object reference is returned; otherwise, a valid `Transaction` object is returned. The `Transaction` object can later be passed to the `resume` method to reinstate the transaction context association with the calling thread.

The `TransactionManager.resume` method re-associates the specified transaction context with the calling thread. If the transaction specified is a valid transaction, the transaction context is associated with the calling thread; otherwise, the thread is associated with no transaction.

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

If `TransactionManager.resume` is invoked when the calling thread is already associated with another transaction, the transaction manager throws the `IllegalStateException` exception.

> Note that some transaction manager implementations allow a suspended transaction to be resumed by a different thread. This feature is not required by Jakarta Transactions.

The application server is responsible for ensuring that the resources in use by the application are properly delisted from the suspended transaction. A resource delist operation triggers the transaction manager to inform the resource manager to disassociate the transaction from the specified resource object (`XAResource.end(TMSUSPEND)`).

When the application's transaction context is resumed, the application server ensures that the resource in use by the application is again enlisted with the transaction. Enlisting a resource as a result of resuming a transaction triggers the transaction manager to inform the resource manager to re-associate the resource object with the resumed transaction (`XAResource.start(TMRESUME)`). Refer to "See Resource Enlistment." and "See Transaction Association," for more details on resource enlistment and transaction association.

3.3. Transaction Interface

The `Transaction` interface allows operations to be performed on the transaction associated with the target object. Every global transaction is associated with one `Transaction` object when the transaction is created. The `Transaction` object can be used to:

- Enlist the transactional resources in use by the application.
- Register for transaction synchronization callbacks.
- Commit or rollback the transaction.
- Obtain the status of the transaction.

These functions are described in the sections below.

3.3.1. Resource Enlistment

An application server provides the application run-time infrastructure that includes transactional resource management. Transactional resources such as database connections are typically managed by the application server in conjunction with some resource adapter and optionally with connection pooling optimization. In order for an external transaction manager to coordinate transactional work performed by the resource managers, the application server must enlist and delist the resources used in the transaction.

Resource enlistment performed by an application server serves two purposes:

- It informs the transaction manager about the resource manager instance that is participating in the global transaction. This allows the transaction manager to inform the participating resource manager on transaction association with the work performed through the connection (resource) object.
- It enables the transaction manager to group the resource types in use by each transaction. The resource grouping allows the transaction manager to conduct the two-phase commit transaction protocol between the transaction manager and the resource managers, as defined by the X/Open XA specification.

For each resource in use by the application, the application server invokes the `enlistResource` method and specifies the `XAResource` object that identifies the resource in use.

The `enlistResource` request results in the transaction manager informing the resource manager to start associating the transaction with the work performed through the corresponding resource—by invoking the `XAResource.start` method. The transaction manager is responsible for passing the appropriate flag in its `XAResource.start` method call to the resource manager. The `XAResource` interface is described in “See XAResource Interface.”

If the target transaction already has another `XAResource` object participating in the transaction, the transaction manager invokes the `XAResource.isSameRM` method to determine if the specified `XAResource` represents the same resource manager instance. This information allows the transaction manager to group the resource managers that are performing work on behalf of the transaction.

If the `XAResource` object represents a resource manager instance that has seen the global transaction before, the transaction manager groups the newly registered resource together with the previous `XAResource` object and ensures that the same resource manager only receives one set of prepare-commit calls for completing the target global transaction.

If the `XAResource` object represents a resource manager that has not previously seen the global transaction, the transaction manager establishes a different transaction branch .[1] and ensures that this new resource manager is informed about the transaction completion with proper prepare-commit calls.

The `isSameRM` method is discussed in “See Identifying Resource Manager Instance.”

The `Transaction.delistResource` method is used to disassociate the specified resource from the transaction context in the target object. The application server invokes the `delistResource` method with the following two parameters:

- The `XAResource` object that represents the resource.
- A `flag` to indicate whether the delistment was due to:
    - The transaction being suspended (`TMSUSPEND`)
    - A portion of the work has failed (`TMFAIL`)

  - A normal resource release by the application (`TMSUCCESS`)

An example of `TMFAIL` could be the situation where an application receives an exception on its connection operation.

The delist request results in the transaction manager informing the resource manager to end the association of the transaction with the target `XAResource`. The flag value allows the application server to indicate whether it intends to come back to the same resource. The transaction manager passes the appropriate flag value in its `XAResource.end` method call to the underlying resource manager.

A container only needs to call `delistResource` to explicitly disassociate a resource from a transaction and it is not a mandatory container requirement to do so as a precondition to transaction completion. A transaction manager is, however, required to implicitly ensure the association of any associated XAResource is ended, via the appropriate `XAResource.end` call, immediately prior to completion; that is before prepare (or commit/rollback in the one-phase optimized case).

3.3.2. Transaction Synchronization

Transaction synchronization allows the application server to get notification from the transaction manager before and after the transaction completes. For each transaction started, the application server may optionally register a `jakarta.transaction.Synchronization` callback object to be invoked by the transaction manager:

- The `Synchronization.beforeCompletion` method is called prior to the start of the two-phase transaction commit process. This call is executed with the transaction context of the transaction that is being committed.
- The `Synchronization.afterCompletion` method is called after the transaction has completed. The status of the transaction is supplied in the parameter.

3.3.3. Transaction Completion

The `Transaction.commit` and `Transaction.rollback` methods allow the target object to be comitted or rolled back. The calling thread is not required to have the same transaction associated with the thread.

If the calling thread is not allowed to commit the transaction, the transaction manager throws an exception.

3.3.4. Transaction Equality and Hash Code

The transaction manager must implement the `Transaction` object’s `equals` method to allow comparison between the target object and another `Transaction` object. The `equals` method should return `true` if the target object and the parameter object both refer to the same global transaction.

For example, the application server may need to compare two `Transaction` objects when trying to reuse a resource that is already enlisted with a transaction. This can be done using the `equals` method.

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

Transaction someOtherTxObj = ...

// ..

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

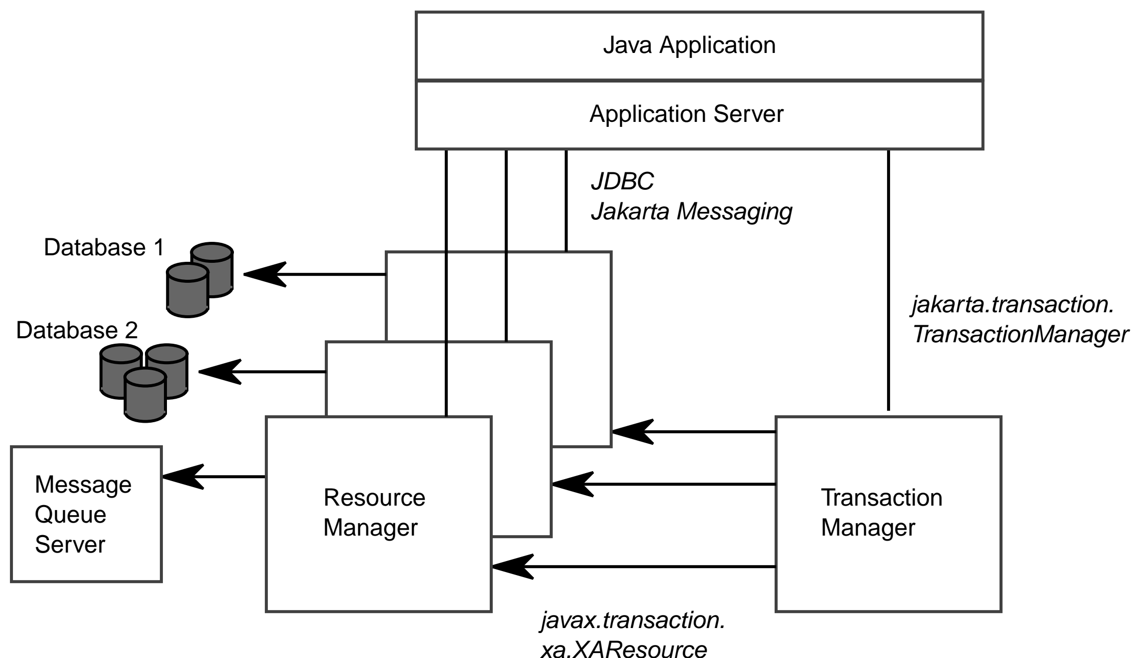
In addition, the transaction manager must implement the `Transaction` object's `hashCode` method so that if two `Transaction` objects are equal, they have the same hash code. However, the converse is not necessarily true. Two `Transaction` objects with the same hash code are not necessarily equal.

3.4. XAResource Interface

The `javax.transaction.xa.XAResource` interface is a Java mapping of the industry standard XA interface based on the X/Open CAE Specification (Distributed Transaction Processing: The XA Specification).

The `XAResource` interface defines the contract between a resource manager and a transaction manager in a distributed transaction processing (DTP) environment. A resource adapter for a resource manager implements the `XAResource` interface to support association of a global transaction to a transaction resource, such as a connection to a relational database.

A global transaction is a unit of work that is performed by one or more resource managers in a DTP system. Such a system relies on an external transaction manager, such as Java Transaction Service (JTS), to coordinate transactions.



The `XAResource` interface can be supported by any transactional resource adapter that is intended to be used by application programs in an environment where transactions are controlled by an external transaction manager. An example of such a resource is a database management system. An application may access data through multiple database connections. Each database connection is associated with an `XAResource` object that serves as a proxy object to the underlying resource manager instance. The transaction manager obtains an `XAResource` for each transaction resource participating in a global transaction. It uses the `start` method to associate the global transaction with the resource, and it uses the `end` method to disassociate the transaction from the resource. The resource manager is responsible for associating the global transaction with all work performed on its data between the `start` and `end` method invocations.

At transaction commit time, these transactional resource managers are informed by the transaction manager to prepare, commit, or rollback the transaction according to the two-phase commit protocol.

The `XAResource` interface, in order to be better integrated with the Java environment, differs from the standard X/Open XA interface in the following ways:

- The resource manager initialization is done implicitly by the resource adapter when the resource (connection) is acquired. There is no `xa_open` equivalent in the `XAResource` interface. This obviates the need for a resource manager to provide a different syntax to open a resource for use within the distributed transaction environment from the syntax used in the environment without distributed transactions.
- `Rmid` is not passed as an argument. We use an object-oriented approach where each `Rmid` is represented by a separate `XAResource` object.
- Asynchronous operations are not supported. Java supports multi-threaded processing and most databases do not support asynchronous operations.
- Error return values that are caused by the transaction manager's improper handling of the `XAResource` object are mapped to Java exceptions via the `XAException` class.
- The DTP concept of "Thread of Control" maps to all Java threads that are given access to the `XAResource` and `Connection` objects. For example, it is legal (although in practice rarely used) for two different Java threads to perform the `start` and `end` operations on the same `XAResource` object.
- Association migration and dynamic registration (optional X/Open XA features) are not supported. We've omitted these features for a simpler `XAResource` interface and simpler resource adapter implementation.

3.4.1. Opening a Resource Manager

The X/Open XA interface specifies that the transaction manager must initialize a resource manager (`xa_open`) prior to any other `xa_` calls. We believe that the knowledge of initializing a resource manager should be embedded within the resource adapter that represents the resource manager. The transaction manager does not need to know how to initialize a resource manager. The transaction manager is only responsible for informing the resource manager about when to start and end work associated with a global transaction and when to complete the transaction.

The resource adapter is responsible for opening (initializing) the resource manager when the connection to the resource manager is established.

3.4.2. Closing a Resource Manager

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

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

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

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

3.4.3. Thread of Control

The X/Open XA interface specifies that the transaction association related `xa_` calls must be invoked from the same thread context. This thread-of-control requirement is not applicable to the object-oriented component-based application run-time environment, in which application threads are dispatched dynamically at method invocation time. Different Java threads may be using the same connection resource to access the resource manager if the connection spans multiple method invocations. Depending on the implementation of the application server, different Java threads may be involved with the same `XAResource` object. The resource context and the transaction context may be operated independent of thread context. This means, for example, that it's possible for different threads to be invoking the `XAResource.start` and `XAResource.end` methods.

If the application server allows multiple threads to use a single `XAResource` object and the associated connection to the resource manager, it is the responsibility of the application server to ensure that there is only one transaction context associated with the resource at any point of time.

Thus the `XAResource` interface specified in this document requires that the resource managers be able to support the two-phase commit protocol from any thread context.

3.4.4. Transaction Association

Global transactions are associated with a transactional resource via the `XAResource.start` method, and disassociated from the resource via the `XAResource.end` method. The resource adapter is responsible for internally maintaining an association between the resource connection object and the `XAResource`

object. At any given time, a connection is associated with a single transaction or it is not associated with any transaction at all.

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

`XAResource` does not support nested transactions. It is an error for the `XAResource.start` method to be invoked on a connection that is currently associated with a different transaction.

Table 1. Transaction Association

XAResource Methods	**XAResource Transaction States**		
	Not Associated (T_0)	**Associated (T_1)**	**Associaton Suspended (T_2)**
`start()`	T_1		
`start(TMRESUME)`			T_1
`start(TMJOIN)`	T_1		
`end(TMSUSPEND)`		T_2	
`end(TMFAIL)`		T_0	T_0
`end(TMSUCCESS)`		T_0	T_0

3.4.5. Externally Controlled connections

Resources for transactional applications, whose transaction states are managed by an application server, must also be managed by the application server so that transaction association is performed properly. If an application is associated with a global transaction, it is an error for the application to perform transactional work through the connection without having the connection's resource object already associated with the global transaction. The application server must ensure that the `XAResource` object in use is associated with the transaction. This is done by invoking the `Transaction.enlistResource` method.

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

3.4.6. Resource Sharing

When the same transactional resource is used to interleave multiple transactions, it is the

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

The resource adapter must be able to handle multiple threads invoking the `XAResource` methods concurrently for transaction commit processing. For example, suppose we have a transactional resource `r1`. Global transaction `xid1` was *started* and *ended* with `r1`. Then a different global transaction `xid2` is associated with `r1`. Meanwhile, the transaction manager may start the two phase commit process for `xid1` using `r1` or any other transactional resource connected to the same resource manager. The resource adapter needs to allow the commit process to be executed while the resource is currently associated with a different global transaction.

The sample code below illustrates the above scenario:

```
// Suppose we have some transactional connection-based
// resource r1 that is connected to an enterprise
// information service system.
XAResource xares = r1.getXAResource();

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

...

xares.end(xid1); // dissociate xid1 frm the connection

...

xares.start(xid2); // associate xid2 to the connection

...

// While the connection is associated with xid2,
// the transaction manager starts the commit process
// for xid1
status = xares.prepare(xid1);

...

xares.commit(xid1, false);
```

3.4.7. Local and Global Transactions

The resource adapter is encouraged to support the usage of both local and global transactions within the same transactional connection. Local transactions are transactions that are started and

coordinated by the resource manager internally. The `XAResource` interface is not used for local transactions.

When using the same connection to perform both local and global transactions, the following rules apply:

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

If a resource adapter does not support mixing local and global transactions within the same connection, the resource adapter should throw the resource specific exception. For example, `java.sql.SQLException` is thrown to the application if the resource manager for the underlying RDBMS does not support mixing local and global transactions within the same JDBC connection.

3.4.8. Failure Recovery

During recovery, the transaction manager must be able to communicate to all resource managers that are in use by the applications in the system. For each resource manager, the transaction manager uses the `XAResource.recover` method to retrieve the list of transactions that are currently in a prepared or heuristically completed state.

Typically, the system administrator configures all transactional resource factories that are used by the applications deployed on the system. An example of such a resource factory is the JDBC `XADataSource` object, which is a factory for the JDBC `XAConnection` objects. The implementation of these transactional resource factory objects are both `javax.naming.Referenceable` and `java.io.Serializable` so that they can be stored in all JNDI naming contexts.

Because `XAResource` objects are not persistent across system failures, the transaction manager needs to have some way to acquire the `XAResource` objects that represent the resource managers which might have participated in the transactions prior to the system failure. For example, a transaction manager might, through the use of the JNDI lookup mechanism and cooperation from the application server, acquire an `XAResource` object representing each of the resource managers configured in the system. The transaction manager then invokes the `XAResource.recover` method to ask each resource manager to return any transactions that are currently in a prepared or heuristically completed state. It is the responsibility of the transaction manager to ignore transactions that do not belong to it.

3.4.9. Identifying Resource Manager Instance

The `isSameRM` method is invoked by the transaction manager to determine if the target `XAResource` object represents the same resource manager instance as that represented by the `XAResource` object in the parameter. The `isSameRM` method returns *true* if the specified target object is connected to the same resource manager instance; otherwise, the method returns *false*. The semi-pseudo code below illustrates the intended usage.

```
public boolean enlistResource(XAResource xares) {
...

 // Assuming xid1 is the target transaction and
 // xid1 already has another resource object xaRes1
 // participating in the transaction
 boolean sameRM = xares.isSameRM(xaRes1);

 if (sameRM) {
 //
 // Same underlying resource manager instance,
 // group together with xaRes1 and join the transaction
 //
 xares.start(xid1, TMJOIN);
 } else {
 //
 // This is a different resource manager instance,
 // make a new transaction branch for xid1
 //
 Xid xid1NewBranch = makeNewBranch(xid1);
 xares.start(xid1NewBranch, TMNOFLAGS);
 }
 ...

}
```

3.4.10. Dynamic Registration

Dynamic registration is not supported in `XAResource` because of the following reasons:

- In the Java component-based application server environment, connections to the resource manager are acquired dynamically when the application explicitly requests for a connection. These resources are enlisted with the transaction manager on an “as-needed” basis (unlike the static `xa_switch` table that exists in the C-XA procedural model).
- If a resource manager requires a way to dynamically register its work to the global transaction, then the implementation can be done at the resource adapter level via a private interface between the resource adapter and the underlying resource manager.

3.5. Xid Interface

The `javax.transaction.xa.Xid` interface is a Java mapping of the X/Open transaction identifier XID structure. This interface specifies three accessor methods which are used to retrieve a global transaction’s format ID, a global transaction ID, and a branch qualifier. The `Xid` interface is used by the transaction manager and the resource managers. This interface is not visible to the application

programs nor the application server.

3.6. TransactionSynchronizationRegistry Interface

The `jakarta.transaction.TransactionSynchronizationRegistry` interface is intended for use by system level application server components such as persistence managers. This provides the ability to register synchronization objects with special ordering semantics, associate resource objects with the current transaction, get the transaction context of the current transaction, get current transaction status, and mark the current transaction for rollback.

This interface is implemented by the application server as a stateless service object. The same object can be used by any number of components with complete thread safety. In standard application server environments, an instance implementing this interface can be looked up via JNDI using a standard name.

The user of `getResource` and `putResource` methods is a library component that manages transaction-specific data on behalf of a caller. The transaction-specific data provided by the caller is not immediately flushed to a transaction-enlisted resource, but instead is cached. The cached data is stored in a transaction-related data structure that is in a zero-or-one-to-one relationship with the transactional context of the caller.

An efficient way to manage such a transaction-related data structure is for the implementation of the `TransactionSynchronizationRegistry` to manage a Map for each transaction as part of the transaction state.

The keys of this Map are objects that are provided by the library components (users of the API). The values of the Map are any values that the library components are interested in storing, for example the transaction-related data structures. This Map has no concurrency issues since it is a dedicated instance for the transaction. When the transaction completes, the Map is cleared, releasing resources for garbage collection.

The scalability of the library code is significantly enhanced by the addition of the `getResource` and `putResource` methods to the `TransactionSynchronizationRegistry`.

3.7. Transactional Annotation

The `jakarta.transaction.Transactional` annotation provides the application the ability to declaratively control transaction boundaries on Jakarta Context Dependency Injection managed beans, as well as classes defined as managed beans by the Jakarta EE specification, at both the class and method level where method level annotations override those at the class level. See the "*Jakarta Enterprise Beans 4.0 specification*" for restrictions on the use of `@Transactional` with Jakarta Enterprise Beans resources. This support is provided via an implementation of Jakarta Context Dependency Injection interceptors that conduct the necessary suspending, resuming, etc. The `Transactional` interceptor interposes on business method invocations only and not on lifecycle events. Lifecycle methods are invoked in an unspecified transaction context. If an attempt is made to call any method of the `UserTransaction`

interface from within the scope of a bean or method annotated with `@Transactional` and a `Transactional.TxType` other than `NOT_SUPPORTED` or `NEVER`, an `IllegalStateException` must be thrown. The use of the `UserTransaction` is allowed within life cycle events. The use of the `TransactionSynchronizationRegistry` is allowed regardless of any `@Transactional` annotation. The `Transactional` interceptors must have a priority of `Interceptor.Priority.PLATFORM_BEFORE+200`. Refer to the "*Interceptors specification*" for more details.

The `TxType` element of the annotation indicates whether a bean method is to be executed within a transaction context where the values provide the following corresponding behavior and `TxType.REQUIRED` is the default:

- `TxType.REQUIRED`: If called outside a transaction context, the interceptor must begin a new Jakarta Transactions transaction, the managed bean method execution must then continue inside this transaction context, and the transaction must be completed by the interceptor.
  If called inside a transaction context, the managed bean method execution must then continue inside this transaction context.
- `TxType.REQUIRES_NEW`: If called outside a transaction context, the interceptor must begin a new Jakarta Transactions transaction, the managed bean method execution must then continue inside this transaction context, and the transaction must be completed by the interceptor.
  If called inside a transaction context, the current transaction context must be suspended, a new Jakarta Transactions transaction will begin, the managed bean method execution must then continue inside this transaction context, the transaction must be completed, and the previously suspended transaction must be resumed.
- `TxType.MANDATORY`: If called outside a transaction context, a `TransactionalException` with a nested `TransactionRequiredException` must be thrown.
  If called inside a transaction context, managed bean method execution will then continue under that context.
- `TxType.SUPPORTS`: If called outside a transaction context, managed bean method execution must then continue outside a transaction context.
  If called inside a transaction context, the managed bean method execution must then continue inside this transaction context.
- `TxType.NOT_SUPPORTED`: If called outside a transaction context, managed bean method execution must then continue outside a transaction context.
  If called inside a transaction context, the current transaction context must be suspended, the managed bean method execution must then continue outside a transaction context, and the previously suspended transaction must be resumed by the interceptor that suspended it after the method execution has completed.
- `TxType.NEVER`: If called outside a transaction context, managed bean method execution must then continue outside a transaction context.
  If called inside a transaction context, a `TransactionalException` with a nested `InvalidTransactionException` must be thrown

By default checked exceptions do not result in the transactional interceptor marking the transaction

for rollback and instances of `RuntimeException` and its subclasses do. This default behavior can be modified by specifying exceptions that result in the interceptor marking the transaction for rollback and/or exceptions that do not result in rollback. The `rollbackOn` element can be set to indicate exceptions that must cause the interceptor to mark the transaction for rollback. Conversely, the `dontRollbackOn` element can be set to indicate exceptions that must not cause the interceptor to mark the transaction for rollback. When a class is specified for either of these elements, the designated behavior applies to subclasses of that class as well. If both elements are specified, `dontRollbackOn` takes precedence.

The following example will override behavior for application exceptions, causing the transaction to be marked for rollback for all application exceptions.

```
@Transactional(rollbackOn={Exception.class})
```

The following example will prevent transactions from being marked for rollback by the interceptor when an `IllegalStateException` or any of its subclasses reaches the interceptor.

```
@Transactional(dontRollbackOn={IllegalStateException.class})
```

The following will cause the transaction to be marked for rollback for all runtime exceptions and all `SQLException` types except for `SQLWarning`.

```
@Transactional(
        rollbackOn={SQLException.class},
        dontRollbackOn={SQLWarning.class})
```

The `TransactionalException` thrown from the `Transactional` interceptors implementation is a `RuntimeException` and therefore by default any transaction that was started as a result of a `Transactional` annotation earlier in the call stream will be marked for rollback as a result of the `TransactionalException` being thrown by the `Transactional` interceptor of the second bean. For example if a transaction is begun as a result of a call to a bean annotated with `Transactional(TxType.REQUIRES)` and this bean in turn calls a second bean annotated with `Transactional(TxType.NEVER)`, the transaction begun by the first bean will be marked for rollback.

3.8. TransactionScoped Annotation

The `jakarta.transaction.TransactionScoped` annotation provides the ability to specify a standard Jakarta Context Dependency Injection scope to define bean instances whose lifecycle is scoped to the currently active Jakarta Transactions transaction. This annotation has no effect on classes which have non-contextual references such those defined as managed beans by the Jakarta EE specification . The transaction scope is active when the return from a call to `UserTransaction.getStatus` or `TransactionManager.getStatus` is one of the following states:

```
Status.STATUS_ACTIVE
Status.STATUS_MARKED_ROLLBACK
Status.STATUS_PREPARED
Status.STATUS_UNKNOWN
Status.STATUS_PREPARING
Status.STATUS_COMMITTING
Status.STATUS_ROLLING_BACK
```

It is not intended that the term “active” as defined here in relation to the `TransactionScoped` annotation should also apply to its use in relation to transaction context, lifecycle, etc. mentioned elsewhere in this specification. The object with this annotation will be associated with the current active Jakarta Transactions transaction when the object is used. This association must be retained through any transaction suspend or resume calls as well as any `Synchronization.beforeCompletion` callbacks. Any `Synchronization.afterCompletion` methods will be invoked in an undefined context. The way in which the Jakarta Transactions transaction is begun and completed (for example via `UserTransaction`, `Transactional` interceptor, etc.) is of no consequence. The contextual references used across different Jakarta Transactions transactions are distinct. Refer to the “*Jakarta Context Dependency Injection 3.0 specification*” for more details on contextual references. A `jakarta.enterprise.context.ContextNotActiveException` must be thrown if a bean with this annotation is used when the transaction context is not active.

The following example test case illustrates the expected behavior.

`TransactionScoped` annotated Jakarta Context Dependency Injection managed bean:

```
@TransactionScoped

 public class TestCDITransactionScopeBean {

 public void test() {
 //...
 }

}
```

Test Class:

```
@Inject
UserTransaction userTransaction;
TransactionManager transactionManager;

@Inject
TestCDITransactionScopeBean testTxAssociationChangeBean;


public void testTxAssociationChange() throws Exception {
userTransaction.begin(); //tx1 begun
testTxAssociationChangeBean.test();

// assert testTxAssociationChangeBean instance has tx1
// association
Transaction transaction = transactionManager.suspend();

// tx1 suspended
userTransaction.begin(); //tx2 begun

testTxAssociationChangeBean.test();

// assert new testTxAssociationChangeBean  instance has
// tx2 association

userTransaction.commit();
// tx2 committed, assert no transaction scope is active

transactionManager.resume(transaction);
// tx1 resumed
testTxAssociationChangeBean.test();

// assert testTxAssociationChangeBean is original tx1
// instance and not still referencing committed/tx2 tx

userTransaction.commit();
// tx1 commit, assert no transaction scope is active

try {
 testTxAssociationChangeBean.test();
 fail("should have thrown ContextNotActiveException");
} catch (ContextNotActiveException contextNotActiveException) {
// do nothing intentionally
}
}
```

[1] Transaction Branch is defined in the X/Open XA spec as follows: “A global transaction has one or

more transaction branches. A branch is a part of the work in support of a global transaction for which the transaction manager and the resource manager engage in a separate but coordinated transaction commitment protocol. Each of the resource manager's internal units of work in support of a global transaction is part of exactly one branch. After the transaction manager begins the transaction commitment protocol, the resource manager receives no additional work to do on that transaction branch. The resource manager may receive additional work on behalf of the same transaction, from different branches. The different branches are related in that they must be completed atomically. Each transaction branch identifier (or XID) that the transaction manager gives the resource manager identifies both a global transaction and a specific branch. The resource manager may use this information to optimize its use of shared resources and locks."

Chapter 4. Jakarta Transactions Support in the Application Server

This chapter provides a discussion on implementation and usage considerations for application servers to support Jakarta Transactions. Our discussion assumes the application's transactions and resource usage are managed by the application server. We further assume that access to the underlying transactional resource manager is through some Java API implemented by the resource adapter representing the resource manager. For example, a JDBC driver may be used to access a relational database, a Jakarta Connectors resource adapter may be used to access an Enterprise Resource Planning (ERP) system, and so on. This section focuses on the usage of Jakarta Transactions and assumes a generic connection based transactional resource is in use without being specific about a particular type of resource manager.

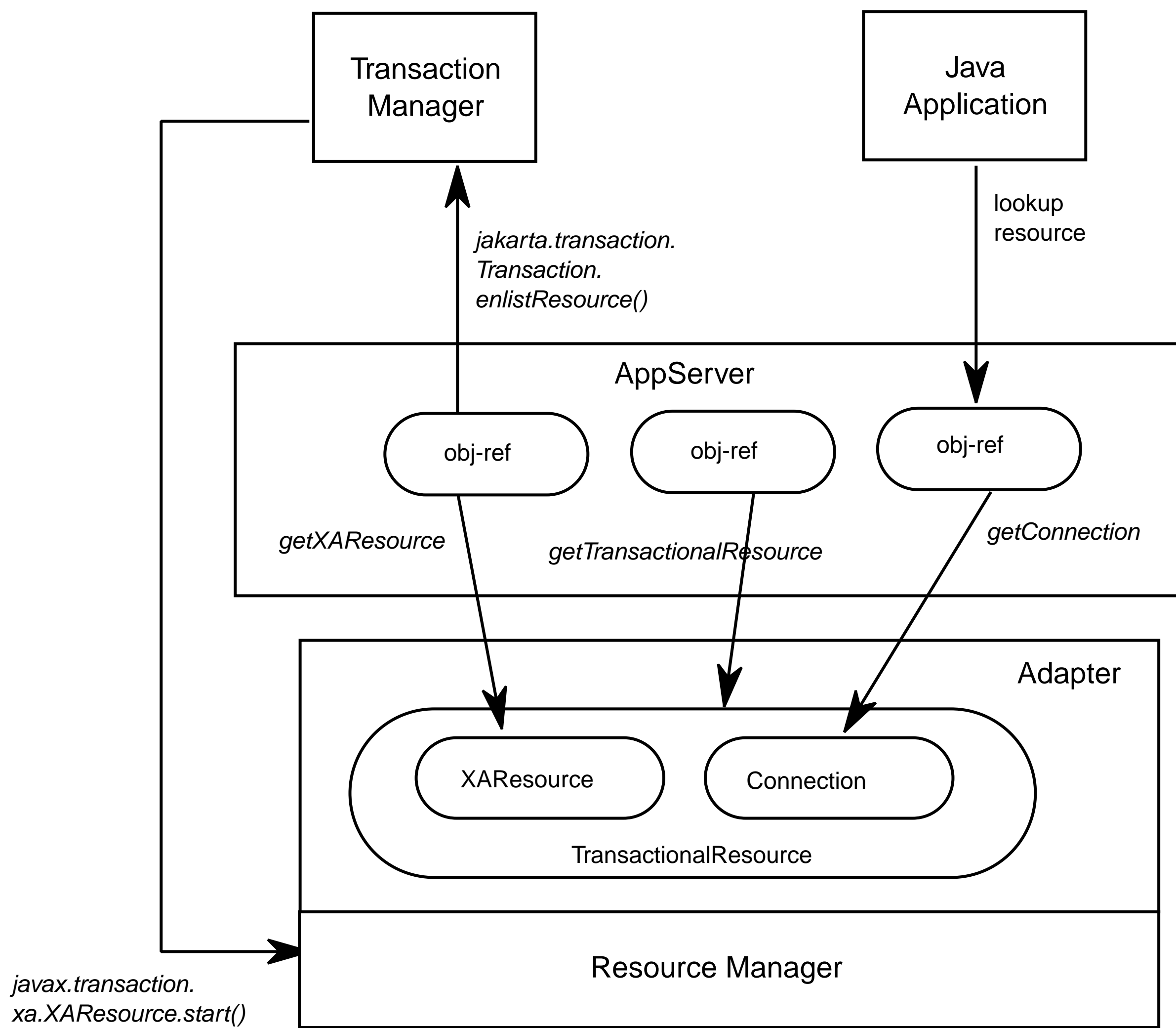
4.1. Connection-Based Resource Usage Scenario

Let's assume that the resource adapter provides a connection-based resource API called *TransactionalResource* to access the underlying resource manager.

In a typical usage scenario, the application server invokes the resource adapter's resource factory to create a *TransactionalResource* object. The resource adapter internally associates the *TransactionalResource* with two other entities: an object that implements the specific resource adapter's connection interface and an object that implements the `javax.transaction.xa.XAResource` interface.

The application server obtains a *TransactionalResource* object and uses it in the following way. The application server obtains the `XAResource` object via a `getXAResource` method. The application server enlists the `XAResource` to the transaction manager using the `Transaction.enlistResource` method. The transaction manager informs the resource manager to associate the work performed (through that connection) with the transaction currently associated with the application. The transaction manager does it by invoking the `XAResource.start` method.

The application server then invokes some `getConnection` method to obtain a `Connection` object and returns it to the application. Note that the `Connection` interface is implemented by the resource adapter and it is specific to the underlying resource supported by the resource manager. The diagram below illustrates a general flow of acquiring resource and enlisting the resource to the transaction manager.



In this usage scenario, the `XAResource` interface is transparent to the application program, and the `Connection` interface is transparent to the transaction manager. The application server is the only party that holds a reference to some *TransactionalResource* object.

The code sample below illustrates how the application server obtains the `XAResource` object reference and enlists it with the transaction manager.

```
// Acquire some connection-based transactional resource to
// access the resource manager

Context ctx = InitialContext();

ResourceFactory rf =(ResourceFactory)ctx.lookup("MyEISResource");

TransactionalResource res = rf.getTransactionalResource();

// Obtain the XAResource part of the connection and
// enlist it with the transaction manager

XAResource xaRes = res.getXAResource();
(TransactionManager.getTransaction()).enlistResource(xaRes);

// get the connection part of the transaction resource
Connection con = (Connection)res.getConnection();

// return the connection to the application
```

4.2. Transaction Association and Connection Request Flow

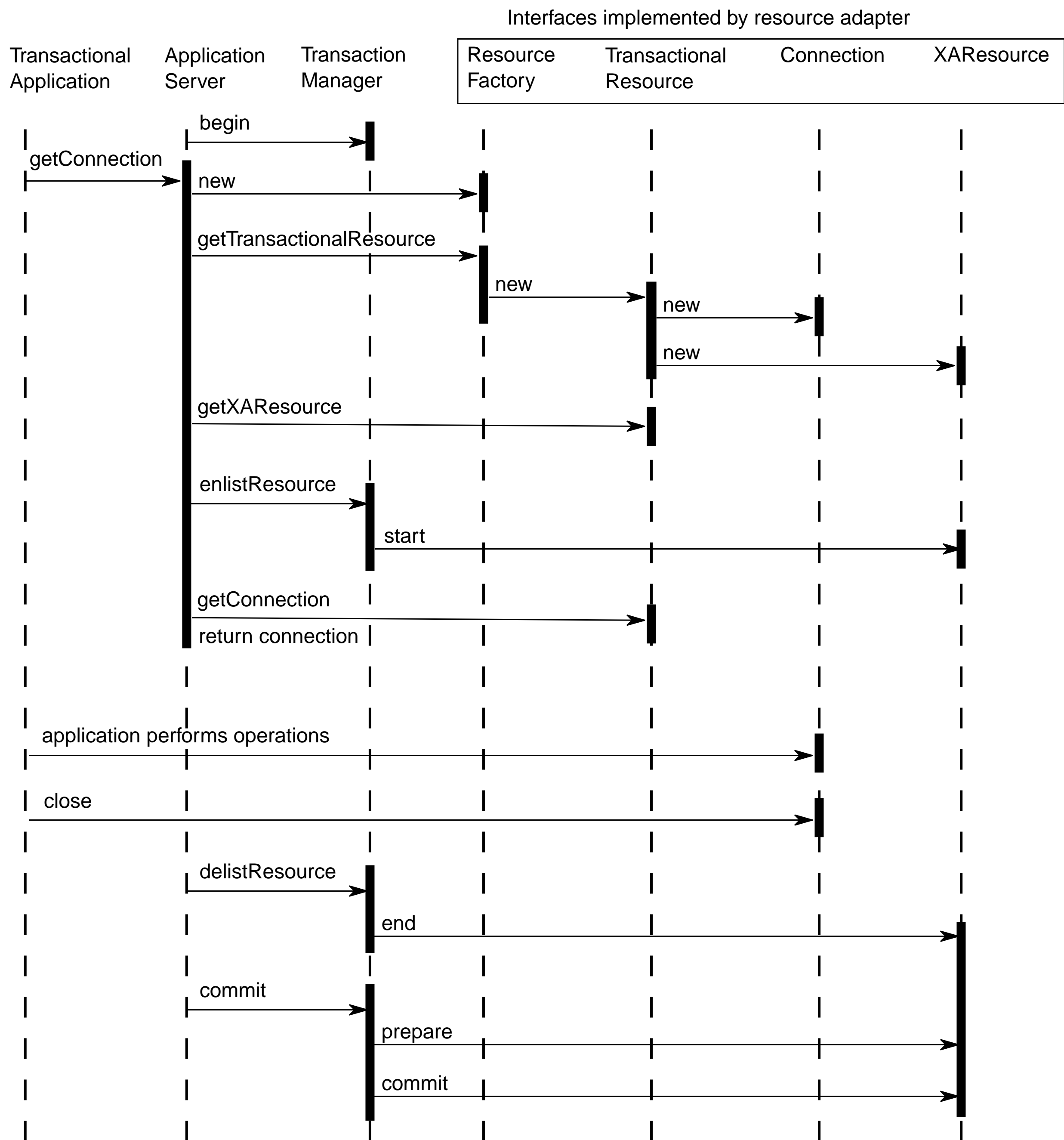
This session provides a brief walkthrough of how an application server may handle a connection request from the application. The figure that follows illustrates the usage of Jakarta Transactions. The steps shown are for illustrative purposes, they are not prescriptive:

1. Assuming a client invokes a Jakarta Context Dependency Injection managed bean annotated with `@Transactional(TxType.REQUIRED)` and the client is not associated with a global transaction, the `Transactional` interceptor starts a global transaction by invoking the `TransactionManager.begin` method.
2. After the transaction starts, the container invokes the bean method. As part of the business logic, the bean requests for a connection-based resource using the API provided by the resource adapter of interest.
3. The application server obtains a resource from the resource adapter via some `ResourceFactory.getTransactionalResource` method.
4. The resource adapter creates the *TransactionalResource* object and the associated `XAResource` and `Connection` objects.
5. The application server invokes the `getXAResource` method.
6. The application server enlists the resource to the transaction manager.
7. The transaction manager invokes `XAResource.start` to associate the current transaction to the

resource.

8. The application server invokes the `getConnection` method.
9. The application server returns the `Connection` object reference to the application.
10. The application performs one or more operations on the connection.
11. The application closes the connection.
12. The application server delists the resource when notified by the resource adapter about the connection close.
13. The transaction manager invokes `XAResource.end` to disassociate the transaction from the `XAResource`.
14. The application server asks the transaction manager to commit the transaction.
15. The transaction manager invokes `XAResource.prepare` to inform the resource manager to prepare the transaction work for commit.
16. The transaction manager invokes `XAResource.commit` to commit the transaction.

This example illustrates the application server's usage of the `TransactionManager` and `XAResource` interfaces as part of the application connection request handling.



4.3. Other Requirements

The behaviors described in the Javadoc specification of the Jakarta Transactions interfaces are required functionality and must be implemented by compliant providers.

Appendix A: Related Documents

This specification refers to the following documents.

1. X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3
2. *Java Transaction Service (JTS) Specification, Version 1.0 (Draft)*, available at https://download.oracle.com/otn-pub/jcp/7309-jts-1.0-spec-oth-JSpec/jts1_0-spec.pdf
3. OMG Object Transaction Service (OTS 1.1)
4. ORB Portability Submission, OMG document orbos/97-04-14
5. *Jakarta Enterprise Beans 4.0 Specification*, available at https://jakarta.ee/specifications/enterprise-beans/4.0/
6. *JDBC™ 4.3 Specification*, available at https://jcp.org/en/jsr/detail?id=221
7. *Jakarta Messaging 3.0 Specification*, available at https://jakarta.ee/specifications/messaging/3.0/
8. *Jakarta Context Dependency Injection 3.0 Specification*, available at https://jakarta.ee/specifications/cdi/3.0/
9. *Jakarta Interceptors 2.0 Specification*, available at *https://jakarta.ee/specifications/interceptors/2.0/*

Appendix B: Revision History

B.1. Changes for Version 2.0

- Changed some former references to Jakarta Transactions where appropriate.
- Updated references to Jakarta specifications where appropriate.
- Updated package references to `jakarta.*` where appropriate
- Small text update where two piece of text appeared to be incorrectly joined
- Removed version information from several places when referencing components of other Jakarta technologies

B.2. Changes for Version 1.3

- Remove the `javax.transaction.xa` types as they have been subsumed by Java SE.

B.3. Changes for Version 1.2

- New annotation `javax.transaction.Transactional` and exception `javax.transaction.TransactionalException`
- New annotation `javax.transaction.TransactionScoped`
- Added the following description to the end of "See Resource Enlistment": "A container only needs to call `delistResource` to explicitly dissociate a resource from a transaction and it is not a mandatory container requirement to do so as a precondition to transaction completion. A transaction manager is, however, required to implicitly insure the association of any associated `XAResource` is ended, via the appropriate `XAResource.end` call, immediately prior to completion; that is before prepare (or commit/rollback in the one-phase optimized case)."
- Various update of stale material, version updates, etc.

B.4. Changes for Version 1.1

- "See XAResource Interface": The line "The transaction manager obtains an `XAResource` for each resource manager participating in a global transaction." has been changed to "The transaction manager obtains an `XAResource` for each transaction resource participating in a global transaction.".
- Interface `javax.transaction.UserTransaction`, method `setTransactionTimeout`, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the begin method by the current thread.".
- Interface `javax.transaction.TransactionManager`, method `setTransactionTimeout`, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the begin method by the current thread.".

- New interface `javax.transaction.TransactionSynchronizationRegistry`
- Interface `javax.transaction.Synchronization`, method `beforeCompletion`, add the following description: "An unchecked exception thrown by a registered `Synchronization` object causes the transaction to be aborted. That is, upon encountering an unchecked exception thrown by a registered synchronization object, the transaction manager must mark the transaction for rollback.".

B.5. Changes for Version 1.0.1B

- Removed the method modifier `abstract` from all interface methods, since interface methods are implicitly abstract.
- Table 1, row 1 (`TMJOIN`) : move transaction association (`T1`) from column 3 (association suspended) to column 1 (not associated).
- Interface `javax.transaction.Synchronization` , method `beforeCompletion`, change the following phrase in the description "start of the transaction completion process" to "start of the two-phase transaction commit process".
- Interface `javax.transaction.Transaction`, method `commit`, added `IllegalStateException` to throws clause.
- Interface `javax.transaction.Transaction`, method `commit`, replace the description of `HeuristicRollbackException` with "Thrown to indicate that a heuristic decision was made and that all relevant updates have been rolled back.".
- Interface `javax.transaction.Transaction`, change spelling of `Transactioin` to `Transaction` in interface description.
- Interface `javax.transaction.Transaction`, method `registerSynchronization`, first paragraph, line 2, change the phrase "transaction completion process" to "two-phase transaction commit process".
- Interface `javax.transaction.Transaction, method `rollback`, spelling correction to method signature description, change `SyetemException` to `SystemException`.
- Interface `javax.transaction.TransactionManager`, method `commit`, replace the description of `HeuristicRollbackException` with "Thrown to indicate that a heuristic decision was made and that all relevant updates have been rolled back.".
- Interface `javax.transaction.TransactionManager`, method `setTransactionTimeout`, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the begin method.".
- Interface `javax.transaction.TransactionManager`, method `setTransactionTimeout`, replace the description of method parameter *seconds* with "The value of the timeout in seconds. If the value is zero, the transaction service restores the default value. If the value is negative a `SystemException` is thrown.".
- Interface `javax.transaction.UserTransaction`, method `commit`, replace the description of `HeuristicRollbackException` with "Thrown to indicate that a heuristic decision was made and that all relevant updates have been rolled back.".

- Interface `javax.transaction.UserTransaction`, method `setTransactionTimeout`, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the `begin` method.".
- Interface `javax.transaction.UserTransaction`, method `setTransactionTimeout`, replace the description of method parameter *seconds* with “The value of the timeout in seconds. If the value is zero, the transaction service restores the default value. If the value is negative a `SystemException` is thrown.”
- Interface `javax.transaction.xa.XAResource`, method `commit`, insert return type `void` to method signature description.
- Interface `javax.transaction.xa.XAResource`, method `commit`, spelling correction to description, change `paramether` to `parameter`.
- Interface `javax.transaction.xa.XAResource`, method `end`, replace return type `int` with `void` in method signature description.
- Interface `javax.transaction.xa.XAResource`, method `end`, corrected spelling of `XAException` errorCode `XAER_RMFAILED` to `XAER_RMFAIL`.
- Interface `javax.transaction.xa.XAResource`, method `recover`, spelling correction to method signature description, replace return type `xid[]` with `Xid[]`.
- Interface `javax.transaction.xa.XAResource`, method `rollback`, add the following to the description of `XAException`, "Possible `XAExceptions` are `XA_HEURHAZ`, `XA_HEURCOM`, `XA_HEURRB`, `XA_HEURMIX`, `XAER_RMERR`, `XAER_RMFAIL`, `XAER_NOTA`, `XAER_INVAL`, or `XAER_PROTO`. Upon return, the resource manager has rolled back the branch’s work and has released all held resources.".
- Interface `javax.transaction.xa.XAResource`, spelling correction to description, replace `TMNOFLAG` with `TMNOFLAGS`.
- Interface `javax.transaction.xa.XAResource`, added constants `XA_OK` and `XA_RDONLY` to be consistent with the actual interface definition.
- Interface `javax.transaction.xa.Xid`, method `getGlobalTransactionId`, spelling correction to method signature description, corrected method name from `getGrid` to `getGlobalTransactionId`.
- Interface `javax.transaction.xa.Xid`, method `getBranchQualifier`, spelling correction to method signature description, corrected method name from `getEqual` to `getBranchQualifier`.
- Class `javax.transaction.xa.XAException`, spelling correction to description of interface definition, replace phrase `javax.transaction.xa.XAEception` with `javax.transaction.xa.XAException`.