Fundamentals of Java Enterprise Components

Session: 12

Transactions

Objectives

- Describe handling transactions in Java EE applications
- Describe container managed and bean managed transactions
- Describe managing transactions that update multiple databases
- Explain transaction management in Web applications and Web components
- Explain how to control concurrent data access through Persistence API

Transaction 1-2

Transaction refers to a series of actions performed on the database.

A transaction is supposed to have the following essential properties known as ACID properties:

- Atomicity
- Consistency
- Integrity
- Durability

Java EE provides a Java transaction API which allows applications to access transactions in a secure manner ensuring the integrity of data.

A transaction may end in two ways - commit or rollback.

Transaction 2-2

- Java supports programmatic transactions:
  - Bean Managed Transactions
  - Container Managed Transactions or Java Transaction API (JTA) transactions
- The lifecycle of the container managed transaction is managed by the container and in case of bean managed transaction, the transaction management is defined through programming.
- The Java Transaction API specifies an interface between the transaction manager and other components participating in the transaction.

Container Managed Transactions 1-2

- Container is responsible for starting and managing the transaction.
- It can work with a message driven bean or a session bean.
- The container is the demarcation of the transaction.
- The transaction characteristics are specified in the deployment descriptor through transaction attributes.
- The transactions in an enterprise bean are associated with the method in the bean.
- A method of the bean can be associated with only a single transaction, it does not allow multiple transactions.

Container Managed Transactions 2-2

Container managed transactions do not use methods such as commit, rollback which are part of java.sql.Connection and javax.jms.message.

Following are the methods which cannot be executed in container managed transactions:

- commit, setAutoCommit, and rollback methods of java.sql.Connection
- getUserTransaction of javax.ejb.EJBContext
- all methods of javax.transaction.UserTransaction

Transaction Attributes

A transaction attribute is required to define the scope of the enterprise bean methods and transactions associated with these methods in the deployment descriptor.

There are six possible values of a transaction attribute:

- **Required** - implies that the enterprise bean method must be executed in a transaction context.
- **RequiresNew** - implies that a method with this attribute requires a new transaction context.
- **Mandatory** - implies that the method must be invoked from a transaction context only. If not, container will throw TransactionRequriedException.
- **NotSupported** - implies that the method cannot execute in a transaction context.
- **Supports** - implies that if the method is invoked from a transactional context then the method will execute within that context.
- **Never** - implies that it should never be invoked from a transactional context. If it is executed within a transaction context, container will throw RemoteException.

Rolling Back a Container Managed Transaction

- The transaction is rolled back by the container whenever an exception is thrown. Following code demonstrates the structure of a Container Managed transaction:

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class StudentAdmissions {
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void addStudent(Connection c){
…..
}
{
```

Bean Managed Transactions 1-3

Bean managed transactions are used when a bean method has to associate with more than one transaction.

The code in the session bean or message driven bean defines the scope of the transaction.

The developer manages the lifecycle of the transaction through the bean.

Application managed transactions can be implemented through JDBC or JTA.

Bean managed transactions are implemented through the objects of the class UserTransaction.

Bean Managed Transactions 2-3

Following code demonstrates the skeletal structure of a method that implements a bean managed transaction:

```
import javax.transaction.UserTransaction;
@TransactionManagement(TransactionManagementType.BEAN)
class UserTrans{
…..
public void transactionMethod() {
  try {
    // obtain access to a UserTransaction
    UserTransaction tx = myEJBContext.getUserTransaction();
    // start a JTA transaction
    tx.begin();
    // call other objects and resources to be used in the transaction
    ...
    // complete the transaction
    tx.commit();
  }
  catch (Exception e) {
    // report any error as a system exception
    throw new javax.ejb.EJBException(e);
  }
}
…}
```

Bean Managed Transactions 3-3

Following are the methods of UserTransaction interface used to manage bean managed transactions:

- begin
- setTransactionTimeOut
- commit
- rollback
- setRollbackOnly
- getStatus

Bean method returning without committing:

- In case of a stateless session bean, the transaction is committed before returning the method.
- In case of stateful session bean, the association between transaction and method is retained between multiple method calls.

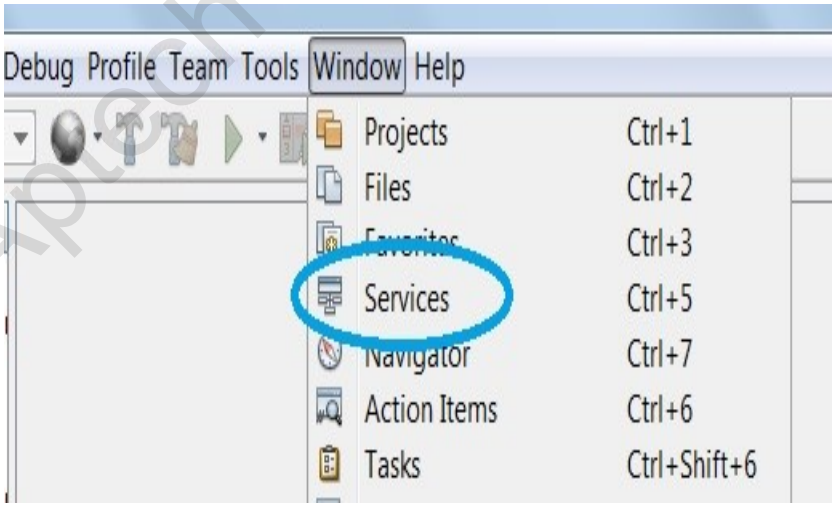
Transaction Timeouts 1-4

Transactions cannot run for indefinite amount of time as they are resource intensive.

In container managed transactions, the timeout period is set through administration console of the server.

Following are the steps to start administration console in the Netbeans IDE:

- Click Servers in the Services tab.
- If the Services tab is not visible, then select it from the Window menu as shown in the following figure:

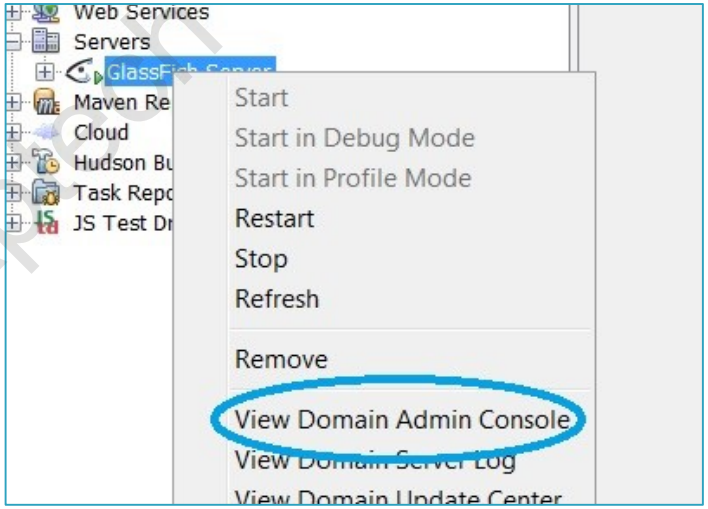


Transaction Timeouts 2-4

Expand the Servers option to view and select the currently used server, for instance, GlassFish Server.

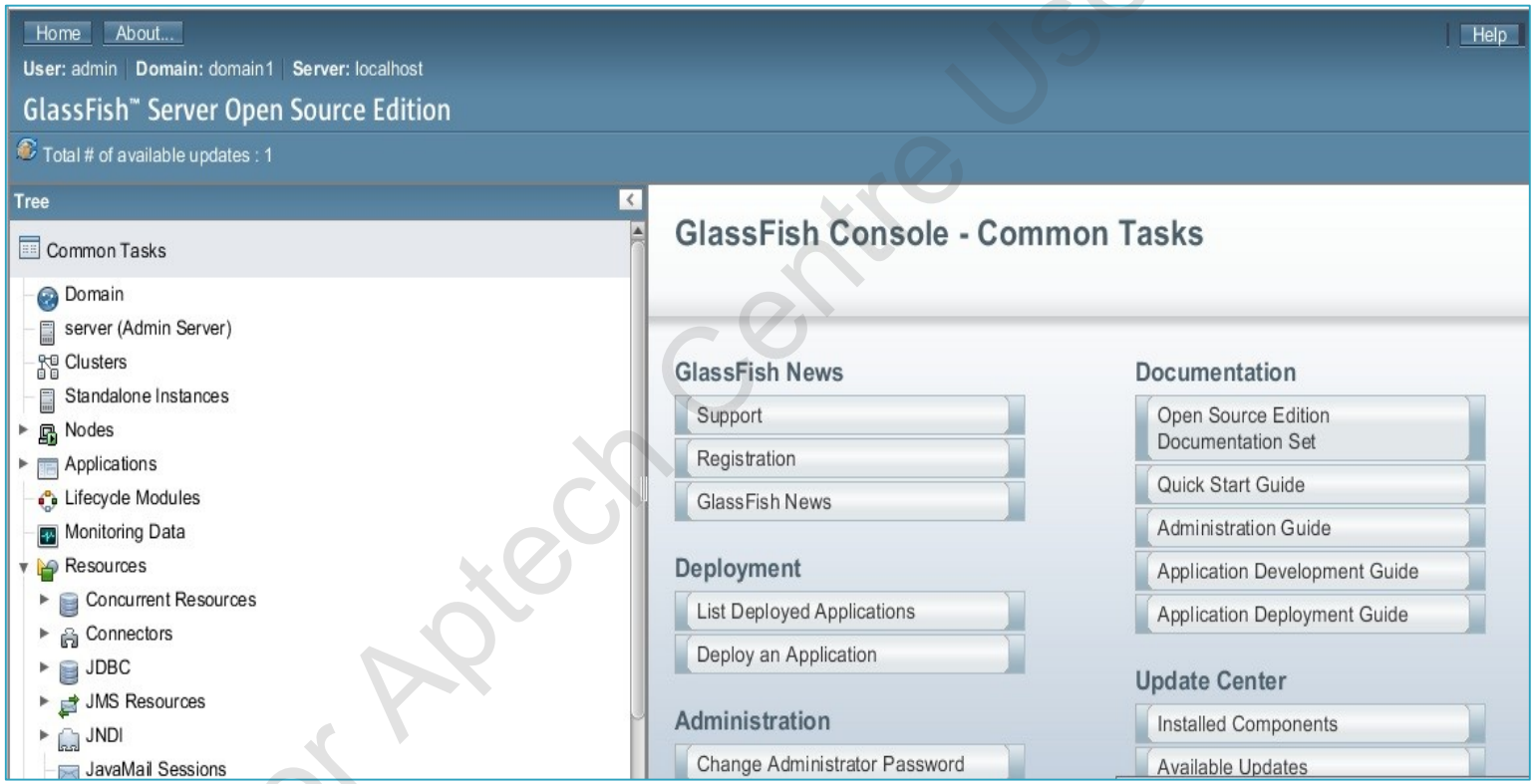
Right-click GlassFish Server to open the context menu.

Select View Domain Admin Console for the server to set the transaction timeout as shown in the following figure:



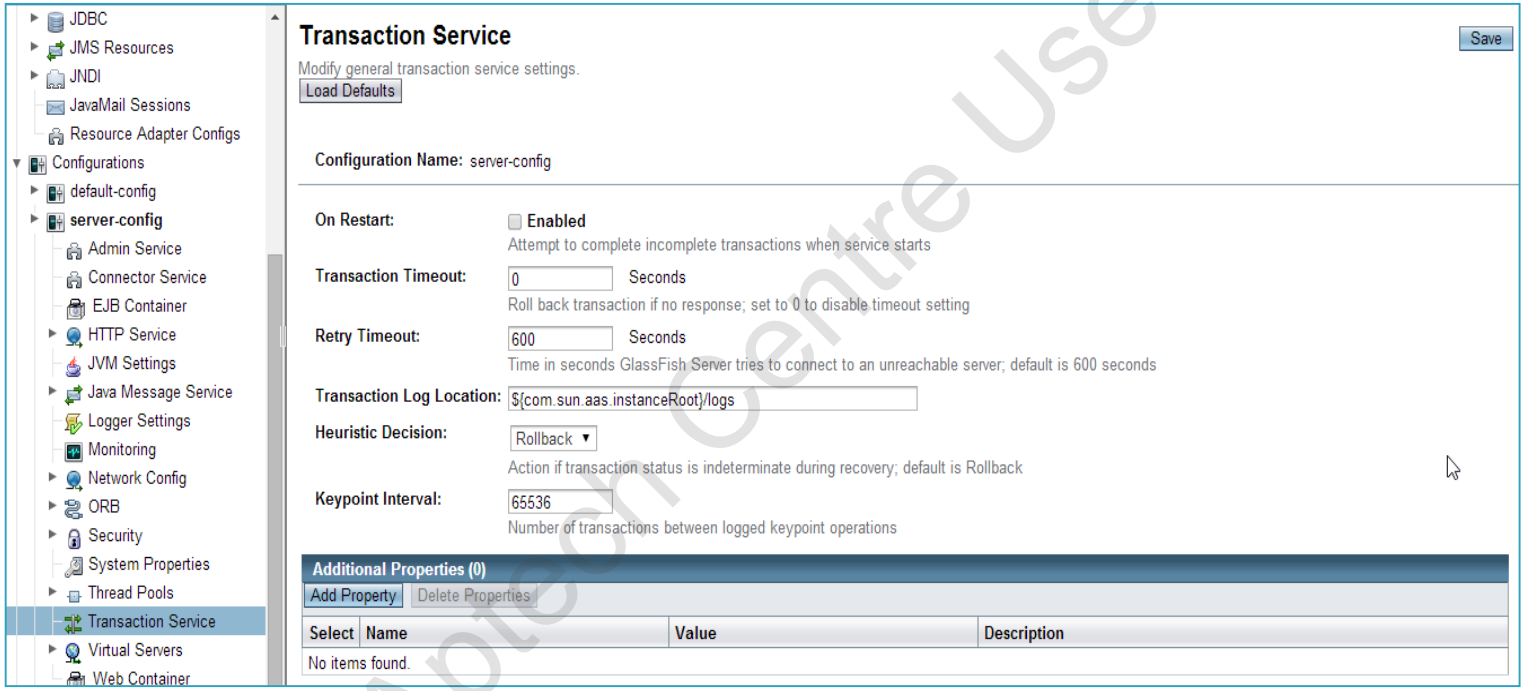
Transaction Timeouts 3-4

This opens a GlassFish server configuration page as shown in the following figure:



Transaction Timeouts 4-4

Expand Configurations in the left pane and click Transaction Service. The Transaction Service screen is displayed as shown in the following figure:



Transaction manager allows updating more than one database through bean managed transactions.

Transactions in Web Components

Web components such as Java Servlets, provide a handle to the current transaction by default through the javax.transaction.UserTransaction interface.

The transactions are initiated in the service() method of the servlets and is committed before the method exits.

The Web components can execute in a multi-threaded environment, but the transaction resource objects are not shared by threads.

The Web component may access an EJB context.

Controlling Concurrent Access 1-3

Multiple transactions access data on the database through different applications.

The transaction manager assumes that the DBMS provides locking mechanisms through read and write locks to ensure data integrity.

On the application side, the transaction manager or the persistence provider delays the writes to the database until the transaction is complete.

By default, the persistence providers for the transactions use optimistic locking.

In optimistic locking, the transaction will not acquire any lock for reading the data and to write it will check the data integrity since the transaction started.

Pessimistic locking is another mechanism where the transaction obtains locks for all the data that might be used by the transaction and holds it until the transaction is complete.

Controlling Concurrent Access 2-3

The lock mode of an entity operation may be set to one of the lock mode types listed in the following table:

Lock Mode	Description
OPTIMISTIC	This lock mode obtains optimistic locks for all the data entries used in the transaction.
OPTIMISTIC_FORCE_INCREMENT	Apart from obtaining an optimistic read lock, this lock mode also increments the version attribute of the data entity.
PESSIMISTIC_READ	Pessimistic read lock mode does not allow any other transactions to modify or delete the data. This lock obtained is retained till the transaction which acquired it is complete.
PESSIMISTIC_WRITE	When a pessimistic write lock is obtained by a transaction the data is not read, written, or updated by any other transaction. This lock is released only when the transaction is complete.
PESSIMISTIC_FORCE_INCREMENT	This lock mode obtains a long term lock on the data object and increments the version attributes of the data.
READ	A read lock implies an optimistic read lock where other transactions can also read, write, and update the data.
WRITE	A write lock implies an optimistic write lock with an increment to the version attribute. The write lock is retained until the write operation is complete.
NONE	This lock mode implies that there is no lock required on the data object.

Controlling Concurrent Access 3-3

Following are the methods which are used to set lock modes:

- lock()
- find()
- refresh()
- setLockMode()

Using Pessimistic locking

- Pessimistic locking is used when data integrity is critical for the application.
- When a transaction cannot obtain a pessimistic lock, then the transaction throws a PessimisticLockException.
- The timeout period can be defined through javax.persistence.lock.timeout property.

Demonstrating Container Managed Transactions 1-2

- Container manages transactions by defining the transaction attributes in the deployment descriptor.
- Create a Web application named TransactionApplication.
- Create a package named CMT and add a Stateless Session bean named StudentAdmissionsBean.
- Following code is added to support Container Managed Transactions:

```
package data;
import com.sun.rowset.CachedRowSetImpl;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.rowset.CachedRowSet;
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class StudentAdmissionsBean {
@TransactionAttribute(TransactionAttributeType.MANDATORY)
```

Demonstrating Container Managed Transactions 2-2

```
public String addStudent(Connection c){
        String message;
        try{
        c.createStatement();
        CachedRowSet st = new CachedRowSetImpl();
        String query = "insert into Students values('S007','Martha')";
        st.setCommand(query);
        st.execute(c);
        message="Row Inserted";
       }catch(SQLException e){
            message=e.toString(); }
        return message;
    }
  @TransactionAttribute(TransactionAttributeType.NEVER)
   public CachedRowSet display(Connection c) throws SQLException{
        Statement stmt = c.createStatement();
        CachedRowSet st = new CachedRowSetImpl();
        String quer = "select * from Students";
        st.setCommand(quer);
        st.execute(c);
        return st;
    }
}
```

Creating the Bean Client 1-7

To create a client, create a new servlet named, CMTServlet in the CMT package. Add reference to the StudentAdmissionsBean. Right-click in the editor and select Insert Code → Call Enterprise Bean → StudentAdmissionsBean from the dialog box that is displayed. Following code is added to the servlet:

```
public class CMTServlet extends HttpServlet {
    @EJB
    private StudentAdmissionsBean studentAdmissionsBean;
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      try{
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet CMTServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>STUDENT DETAILS</h1>");
            String driver = "org.apache.derby.jdbc.ClientDriver";
            String url = "jdbc:derby://localhost:1527/sample";
```

Creating the Bean Client 2-7

```
String username = "app";
            String password = "app";
            Class.forName(driver).newInstance();
            Connection conn = DriverManager.getConnection(url, username, password);
            out.println("Connection Done");
            out.println(studentAdmissionsBean.addStudent(conn));

             out.println("</body>");
              out.println("</html>");
        } catch (EJBTransactionRequiredException ex) {
            out.println(ex+ "- Error: Transaction Mandatory: Method must execute within
a transaction. ");
        }
        catch(EJBException ex){
            out.println(ex + "- Error: Method cannot execute in a transaction.");
        }
        catch(Exception ex){
             out.println("Error: Other exception - " +ex);
        }}…}
```

Creating the Bean Client 3-7

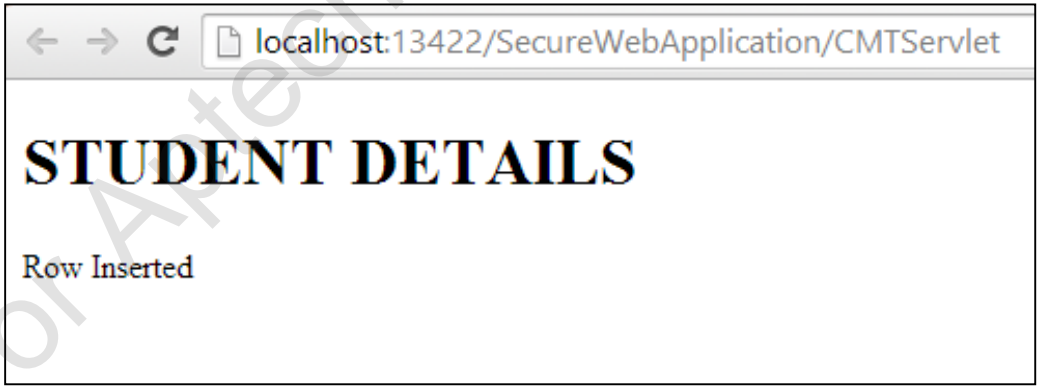
Deploy the application and run the servlet. Right-click the CMTServlet and click Run File. The output is shown in the following figure:

Creating the Bean Client 4-7

Now, add the following statements above and below the print statement, out.println(studentAdmissionsBean.addStudent(conn)); as follows:

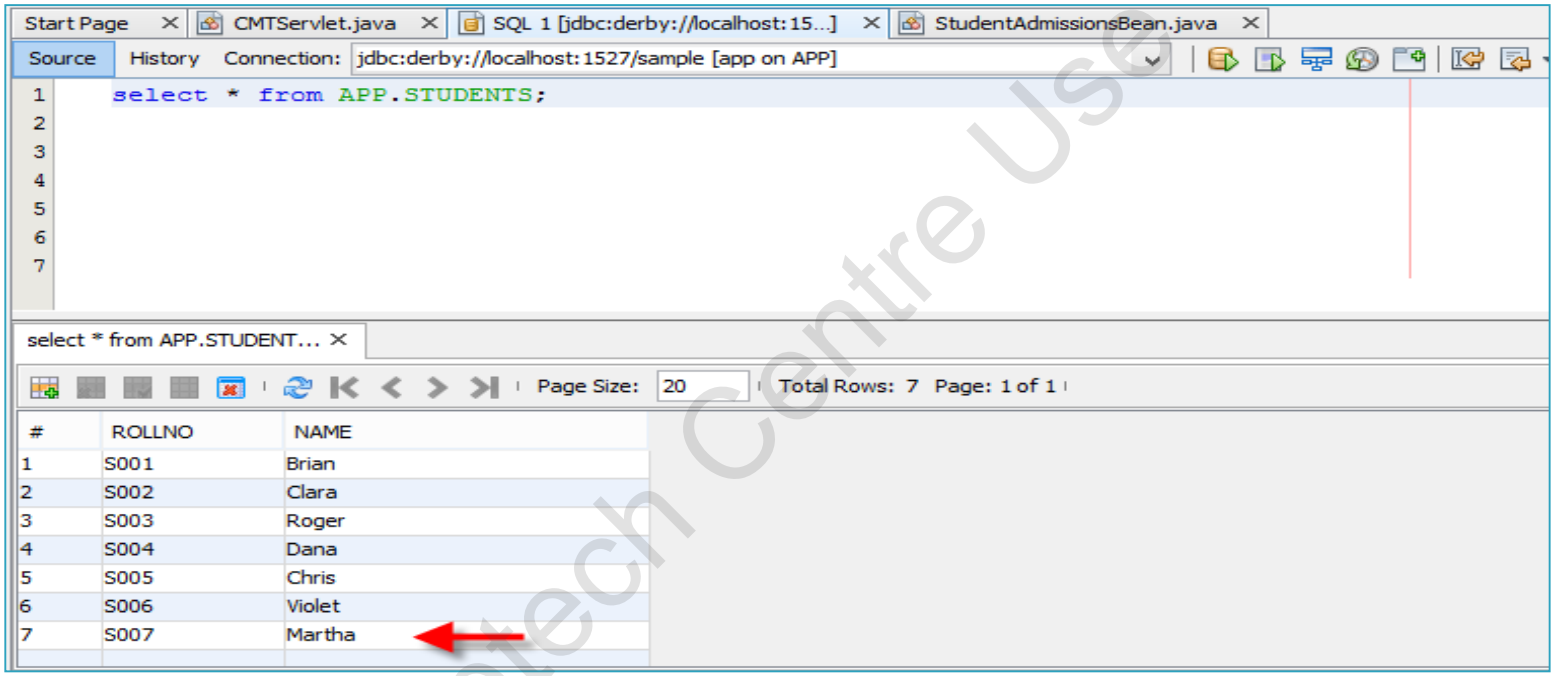
- `utx.begin();`
- `out.println(studentAdmissionsBean.addStudent(conn));`
- `utx.commit();`

Save the changes and run the servlet. The output is shown in the following figure:



Creating the Bean Client 5-7

The record inserted in the Students table is shown in the following figure:



Creating the Bean Client 6-7

Now, remove the addStudent method and invoke the display method within the transaction context as follows:

```
utx.begin();
CachedRowSet st =studentAdmissionsBean.display(conn);
while (st.next()) {
out.println("Roll No: " + st.getString(1)+ ", Name: " + st.getString(2));
}
utx.commit();
```

Save the changes and run the servlet. The output is shown in the following figure:

Creating the Bean Client 7-7

Now, remove the utx.begin() and utx.commit() statements.
Save the changes and run the servlet. The output is as shown in the following figure:

Demonstrating Bean Managed Transactions 1-4

To understand Bean Managed Transaction, create a new package BMT and add a Stateless Session bean named, StudentBean in the package. Following is the code to implement Bean Managed Transaction:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class StudentBean {
    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
    public String addStudent(Connection c){
        String message;
        try{
        c.createStatement();
        CachedRowSet st = new CachedRowSetImpl();
        String query = "insert into Students values('S008','Alice')";
        st.setCommand(query);
        st.execute(c);
        message="Row Inserted";
       }catch(SQLException e){
            message=e.toString(); }
        return message;
    }
}
```

Demonstrating Bean Managed Transactions 2-4

Create a new servlet in the BMT folder named, BMTServlet and modify the code as follows:

```
public class BMTServlet extends HttpServlet {
    @EJB
    private StudentBean studentBean;
    @Resource
    javax.transaction.UserTransaction utx;
        protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try{
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet CMTServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>STUDENT DETAILS</h1>");
```

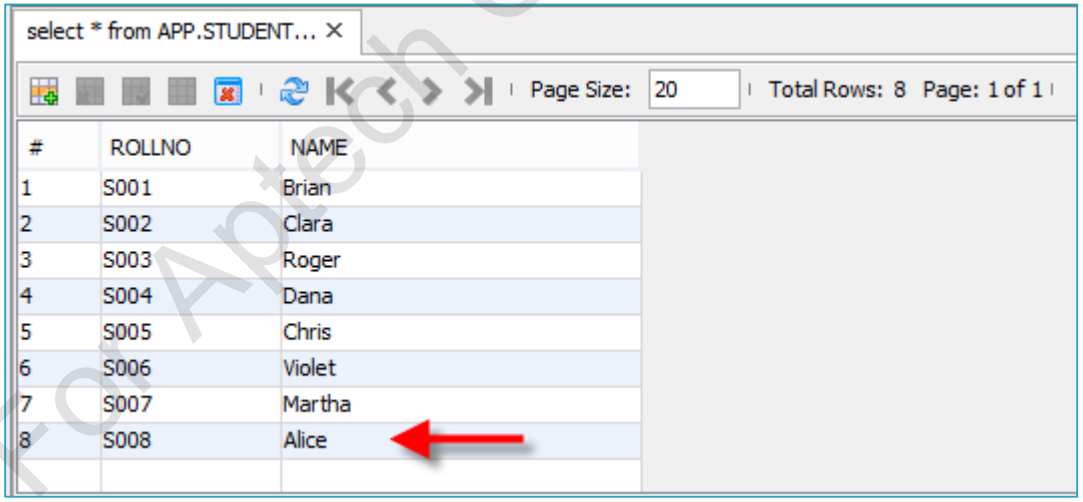
Demonstrating Bean Managed Transactions 3-4

```
String driver = "org.apache.derby.jdbc.ClientDriver";
            String url = "jdbc:derby://localhost:1527/sample";
            String username = "app";
            String password = "app";
            Class.forName(driver).newInstance();
            Connection conn = DriverManager.getConnection(url, username, password);
            System.out.println("Connection Done");
           utx.begin();

           out.println(studentBean.addStudent(conn));

           utx.commit();
             out.println("</body>");
              out.println("</html>");
        }
        catch(Exception ex){
             out.println("Error: Other exception - " +ex);
        }
    }
…
}
```

Demonstrating Bean Managed Transactions 4-4

Save the changes and run the BMTServlet file. The output is shown in the following figure:

The new row inserted in the Students table is shown in the following figure:



#	ROLLNO	NAME
1	S001	Brian
2	S002	Clara
3	S003	Roger
4	S004	Dana
5	S005	Chris
6	S006	Violet
7	S007	Martha
8	S008	Alice

Summary

- Transactions in Java EE applications are initiated by the bean methods. They can be container managed or bean managed.
- The context of container managed transactions is defined by the container. The developer cannot use transaction management methods in the bean.
- In bean managed transactions, the developer can explicitly invoke the transaction management methods.
- Transactions are associated with a timeout period. The transactions have to release the resources before the timeout period expires.
- Transactions can access multiple databases through bean methods.
- Optimistic and pessimistic locking techniques are used to ensure data integrity in the database.