

CHAPTER 3

Java Transaction API

The Java Transaction 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 `javax.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.
    tx.commit();
}
```

CHAPTER 3

Java Transaction API

The Java Transaction 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 `javax.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.
    tx.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. **Use of `UserTransaction` from within a method or bean annotated with `@Transactional` is not restricted. For example, `UserTransaction` may be needed in these cases to mark the transaction for rollback or obtain the status of a current transaction. However, the application must ensure `UserTransaction` is not used in a way that will compromise the behavior of any transaction that may be managed by the container. In particular, `UserTransaction` should not be used to commit or rollback a transaction that was started by a transaction interceptor, as such an action might compromise the integrity of the application**

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,

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.
    tx.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.

it will use the transaction context propagation format as specified in the CORBA OTS specification. Transaction propagation is transparent to application programs.

3.2 Transaction Manager Interface

The `javax.transaction.TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed. For example, the EJB container manages the transaction states for transactional EJB 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 transactions 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 Transaction Manager 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 Section 3.3, "Transaction Interface."

3.2 Transaction Manager Interface

The `javax.transaction.TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed. For example, the EJB container manages the transaction states for transactional EJB 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 transactions 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 Transaction Manager 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 Section 3.3, "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

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 JTA.

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 JTA.

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));
```

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 Section 3.3.1, "Resource Enlistment," and Section 3.4.4, "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:

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 Section 3.3.1, "Resource Enlistment," and Section 3.4.4, "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

- 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 Section 3.4, “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 ID¹ and ensures that this new resource manager is informed about the transaction completion with proper prepare-commit calls.

The `isSameRM` method is discussed in Section 3.4.9, “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:

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 Section 3.4, “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 ID¹ and ensures that this new resource manager is informed about the transaction completion with proper prepare-commit calls.

The `isSameRM` method is discussed in Section 3.4.9, “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:

3.6 TransactionSynchronizationRegistry Interface

The `javax.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 `javax.transaction.Transactional` annotation provides the application the ability to declaratively control transaction boundaries on CDI managed beans, as well as classes defined as managed beans by the Java EE specification, at both the class and method level. Method level annotations override class level annotations.

3.6 TransactionSynchronizationRegistry Interface

The `javax.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 `javax.transaction.Transactional` annotation provides the application the ability to declaratively control transaction boundaries on CDI managed beans, as well as classes defined as managed beans by the Java EE specification, at both the

This support is provided via an implementation of CDI interceptors that conduct the necessary suspending, resuming, etc. The `Transactional` interceptor interposes on business method `invocation` and lifecycle events. Lifecycle methods are invoked in an unspecified transaction context unless the method is annotated explicitly with `@Transactional`. 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 JTA 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 JTA 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 JTA 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 `TransactionException` 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.

class and method level where method level annotations override those at the class level. See the EJB specification for restrictions on the use of `@Transactional` with EJBs. This support is provided via an implementation of CDI interceptors that conduct the necessary suspending, resuming, etc. The `Transactional` interceptor interposes on business method `invocations` and lifecycle events. Lifecycle methods are invoked in an unspecified transaction context unless the method is annotated explicitly with `@Transactional`. 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 JTA 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 JTA 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 JTA 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 `TransactionException` 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.

```
@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.

If an attempt to use the `UserTransaction` is made from within a bean or method annotated with `@Transactional`, an `IllegalStateException` must be thrown, however, use of the `TransactionSynchronizationRegistry` is allowed.

See the EJB specification for restrictions on the use of `@Transactional` with EJBs.

3.8 TransactionScoped Annotation

The `javax.transaction.TransactionScoped` annotation provides the ability to specify a standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction. This annotation cannot be used by classes defined as managed beans by the Java EE specification which have non-contextual references. 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
```

```
@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 `javax.transaction.TransactionScoped` annotation provides the ability to specify a standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction. This annotation has no effect on classes which have non-contextual references such those defined as managed beans by the Java 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 JTA transaction when the object is used. This association must be retained through any transaction suspend or resume calls as well as any

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 JTA 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. The transaction context must be destroyed after completion calls have been made on enlisted resources. Any `Synchronization.afterCompletion` methods will be invoked in an undefined context. The way in which the JTA transaction is begun and completed (for example via `UserTransaction`, `Transactional` interceptor, etc.) is of no consequence. The contextual references used across different JTA transactions are distinct. Refer to the CDI specification for more details on contextual references. A `javax.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 CDI managed bean:

```
@TransactionScoped
public class TestCDITransactionScopeBean {
    public void test()
    {
        //...
    }
}
```

Test Class:

`Synchronization.beforeCompletion` callbacks. The transaction context must be destroyed after completion calls have been made on enlisted resources. Any `Synchronization.afterCompletion` methods will be invoked in an undefined context. The way in which the JTA transaction is begun and completed (for example via `UserTransaction`, `Transactional` interceptor, etc.) is of no consequence. The contextual references used across different JTA transactions are distinct. Refer to the CDI specification for more details on contextual references. A `javax.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 CDI managed bean:

```
@TransactionScoped
public class TestCDITransactionScopeBean {
    public void test()
    {
        //...
    }
}
```

Test Class:

```
UserTransaction userTransaction;
TransactionManager transactionManager;
@Inject
TestCDITransactionScopeBean testTxAssociationChangeBean;

public void testTxAssociationChange() throws Exception {
    userTransaction.begin(); //tx1 begun
    testTxAssociationChangeBean.test();
    // assert testTxAssociationChangeBean instance has tx1
    // association
}
```

30

```

UserTransaction userTransaction;
TransactionManager transactionManager;
@Inject
TestCDITransactionScopeBean testTxAssociationChangeBean;

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

```

Public Review Draft

30

```

Transaction transaction = |
    transactionManager.suspend(); |
// tx1 suspended
userTransaction.begin(); //tx2 begun
testTxAssociationChangeBean.test();
// assert new testTxAssociationChangeBean instance has |
// tx2 association
userTransaction.commit(); |
// tx2 committed, assert notransaction scope is active
transactionManager.resume(tx); |
// tx1 resumed |
testTxAssociationChangeBean.test();
// assert testTxAssociationChangeBean is original tx1
// instance and not still referencing committed/tx2 tx
userTransaction.commit(); |
// tx1 commit, assert no transactionscope is active
try {
    testTxAssociationChangeBean.test();
    fail(
        "should have thrownContextNotActiveException");
} catch (ContextNotActiveException |
        contextNotActiveException) {
    // do nothing intentionally
}
}

```

Public Review Draft

```
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(tx); |
// 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 thrownContextNotActiveException");
} catch (ContextNotActiveException |
        contextNotActiveException) {
    // do nothing intentionally
}
}
```

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 **delist** 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.

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.

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, available at <http://www.oracle.com/technetwork/java/javaee/jts-spec095-1508547.pdf>
- [3] OMG Object Transaction Service (OTS 1.1)
- [4] ORB Portability Submission, OMG document orbos/97-04-14
- [5] Enterprise JavaBeans™ (EJB) 3.2 Specification, available at <http://jcp.org/en/jsr/detail?id=345>
- [6] JDBC™ 4.1 Specification, available at <http://jcp.org/en/jsr/detail?id=221>
- [7] JMS 2.0 Specification, available at <http://jcp.org/en/jsr/detail?id=343>
- [8] Contexts and Dependency Injection for the Java EE Platform (CDI) 1.1 Specification, available at <http://jcp.org/en/jsr/detail?id=346>

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, available at <http://www.oracle.com/technetwork/java/javaee/jts-spec095-1508547.pdf>
- [3] OMG Object Transaction Service (OTS 1.1)
- [4] ORB Portability Submission, OMG document orbos/97-04-14
- [5] Enterprise JavaBeans™ (EJB) 3.2 Specification, available at <http://jcp.org/en/jsr/detail?id=345>
- [6] JDBC™ 4.1 Specification, available at <http://jcp.org/en/jsr/detail?id=221>
- [7] JMS 2.0 Specification, available at <http://jcp.org/en/jsr/detail?id=343>
- [8] Contexts and Dependency Injection for the Java EE Platform (CDI) 1.1 Specification, available at <http://jcp.org/en/jsr/detail?id=346>
- [9] Interceptor Specification, available at <http://jcp.org/en/jsr/detail?id=318>