◆ ◆ ◆ CHAPTER 15

Using the Transaction Service

The Java EE platform provides several abstractions that simplify development of dependable transaction processing for applications. This chapter discusses Java EE transactions and transaction support in the Oracle GlassFish Server.

This chapter contains the following sections:

For more information about the Java Transaction API (JTA) and Java Transaction Service (JTS), see Chapter 18, "Administering Transactions," in *GlassFish Server Open Source Edition 3.1 Administration Guide* and the following sites: `http://java.sun.com/javaee/technologies/jta/index.jsp` and `http://java.sun.com/javaee/technologies/jts/index.jsp`.

You might also want to read Chapter 28, "Transactions," in *The Java EE 6 Tutorial*.

Handling Transactions with Databases

The following JDBC features pertain to transactions:

Using JDBC Transaction Isolation Levels

Not all database vendors support all transaction isolation levels available in the JDBC API. The GlassFish Server permits specifying any isolation level your database supports. The following table defines transaction isolation levels.

TABLE 15–1 Transaction Isolation Levels

Transaction Isolation Level	`getTransactionIsolation` Return Value	Description
`read-uncommitted`	1	Dirty reads, non-repeatable reads, and phantom reads can occur.
`read-committed`	2	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
`repeatable-read`	4	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
`serializable`	8	Dirty reads, non-repeatable reads and phantom reads are prevented.

By default, the transaction isolation level is undefined (empty), and the JDBC driver's default isolation level is used. You can specify the transaction isolation level in the following ways:

- Select the value from the Transaction Isolation drop-down list on the New JDBC Connection Pool or Edit Connection Pool page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--isolationlevel` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.
- Specify the `transaction-isolation-level` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.transaction-isolation-level=serializable
```

For more information, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.

Note that you cannot call `setTransactionIsolation` during a transaction.

You can set the default transaction isolation level for a JDBC connection pool. For details, see Creating a JDBC Connection Pool.

To verify that a level is supported by your database management system, test your database programmatically using the `supportsTransactionIsolationLevel` method in `java.sql.DatabaseMetaData`, as shown in the following example:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

For more information about these isolation levels and what they mean, see the JDBC API specification.

Setting or resetting the transaction isolation level for every `getConnection` call can degrade performance. So by default the isolation level is not guaranteed.

Applications that change the transaction isolation level on a pooled connection programmatically risk polluting the JDBC connection pool, which can lead to errors. If an application changes the isolation level, enabling the `is-isolation-level-guaranteed` setting in the pool can minimize such errors.

You can guarantee the transaction isolation level in the following ways:

- Check the Isolation Level Guaranteed box on the New JDBC Connection Pool or Edit Connection Pool page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--isisolationguaranteed` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.
- Specify the `is-isolation-level-guaranteed` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.is-isolation-level-guaranteed=true
```

For more information, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.

Using Non-Transactional Connections

You can specify a non-transactional database connection in any of these ways:

- Check the Non-Transactional Connections box on the New JDBC Connection Pool or Edit Connection Pool page in the Administration Console. The default is unchecked. For more information, click the Help button in the Administration Console.
- Specify the `----nontransactionalconnections` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.
- Specify the `non-transactional-connections` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.non-transactional-connections=true
```

For more information, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.

- Use the `DataSource` implementation in the GlassFish Server, which provides a `getNonTxConnection` method. This method retrieves a JDBC connection that is not in the scope of any transaction. There are two variants.

```
public java.sql.Connection getNonTxConnection() throws java.sql.SQLException

public java.sql.Connection getNonTxConnection(String user, String password)
   throws java.sql.SQLException
```

- Create a resource with the JNDI name ending in `__nontx`. This forces all connections looked up using this resource to be non transactional.

Typically, a connection is enlisted in the context of the transaction in which a `getConnection` call is invoked. However, a non-transactional connection is not enlisted in a transaction context even if a transaction is in progress.

The main advantage of using non-transactional connections is that the overhead incurred in enlisting and delisting connections in transaction contexts is avoided. However, use such connections carefully. For example, if a non-transactional connection is used to query the database while a transaction is in progress that modifies the database, the query retrieves the unmodified data in the database. This is because the in-progress transaction hasn't committed. For another example, if a non-transactional connection modifies the database and a transaction that is running simultaneously rolls back, the changes made by the non-transactional connection are not rolled back.

Here is a typical use case for a non-transactional connection: a component that is updating a database in a transaction context spanning over several iterations of a loop can refresh cached data by using a non-transactional connection to read data before the transaction commits.

Handling Transactions with Enterprise Beans

This section describes the transaction support built into the Enterprise JavaBeans programming model for the GlassFish Server.

As a developer, you can write an application that updates data in multiple databases distributed across multiple sites. The site might use EJB servers from different vendors. This section provides overview information on the following topics:

Flat Transactions

The Enterprise JavaBeans Specification, v3.0 requires support for flat (as opposed to nested) transactions. In a flat transaction, each transaction is decoupled from and independent of other transactions in the system. Another transaction cannot start in the same thread until the current transaction ends.

Flat transactions are the most prevalent model and are supported by most commercial database systems. Although nested transactions offer a finer granularity of control over transactions, they are supported by far fewer commercial database systems.

Global and Local Transactions

Both local and global transactions are demarcated using the javax.transaction.UserTransaction interface, which the client must use. Local transactions bypass the XA commit protocol and are faster. For more information, see "The Transaction Manager, the Transaction Synchronization Registry, and `UserTransaction`" on page 25.

Commit Options

The EJB protocol is designed to give the container the flexibility to select the disposition of the instance state at the time a transaction is committed. This allows the container to best manage caching an entity object's state and associating an entity object identity with the EJB instances.

There are three commit-time options:

- **Option A** – The container caches a ready instance between transactions. The container ensures that the instance has exclusive access to the state of the object in persistent storage.

  In this case, the container does *not* have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.

  Note – Commit option A is not supported for this GlassFish Server release.

- **Option B** – The container caches a ready instance between transactions, but the container does *not* ensure that the instance has exclusive access to the state of the object in persistent storage. This is the default.

  In this case, the container must synchronize the instance's state by invoking `ejbLoad` from persistent storage at the beginning of the next transaction.

- **Option C** – The container does *not* cache a ready instance between transactions, but instead returns the instance to the pool of available instances after a transaction has completed.

  The life cycle for every business method invocation under commit option C looks like this.

`ejbActivate` `ejbLoad` *business method* `ejbStore` `ejbPassivate`

If there is more than one transactional client concurrently accessing the same entity, the first client gets the ready instance and subsequent concurrent clients get new instances from the pool.

The `glassfish-ejb-jar.xml` deployment descriptor has an element, `commit-option`, that specifies the commit option to be used. Based on the specified commit option, the appropriate handler is instantiated.

Bean-Level Container-Managed Transaction Timeouts

The transaction timeout for the domain is specified using the Transaction Timeout setting of the Transaction Service. A transaction started by the container must commit (or rollback) within this time, regardless of whether the transaction is suspended (and resumed), or the transaction is marked for rollback. The default value, `0`, specifies that the server waits indefinitely for a transaction to complete.

To override this timeout for an individual bean, use the optional `cmt-timeout-in-seconds` element in `glassfish-ejb-jar.xml`. The default value, `0`, specifies that the Transaction Service timeout is used. The value of `cmt-timeout-in-seconds` is used for all methods in the bean that start a new container-managed transaction. This value is *not* used if the bean joins a client transaction.

Handling Transactions with the Java Message Service

The following JMS features pertain to transactions:

Transactions and Non-Persistent Messages

During transaction recovery, non-persistent messages might be lost. If the broker fails between the transaction manager's prepare and commit operations, any non-persistent message in the transaction is lost and cannot be delivered. A message that is not saved to a persistent store is not available for transaction recovery.

Using the ConfigurableTransactionSupport Interface

The Java EE Connector 1.6 specification allows a resource adapter to use the `transaction-support` attribute to specify the level of transaction support that the resource

adapter handles. However, the resource adapter vendor does not have a mechanism to figure out the current transactional context in which a ManagedConnectionFactory is used.

If a ManagedConnectionFactory implements an optional interface called com.sun.appserv.connectors.spi.ConfigurableTransactionSupport, the GlassFish Server notifies the ManagedConnectionFactory of the transaction-support configured for the connector connection pool when the ManagedConnectionFactory instance is created for the pool. Connections obtained from the pool can then be used with a transaction level at or lower than the configured value. For example, a connection obtained from a pool that is set to XA_TRANSACTION could be used as a LOCAL resource in a last-agent-optimized transaction or in a non-transactional context.

The Transaction Manager, the Transaction Synchronization Registry, and UserTransaction

To access a UserTransaction instance, you can either look it up using the java:comp/UserTransaction JNDI name or inject it using the @Resource annotation.

Accessing a DataSource using the Synchronization.beforeCompletion() method requires setting Allow Non Component Callers to true. The default is false. For more information about non-component callers, see "Allowing Non-Component Callers" on page .

If possible, you should use the javax.transaction.TransactionSynchronizationRegistry interface instead of javax.transaction.TransactionManager, for portability. You can look up the implementation of this interface by using the JNDI name java:comp/TransactionSynchronizationRegistry. For details, see the Javadoc page for Interface TransactionSynchronizationRegistry (http://java.sun.com/javaee/5/docs/api/javax/transaction/TransactionSynchronizationRegistry.html) and Java Specification Request (JSR) 907 (http://www.jcp.org/en/jsr/detail?id=907).

If accessing the javax.transaction.TransactionManager implementation is absolutely necessary, you can look up the GlassFish Server implementation of this interface using the JNDI name java:appserver/TransactionManager. This lookup should not be used by the application code.