Session #3  
Remote Database Connectivity

short line

# Learning Objective

To connect to a remote database server through a Socket connection and insert, update, retrieve and delete data stored in the remote database

# Learning Context

Java Database Connectivity or JDBC for short is set of Java API's that enables the developers to create platform and database independent applications in java. The biggest advantage of programming in Java is its platform independence. An application written to access the MS Access database on Win 95/Win NT platform can work on Linux against Oracle database, only by changing the name of driver, provided none of the database calls it makes are vendor specific.

The JDBC (Java Database Connectivity) API defines interfaces and classes for writing database applications in Java by making database connections. Using JDBC you can send SQL, PL/SQL statements to almost any relational database. JDBC is a Java API for executing SQL statements and supports basic SQL functionality. It provides RDBMS access by allowing you to embed SQL inside Java code. Because Java can run on a thin client, applets embedded in Web pages can contain downloadable JDBC code to enable remote database access

Although JDBC was designed specifically to provide a Java interface to relational databases, you may find that you need to write Java code to access non-relational databases as well.

JDBC Drivers are set of classes that enables the Java application to communicate with databases. Java.sql that ships with JDK contains various classes for using relational databases. But these classes do not provide any implementation, only the behaviors are defined. The actual implementations are done in third-party drivers. Third party vendors implements the java.sql.Driver interface in their database driver. A list of currently available JDBC drivers can be found at http://java.sun.com/products/jdbc/jdbc.drivers.html

**JDBC Drivers Types**

Sun has defined four JDBC driver types. They are:

1. **Type 1: JDBC-ODBC Bridge Driver -** The first type of JDBC driver is JDBC-ODBC Bridge which provide JDBC access to any ODBC complaint databases through ODBC drivers. Sun's JDBC-ODBC bridge is example of type 1 driver.
2. **Type 2: Native -API Partly - Java Driver -** Type 2 drivers are developed using native code libraries, which were originally designed for accessing the database through C/C++. Here a thin code of Java wrap around the native code and converts JDBC commands to DBMS-specific native calls.
3. **Type 3: JDBC-Net Pure Java Driver -** Type 3 drivers are a three-tier solutions. This type of driver communicates to a middleware component which in turn connects to database and provide database connectivity.
4. **Type 4: Native-Protocol Pure Java Driver -** Type 4 drivers are entirely written in Java that communicate directly with vendor's database through socket connection. Here no translation or middleware layer, are required which improves performance tremendously.

**JDBC Driver Manager**

The **JDBC DriverManager** class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

This is a very important class. Its main purpose is to provide a means of managing the different types of JDBC database driver. On running an application, it is the DriverManager's responsibility to load all the drivers found in the system property jdbc. drivers. For example, this is where the driver for the Oracle database may be defined. This is not to say that a new driver cannot be explicitly stated in a program at runtime which is not included in jdbc.drivers. When opening a connection to a database it is the DriverManager' s role to choose the most appropriate driver from the previously loaded drivers.

The JDBC API defines the Java interfaces and classes that programmers use to connect to databases and send queries. A JDBC driver implements these interfaces and classes for a particular DBMS vendor.

A Java program that uses the JDBC API loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC DriverManager class then sends all JDBC API calls to the loaded driver.

**JDBC Driver**

This topic defines the Java(TM) Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC  driver types are better suited for some applications than others.

**Types of JDBC drivers**

This topic defines the Java(TM) Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

There are  four types of JDBC drivers known as:

* JDBC-ODBC bridge plus ODBC driver, also called Type 1.
* Native-API, partly Java driver, also called Type 2.
* JDBC-Net, pure Java driver, also called Type 3.
* Native-protocol, pure Java driver, also called Type 4.

**Type 1 Driver- the JDBC-ODBC bridge**

The JDBC type 1 driver, also known as the JDBC-ODBC bridge is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls. The bridge is usually used when there is no pure-Java driver available for a particular database.

The driver is implemented in the sun.jdbc.odbc.JdbcOdbcDriver class and comes with the Java 2 SDK, Standard Edition. The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the operating system. Also, using this driver has got other dependencies such as ODBC must be installed on the computer having the driver and the database which is being connected to must support an ODBC driver. Hence the use of this driver is discouraged if the alternative of a pure-Java driver is available.

Type 1 is the simplest of all but platform specific i.e only to Microsoft platform.

A JDBC-ODBC bridge provides JDBC API access via one or more ODBC drivers. Note that some ODBC native code and in many  cases native database client code must be loaded on each client machine that uses this type of driver. Hence, this kind of  driver is generally most appropriate when automatic installation and downloading of a Java technology application is not important. For information on the JDBC-ODBC bridge driver provided by Sun, see JDBC-ODBC Bridge Driver.

Type 1 drivers are "bridge" drivers. They use another technology such as Open Database Connectivity (ODBC) to communicate  with a database. This is an advantage because ODBC drivers exist for many Relational Database Management System (RDBMS) platforms. The Java Native Interface (JNI) is used to call ODBC functions from the JDBC driver.

A Type 1 driver needs to have the bridge driver installed and configured before JDBC can be used with it. This can be a serious drawback for a production application. Type 1 drivers cannot be used in an applet since applets cannot load native code.

**Functions:**

1. Translates query obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver.
2. Sun provides a JDBC-ODBC Bridge driver. sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java, and is closed  
    source.
3. Client -> JDBC Driver -> ODBC Driver -> Database
4. There is some overhead associated with the translation work to go from JDBC to ODBC.

**Advantages:**  
Almost any database for which ODBC driver is installed, can be accessed.

**Disadvantages:**

1. Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native database connectivity interface.
2. The ODBC driver needs to be installed on the client machine.
3. Considering the client-side software needed, this might not be suitable for applets.

**Type 2 Driver - the Native-API Driver**

The JDBC type 2 driver, also known as the Native-API driver is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

The type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the final database calls.

The driver is compiled for use with the particular operating system. For platform interoperability,the Type 4 driver, being a full-Java implementation, is preferred over this driver.

A native-API partly Java technology-enabled driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.  
  
However the type 2 driver provides more functionality and performance than the type 1 driver as it does not have the overhead of the additional ODBC function calls.

Type 2 drivers use a native API to communicate with a database system. Java native methods are used to invoke the API functions that perform database operations. Type 2 drivers are generally faster than Type 1 drivers.

Type 2 drivers need native binary code installed and configured to work. A Type 2 driver also uses the JNI. You cannot use a Type 2 driver in an applet since applets cannot load native code. A Type 2 JDBC driver may require some Database Management System

grey_loader

(DBMS) networking software to be installed.

The Developer Kit for Java JDBC driver is a Type 2 JDBC driver.

**Functions:**

1. This type of driver converts JDBC calls into calls to the client API for that database.
2. Client -> JDBC Driver -> Vendor Client DB Library -> Database

**Advantage**  
Better performance than Type 1 since no jdbc to odbc translation is needed.  
  
**Disadvantages**

1. The vendor client library needs to be installed on the client machine.
2. Cannot be used in internet due the client side software needed.
3. Not all databases give the client side library.

**Type 3 driver - the Network-Protocol Driver**

The JDBC type 3 driver, also known as the network-protocol driver is a database driver implementation which makes use of a middle-tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier. However, like type 4 drivers, the type 3 driver is written entirely in Java.

The same driver can be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The type 3 driver is platform-independent as the platform-related differences are taken care by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

A net-protocol fully Java technology-enabled driver translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect all of its Java technology-based clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC API alternative. It is likely that all vendors of this solution will provide products suitable for Intranet use. In order for these products to also support Internet access they must handle the additional requirements for security, access through firewalls, etc., that the Web imposes. Several vendors are adding JDBC technology-based drivers to     their existing database middleware products.

These drivers use a networking protocol and middleware to communicate with a server. The server then translates the protocol to DBMS function calls specific to DBMS.

Type 3 JDBC drivers are the most flexible JDBC solution because they do not require any native binary code on the client. A Type 3 driver does not need any client installation.  
  
**Functions:**

1. Follows a three tier communication approach.
2. Can interface to multiple databases - Not vendor specific.
3. The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent  protocol, and then this net server translates this request into database commands for that database.
4. Thus the client driver to middleware communication is database independent.
5. Client -> JDBC Driver -> Middleware-Net Server -> Any Database

**Advantages**

1. Since the communication between client and the middleware server is database independent, there is no need for the vendor db library on the client machine. Also the client to middleware need'nt be changed for a new database.
2. The Middleware Server (Can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing etc..
3. eg. for the above include jdbc driver features in Weblogic.
4. Can be used in internet since there is no client side software needed.
5. At client side a single driver can handle any database.(It works provided the middlware supports that database!!)

**Disadvantages**

1. Requires database-specific coding to be done in the middle tier.
2. An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware    services described above.

**Type 4 - The Native-Protocol Driver**

The JDBC type 4 driver, also known as the native-protocol driver is a database driver implementation that converts JDBC calls directly into the vendor-specific database protocol.  
  
The type 4 driver is written completely in Java and is hence platform independent. It is installed inside the Java Virtual Machine of the client. It provides better performance over the type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 1 and 2 drivers, it does not need associated software to work.  
  
A native-protocol fully Java technology-enabled driver converts JDBC technology calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver. Several database vendors have these in progress.

As the database protocol is vendor-specific, separate drivers, usually vendor-supplied, need to be used to connect to the database.

A Type 4 driver uses Java to implement a DBMS vendor networking protocol. Since the protocols are usually proprietary, DBMS vendors are generally the only companies providing a Type 4 JDBC driver.

Type 4 drivers are all Java drivers. This means that there is no client installation or configuration. However, a Type 4 driver may not be suitable for some applications if the underlying protocol does not handle issues such as security and network connectivity well.  
  
The IBM Toolbox for Java JDBC driver is a Type 4 JDBC driver, indicating that the API is a pure Java networking protocol driver.

**Functions**

1. Type 4 drivers are entirely written in Java that communicate directly with a vendor's database through socket connections. No translation or middleware layers, are required, improving performance.
2. The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
3. Completely implemented in Java to achieve platform independence.
4. e.g include the widely used Oracle thin driver - oracle.jdbc.driver. OracleDriver which connect to jdbc:oracle:thin URL format.
5. Client Machine -> Native protocol JDBC Driver -> Database server

**Advantages**  
These drivers don't translate the requests into db request to ODBC or pass it to client api for the db, nor do they need a middleware layer for request indirection. Thus the performance is considerably improved.

**Disadvantage**  
At client side, a separate driver is needed for each database.

**Which Driver should be Used?**

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

**JDBC Versions**

1). The JDBC 1.0 API.

2). The JDBC 1.2 API.

3). The JDBC 2.0 Optional Package API.

4). The JDBC 2.1 core API.

5) The JDBC 3.0 API.

6) The JDBC 4.0 API.

**Features of JDBC 1.0 API**

The JDBC 1.0 API was the first officially JDBC API launched consists of the following java classes and interfaces that you can open connections to particular databases.

This version includes a completely redesigned administration console with an enhanced graphical interface to manage and monitor distributed virtual databases.

**Features of JDBC 1.2 API**

1). It supports Updatabale ResultSets.

2). The DatabaseMetaData code has been refactored to provide more transparency with regard to the underlying database engine.

3) New pass through schedulers for increased performance.

**Features of The JDBC 2.0 Optional Pacakage API**

1). The use of DataSource interface for making a connection.

2). Use of JNDI to specify and obtain database connections.

3). It allows us to use Pooled connections, that is we can reuse the connections.

4). In this version the distrbuted transactions is possible.

5). It provides a way of handling and passing data using Rowset technology.

**Features of the JDBC 2.1 core API.**

1). Scroll forward and backward in a result set or has the ability to move to a specific row.

2). Instead of using SQL commands, we can make updates to a database tables using methods in the Java programming language

3). We can use multiple SQL statements in a a database as a unit, or batch.

4). It uses the SQL3 datatypes as column values. SQL3 types are Blob, Clob, Array, Structured type, Ref.

5). Increased support for storing persistent objects in the java programming language.

6). Supports for time zones in Date, Time, and Timestamp values.

7). Full precision for java.math.BigDecimal values.

**Features of JDBC 3.0 API**

1). Reusabilty of prepared statements by connection pools.

2). In this version there is number of properties defined for the ConnectionPoolDataSource. These properties can be used to describe how the PooledConnection objects created by DataSource objects should be pooled.

3) A new concept has been added to this API is of savepoints.

4). Retrieval of parameter metadata.

5). It has added a means of retrieving values from columns containing automatically generated values.

6). Added a new data type i.e. java.sql.BOOLEAN.

7). Passing parameters to CallableStatement.

8). The data in the Blob and Clob can be altered.

9). DatabaseMetaData API has been added.

**Features of JDBC 4.0 :**

1). Auto- loading of JDBC driver class.

2). Connection management enhancements.

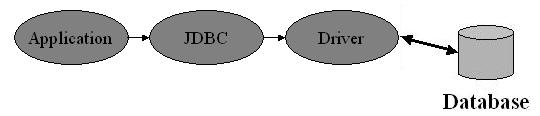
3.) Support for RowId SAL type.

4). SQL exception handling enhancements.

5). DataSet implementation of SQl using Annotations.

6). SQL XML support

**JDBC Architecture**



Java application calls the JDBC library. JDBC loads a driver which talks to the database. We can change database engines without changing database code.

**JDBC Basics - Java Database Connectivity Steps**

Before you can create a java jdbc connection to the database, you must first import the  
java.sql package.

import java.sql.\*; The star ( \* ) indicates that all of the classes in the package java.sql are to be imported.

**1. Loading a database driver**

In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used. The return type of the Class.forName (String ClassName) method is “Class”. Class is a class in java.lang package.

|  |
| --- |
| try {  Class.forName(”sun.jdbc.odbc.JdbcOdbcDriver”); // Type 1 (Bridge) driver  Class.forName (“ oracle.jdbc.driver.OracleDriver”);// Type 4 driver (oracle)  }  catch(Exception x){  System.out.println( “Unable to load the driver class!” );  } |

2. **Creating a oracle jdbc Connection**

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system. Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases. A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

|  |
| --- |
| JDBC URL Syntax:: jdbc: <subprotocol>: <subname> |

JDBC URL Example:: jdbc: <subprotocol>: <subname>•Each driver has its own subprotocol. Each subprotocol has its own syntax for the source. We’re using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

For Type 1 driver url = “sun.jdbc.odbc.JdbcOdbcDriver”;

For Type 4 driver url =” jdbc:oracle:thin:@172.16.4.100:1521:ora9”;

|  |
| --- |
| try{  Connection dbConnection=DriverManager.getConnection(url,”loginName”,”Password”)  }  catch( SQLException x ){  System.out.println( “Couldn’t get connection!” );  } |

**3. Creating a jdbc Statement object**

Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.

Statement statement = dbConnection.createStatement();

A statement object is used to send and execute SQL statements to a database.

Three kinds of Statements

**Statement:** Execute simple sql queries without parameters.Statement createStatement() Creates an SQL Statement object.

**Prepared Statement:** Execute precompiled sql queries with or without parameters.PreparedStatement prepareStatement(String sql)returns a new PreparedStatement object. PreparedStatement objects are precompiled SQL statements.

**Callable Statement:** Execute a call to a database stored procedure.CallableStatement prepareCall(String sql)returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

**4. Executing a SQL statement with the Statement object, and returning a jdbc resultSet.**

Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements:  
executeQuery(), executeUpdate(), and execute(). For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate. Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. execute() executes an SQL statement that is written as String object.

**ResultSet** provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

**ResultSetMetaData** Interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

Review