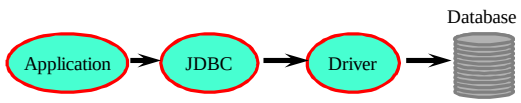
JDBC - Java Database Connectivity

What is JDBC?

- "An API that lets you access virtually any tabular data source from the Java programming language"
- It allows to access virtually any data source, from relational databases to spreadsheets and flat files.

JDBC Architecture



- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- Can have more than one driver -> more than one database
- Ideal: can change database engines without changing any application code

JDBC Components

1. The JDBC API

- It provides programmatic access to relational data from the Java programming language.
- Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source.
- The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

JDBC Components

2. 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.

3. JDBC Test Suite

- The JDBC driver test suite helps you to determine that JDBC drivers will run your program.
- These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

JDBC Components

4. JDBC-ODBC Bridge

- The Java Software bridge provides JDBC access via ODBC drivers.
- Note that you need to load ODBC binary code onto each client machine that uses this driver.
- As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

JDBC Drivers

Type 1 : JDBC-ODBC bridge plus ODBC driver,

Type 2 : Native-API, partly Java driver,

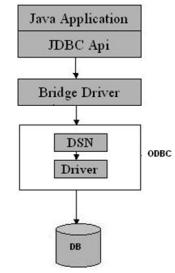
Type 3 : JDBC Network Driver, partially java

Type 4 : Native-protocol, pure Java driver

Type 1 Driver

Functions:

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



Type 1: JDBC-ODBC Bridge

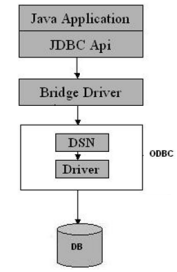
Type 1 Driver

Advantages:

- Almost any database, for which ODBC driver is installed, can be accessed.

Disadvantages:

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



Type 1: JDBC-ODBC Bridge

Type 2 Driver

Functions:

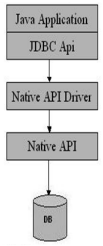
- This type of driver converts JDBC calls into calls to the client API for that database.
- Client -> JDBC Driver -> Vendor Client DB Library -> Database

Advantage

- Better performance than Type 1 since no Jdbc to Odbc translation is needed.

Disadvantages

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

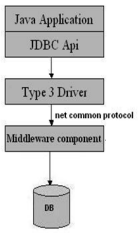


Type 2: Native api/ Partly Java Driver

Type 3 Driver

Functions:

- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- 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.
- Thus the client driver to middleware communication is database independent.
- Client -> JDBC Driver -> Middleware-Net Server -> Any Database

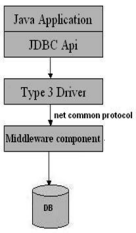


Type 3: All Java/ Net-Protocol Driver

Type 3 Driver

Advantages:

- 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 needn't be changed for a new database.
- 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..
- Eg. The above jdbc driver features in Weblogic.
- Can be used in internet since there is no client side software needed.
- At client side a single driver can handle any database. (It works provided the middleware supports that database!!)

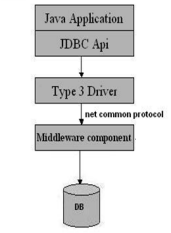


Type 3: All Java/ Net-Protocol Driver

Type 3 Driver

Disadvantages:

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

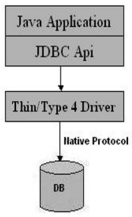


Type 3: All Java/ Net-Protocol Driver

Type 4 Driver

Functions

- Type 4 drivers are entirely written in Java that communicates directly with a vendor's database through socket connections. No translation or middleware layers are required, improving performance.
- The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
- Completely implemented in Java to achieve platform independence.
- E.g MySQL Connector/J
- Client Machine -> Native protocol JDBC Driver -> Database server



Type 4: Native-protocol/all-Java driver

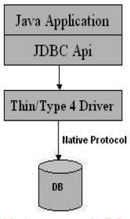
Type 4 Driver

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.

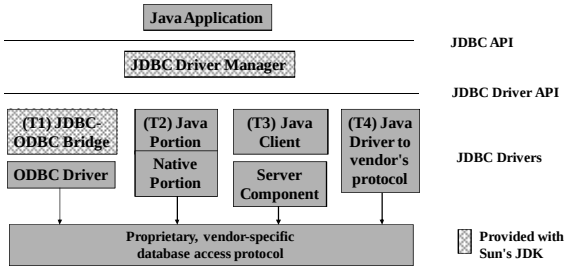
Disadvantage:

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



Type 4: Native-protocol/all-Java driver

JDBC Architecture



JDBC Classes

- DriverManager
  - Manages JDBC Drivers
  - Used to Obtain a connection to a Database
- Types
  - Defines constants which identify SQL types
- Date
  - Used to Map between java.util.Date and the SQL DATE type
- Time
  - Used to Map between java.util.Date and the SQL TIME type
- TimeStamp
  - Used to Map between java.util.Date and the SQL TIMESTAMP type

JDBC Interfaces

- Driver
  - All JDBC Drivers must implement the Driver interface. Used to obtain a connection to a specific database type
- Connection
  - Represents a connection to a specific database
  - Used for creating statements
  - Used for managing database transactions
  - Used for accessing stored procedures
  - Used for creating callable statements
- Statement
  - Used for executing SQL statements against the database

JDBC Interfaces

- ResultSet
  - Represents the result of an SQL statement
  - Provides methods for navigating through the resulting data
- PreparedStatement
  - Similar to a stored procedure
  - An SQL statement (which can contain parameters) is compiled and stored in the database
- CallableStatement
  - Used for executing stored procedures
- DatabaseMetaData
  - Provides access to a database's system catalogue
- ResultSetMetaData
  - Provides information about the data contained within a ResultSet

Using JDBC

- To execute a statement against a database, the following flow is observed
  - Load the driver (Only performed once)
  - Obtain a Connection to the database (Save for later use)
  - Obtain a Statement object from the Connection
  - Use the Statement object to execute SQL.
    - Update, insert and delete return Boolean.
    - Select returns a ResultSet
  - Navigate ResultSet, using data as required
  - Close ResultSet
  - Close Statement
- Do NOT close the connection
  - The same connection object can be used to create further statements
  - A Connection may only have one active Statement at a time. Do not forget to close the statement when it is no longer needed.
  - Close the connection when you no longer need to access the database

Loading Drivers

- Even a good API can have problems
  - Loading drivers fits into this category
- The DriverManager is a singleton
- Each JDBC Driver is also a singleton
- When a JDBC Driver class is loaded, it must create an instance of itself and register that instance with the JDBC DriverManager
- How does one load a "class" into the Virtual machine?
  - Use the static method Class.forName()
  - Example : Class.forName("com.mysql.jdbc.Driver");

Connecting to a Database

- Once a Driver is loaded, a connection can be made to the database
- A connection is obtained in the following manner:

```
Connection aConnection =
        DriverManager.getConnection("jdbc:mysql://localhost/MyDatabase");
```

- Overloaded versions of the getConnection method allow the specification of a username and password for authentication with the database.

Using a Connection

- The Connection interface defines many methods for managing and using a connection to the database

```
public Statement createStatement()
public PreparedStatement prepareStatement(String sql)
public void setAutoCommit(boolean)
public void commit()
public void rollback()
public void close()
```

- The most commonly used method is createStatement()

  When an SQL statement is to be issued against the database, a Statement object must be created through the Connection

Using a Statement

- The Statement interface defines two methods for executing SQL against the database
  - public ResultSet executeQuery(String sql)
  - public int executeUpdate(String sql)
- executeQuery returns a ResultSet
  - All rows and columns which match the query are contained within the ResultSet
  - The developer navigates through the ResultSet and uses the data as required.
- executeUpdate returns the number of rows changed by the update statement
  - This is used for insert statements, update statements and delete statements

Using a ResultSet

- The ResultSet interface defines many navigation methods

  public boolean first()
  public boolean last()
  public boolean next()
  public boolean previous()

- The ResultSet interface also defines data access methods

  public int getInt(int columnNumber) -- Note: Columns are numbered
  public int getInt(String columnName) -- from 1 (not 0)
  public long getLong(int columnNumber)
  public long getLong(String columnName)
  public String getString(int columnNumber)
  public String getString(String columnName)

- There are MANY more methods. Check the API documentation for a complete list

SQL Types/Java Types Mapping

SQL Type	*Java Type*
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.Math.BigDecimal
DECIMAL	java.Math.BigDecimal
BIT	boolean
TINYINT	int
SMALLINT	int
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Example Code:

```
Connection aConnection;

try
{
  Class.forName("com.mysql.jdbc.Driver").newInstance();
}
catch(ClassNotFoundException x)
{
  System.out.println("Cannot find driver class.  Check CLASSPATH");
  return;
}

try
{
  aConnection =
  DriverManager.getConnection("jdbc:mysql://localhost/MyDatabase",

          "Username", "Password");
}
catch(SQLException x)
{
  System.out.println("Exception connecting to database:" + x);
  return;
}
```

Example Code (continued):

```
try
{
  Statement aStmt = aConnection.createStatement();
  String query = "SELECT Employee_id, Employee_Name FROM Employee WHERE
  EmployeeId>100";

  ResultSet rs = aStmt.executeQuery(query);

  while(rs.next())
  {
          int employeeId = rs.getInt(1);
          String employeeName = rs.getString(2);

          System.out.println("Id:" + employeeId + "\nName:" + employeeName);
  }

  rs.close();
  aStmt.close();
}
catch(SQLException x)
{
  System.out.println("Exception while executing query:" + x);
}
```