Servlet Cookies & Session

Unit I

Introduction to Cookies

- Cookies are small bits of textual information that a Web server sends to a browser and that the browser returns unchanged when later visiting the same Web site or domain.
- By letting the server read information it sent the client previously, the site can provide visitors with a number of conveniences such as presenting the site the way the visitor previously customized it or letting identifiable visitors in without their having to enter a password.
- Most browsers avoid caching documents associated with cookies, so the site can return different content each time

Benefits of Cookies

- Identifying a User During an E-commerce Session
- Avoiding Username and Password(for low-security sites)
- Customizing a Site
- Focusing Advertising
- Cookies do not require any server resources since they are stored on the client.
- Cookies are easy to implement.
- You can configure cookies to expire when the browser session ends (session cookies) or they can exist for a specified length of time on the client computer (persistent cookies).

Some Problems with Cookies

- Can present a significant threat to privacy: some people don’t like the fact that search engines can remember that they’re the user who usually does searches on certain topics.
- A second privacy problem occurs when sites rely on cookies for overly sensitive data
- Sometimes clients disable cookies on their browsers in response to security or privacy worries which will cause problem for web applications that require them.

- Individual cookie can contain a very limited amount of information (not more than 4 kb).
- Cookies are limited to simple string information. They cannot store complex information.
- Cookies are easily accessible and readable if the user finds and reopens.
- Most browsers restrict the number of cookies that can be set by a single domain to not more than20 cookies (except Internet Explorer). If you attempt to set more than 20 cookies, the oldest cookies are automatically deleted.

Cookies Handling

- Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.
- There are three steps involved in identifying returning users:
  - Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
  - Browser stores this information on local machine for future use.
  - When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

- Cookies are usually set in an HTTP header
- Servlet Cookies Methods:
- **public void setMaxAge(int expiry)**
  - This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
- **public int getMaxAge()**
  - This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
- **public String getName()**
  - This method returns the name of the cookie. The name cannot be changed after creation.

- **public void setValue(String newValue)**
  - This method sets the value associated with the cookie.
- **public String getValue()**
  - This method gets the value associated with the cookie.
- **public void setComment(String purpose)**
  - This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
- **public String getComment()**
  - This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

- **public String getDomain()**
- **public void setDomain(String domainPattern)**
- These methods get or set the domain to which the cookie applies. The browser only returns cookies to the exact same hostname that sent them. You can use setDomain method to instruct the browser to return them to other hosts within the same domain.
- **public String getPath()**
- **public void setPath(String path)**
- These methods get or set the path to which the cookie applies.
- someCookie.setPath("/") specifies that all pages on the server should receive the cookie. The path specified must include the current page; that is, you may specify a more general path than the default, but not a more specific one.

- **public boolean getSecure()**
- **public void setSecure(boolean secureFlag)**
- This pair of methods gets or sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e., SSL) connections. The default is false; the cookie should apply to all connections.

- Setting Cookies with Servlet:

(1) Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

- Cookie cookie = new Cookie("key","value");
- Keep in mind, neither the name nor the value should contain white space or any of the following characters:
- [ ] ( ) = , " / ? @ : ;

(2) Setting the maximum age: You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

- cookie.setMaxAge(60*60*24);
- **(3) Sending the Cookie into the HTTP response headers:** You use**response.addCookie** to add cookies in the HTTP response header as follows:
- response.addCookie(cookie);

- **Delete Cookies with Servlet:**
- To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps:
  - Read an already exsiting cookie and store it in Cookie object.
  - Set cookie age as zero using **setMaxAge()** method to delete an existing cookie.
  - Add this cookie back into response header.

```
for (int i = 0; i < cookies.length; i++){
    cookie = cookies[i];
        if((cookie.getName( )).compareTo("first_name") == 0 ) {
                cookie.setMaxAge(0);
                response.addCookie(cookie);
                out.print("Deleted cookie : " + cookie.getName());
}
```

Servlets - Session Tracking

- **Need for Session Tracking:**
  - HTTP is a “stateless” protocol: each time a client retrieves a Web page, it opens a separate connection to the Web server, and the server does not automatically maintain contextual information about a client.
  - Even with servers that support persistent (keep-alive) HTTP connections and keep a socket open for multiple client requests that occur close together in time, there is no built-in support for maintaining contextual information. This lack of context causes a number of difficulties.
- There are three typical solutions to this problem: cookies, URL-rewriting, and hidden form fields.

- **Hidden Form Fields:**
- A web server can send a hidden HTML form field along with a unique session ID as follows:
- <input type="hidden" name="sessionid" value="12345">
- Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.
- This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

- **URL Rewriting:**
- You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.
- For example, with http://abc.com/file.htm;sessionid=12345, the session identifier is attached as sessionid=12345 which can be accessed at the web server to identify the client.
- URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.
- If the user leaves the session and comes back via a bookmark or link, the session information can be lost

HttpSession

- **The HttpSession Object:**
- Servlet provides HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
- The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.
- You would get HttpSession object by calling the public method **getSession()** of HttpServletRequest, as below:
- HttpSession session = request.getSession();
- You need to call *request.getSession()* before you send any document content to the client.

- The important methods available through HttpSession object:
- public Object getAttribute(String name)
- public Enumeration getAttributeNames()
- public long getCreationTime()
- public String getId()
- public long getLastAccessedTime()
- public int getMaxInactiveInterval()
- public void invalidate()
- public boolean isNew()
- public void removeAttribute(String name)
- public void setAttribute(String name, Object value)
- public void setMaxInactiveInterval(int interval)

Deleting Session Data

- **Remove a particular attribute:** You can call *public void removeAttribute(String name)* method to delete the value associated with a particular key.
- **Delete the whole session:** You can call *public void invalidate()*method to discard an entire session.
- **Setting Session timeout:** You can call *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.
- **Log the user out:** The servers that support servlets 2.4, you can call**logout** to log the client out of the Web server and invalidate all sessions belonging to all the users.
- **web.xml Configuration:** If you are using Tomcat, apart from the above mentioned methods, you can configure session time out in web.xml file as follows.

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

JDBC

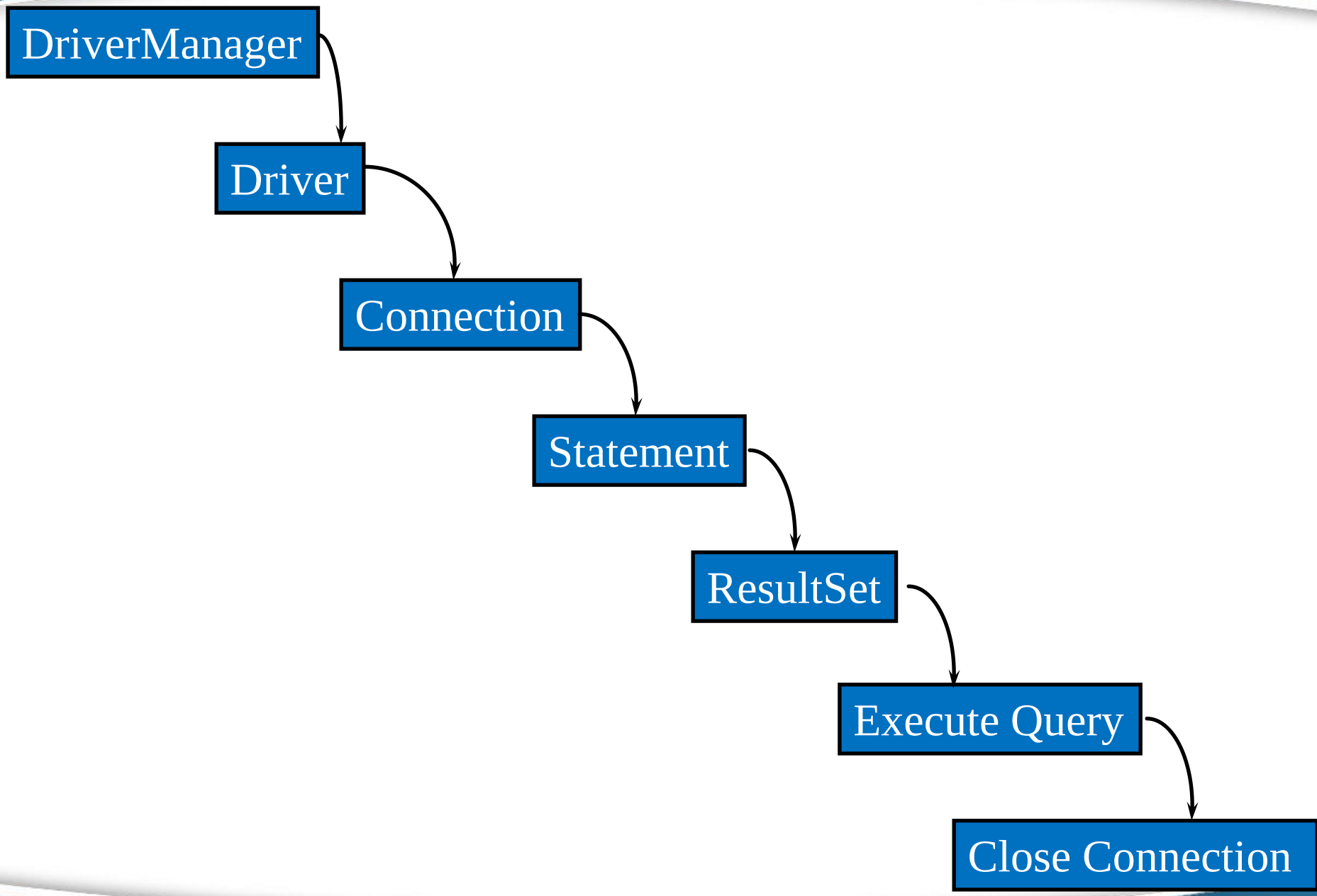
Introduction

- JDBC provides a standard library for accessing relational databases. Using the JDBC API, you can access a wide variety of different SQL databases with exactly the same Java syntax.
- It is important to note that although JDBC standardizes the mechanism for connecting to databases, the syntax for sending queries and committing transactions, and the data structure representing the result, it does *not attempt to standardize* the SQL syntax.
- So, you can use any SQL extensions your database vendor supports. However, since most queries follow standard SQL syntax, using JDBC lets you change database hosts, ports, and even database vendors with minimal changes in your code.

- Officially, JDBC is not an acronym and thus does not stand for anything. Unofficially, “Java Database Connectivity” is commonly used.

Basic Steps in Using JDBC

- There are seven standard steps in querying databases:
- 1. Load the JDBC driver.
- 2. Define the connection URL.
- 3. Establish the connection.
- 4. Create a statement object.
- 5. Execute a query or update.
- 6. Process the results.
- 7. Close the connection.
- Here are some details of the process.


DriverManager
Driver
Connection
Statement
ResultSet
Execute Query
Close Connection

1. ***Load the Driver***

- The driver is the piece of software that knows how to talk to the actual database server.
- To load the driver, all you need to do is to load the appropriate class; a static block in the class itself automatically makes a driver instance and registers it with the JDBC driver manager.
- Use Class.forName. This method takes a string represent ing a fully qualified class name (i.e., one that includes package names) and loads the corresponding class.
- This call could throw a ClassNotFound- Exception, so should be inside a try/catch block.

```
try {
    Class.forName("connect.microsoft.MicrosoftDriver");
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Class.forName("com.sybase.jdbc.SybDriver");
} catch(ClassNotFoundException cnfe) {
    System.err.println("Error loading driver: " + cnfe);
}
```

- For an up-to-date list of JDBC drivers, see http://java.sun.com/products/jdbc/drivers.html
- Most JDBC driver vendors distribute their drivers inside JAR files. So, be sure to include the path to the JAR file in your CLASSPATH setting.

2. Define the Connection URL

- Once you have loaded the JDBC driver, you need to specify the location of the database server. URLs referring to databases use the jdbc: protocol and have the server host, port, and database name (or reference) embedded within the URL.

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +":" + port + ":" +
    dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host +":" + port + ":" +
    "?SERVICENAME=" + dbName;
```

3. ***Establish the Connection***

- To make the actual network connection, pass the URL, the database username, and the password to the getConnection method of the Driver-Manager class, as illustrated in the following example.
- Note that getConnection throws an SQLException, so you need to use a try/catch block.

```
String username = "jay_debesee";
String password = "secret";
Connection connection =
DriverManager.getConnection(oracleURL, username, password);
```

- An optional part of this step is to look up information about the database by using the getMetaData method of Connection.
- This method returns a DatabaseMetaData object which has methods to let you discover the name and version of the database itself (getDatabaseProductName, getDatabaseProductVersion)
- or of the JDBC driver (getDriverName, get-DriverVersion).

```
DatabaseMetaData dbMetaData=connection.getMetaData();
String productName=dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion=dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```

- Other useful methods in the Connection class include prepareStatement, prepare-call, rollback, commit, close, and isClosed.

4. Create a Statement

- A Statement object is used to send queries and commands to the database and is created from the Connection as follows:
- Statement statement = connection.createStatement();

5. Execute a Query

- Once you have a Statement object, you can use it to send SQL queries by using the executeQuery method, which returns an object of type Result-Set. Here is an example:

String query = "SELECT col1, col2, col3 FROM sometable";

ResultSet resultSet = statement.executeQuery(query);

- To modify the database, use executeUpdate instead of executeQuery, and supply a string that uses UPDATE, INSERT, or DELETE.
- Other useful methods in the Statement class include execute and setQueryTimeout.
- You can also create parameterized queries where values are supplied to a precompiled fixed-format query.

6. ***Process the Results***

- The simplest way to handle the results is to process them one row at a time, using the ResultSet's next method to move through the table a row at a time.
- Within a row, ResultSet provides various get*Xxx methods that take a* column index or column name as an argument and return the result as a variety of different Java types.
- For instance, use getInt if the value should be an integer, getString for a String, and so on.
- *The first column in a ResultSet row has index 1, not 0.*

```
while(resultSet.next()) {
System.out.println(results.getString(1) + " " +
results.getString(2) + " " +
results.getString(3));
}
```

- other useful methods in the ResultSet class include findColumn (get the index of the named column), wasNull (was the last get*Xxx result SQL NULL?*
- *Alternatively, for strings you* can simply compare the return value to null), and getMetaData (retrieve information about the ResultSet in a ResultSetMetaData object).
- Useful ResultSetMetaData methods include getColumnCount (the number of columns), getColumn-Name(colNumber) (a column name, indexed starting at 1), getColumnType (an int to compare against entries in java.sql.Types), isReadOnly (is entry a read-only value?), isSearchable (can it be used in a WHERE clause?), isNullable (is a null value permitted?), and several others that give details on the type and precision of the column.

7. *Close the Connection*

- To close the connection explicitly, you would do:
- connection.close();

Basic Example

```
import java.sql.*;
class JDBCDemoOr{
public static void main(String args[]){
try{
System.out.println("test");    //step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");    //step2 create the con obj
Connection
    con=DriverManager.getConnection("jdbc:oracle:thin:@192.168.9.251:152
    1:xe","kirtan","kirtan");
System.out.println(con);  //step3 create the statement object
Statement stmt=con.createStatement();    //step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
  //step5 close the connection object
con.close();
  }catch(Exception e){ System.out.println(e);}
  }
```

JDBC - Statements, PreparedStatement and CallableStatement

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

- The Statement Objects
  - Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method.
  - Statement stmt = conn.createStatement( );
  - Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.
  - **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

- The PreparedStatement Objects
  - This statement gives you the flexibility of suppl ying arguments dynamically.

    ```
    String SQL = "Update Employees SET age = ? WHERE id = ?";
    PreparedStatement pstmt = conn.prepareStatement(SQL);
    ```

  - All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.
  - The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

- Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth.
- All of the Statement object's methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object.

- The CallableStatement Objects:
  - This would be used to execute a call to a database stored procedure.
  - Suppose, you need to execute the following Oracle stored procedure.

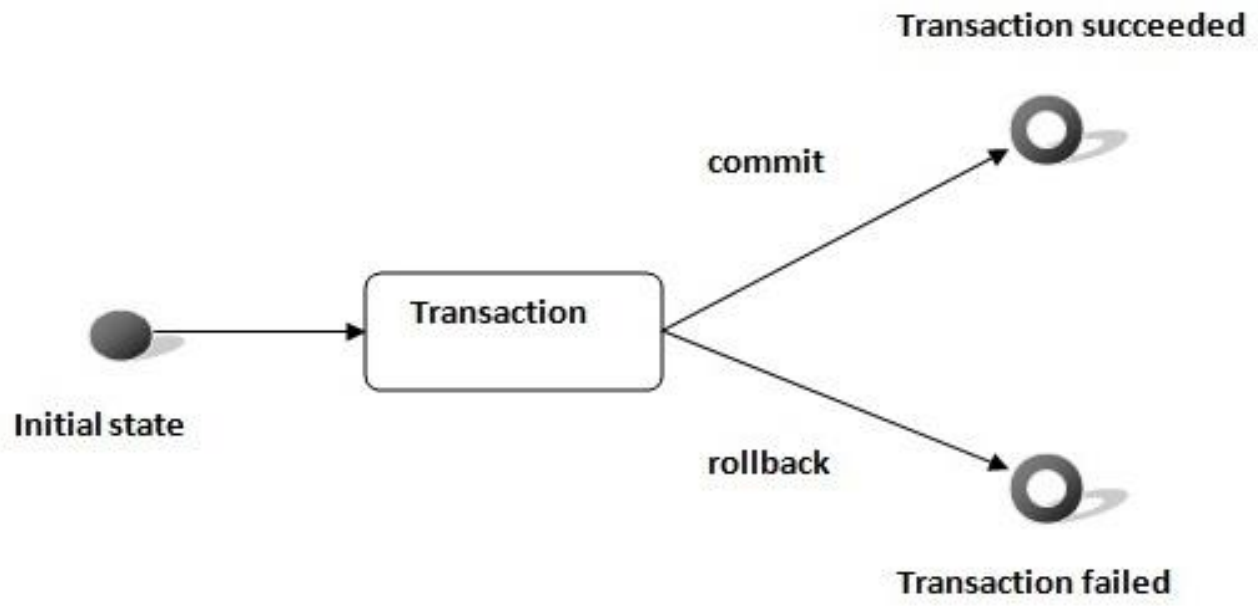
```
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END;
```

```
    String SQL = "{call getEmpName (?, ?)}";
    CallableStatement cstmt = conn.prepareCall (SQL);
//Bind IN parameter first, then bind OUT parameter
    int empID = 102;
    stmt.setInt(1, empID); // This would set ID as 102
// Because second parameter is OUT so register it
    stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
//Use execute method to run stored procedure.
    System.out.println("Executing stored procedure..." );
    stmt.execute();
```

Transaction Management in JDBC

- Transaction represents **a single unit of work**.
- The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.
  - **Atomicity** means either all successful or none.
  - **Consistency** ensures bringing the database from one consistent state to another consistent state.
  - **Isolation** ensures that transaction is isolated from other transaction.
  - **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

- Advantage of Transaction Mangaement:
- **fast performance** It makes the performance fast because database is hit at the time of commit.



- In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit (boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database.There are 4 types of JDBC drivers:
  - JDBC-ODBC bridge driver
  - Native-API driver (partially java driver)
  - Network Protocol driver (fully java driver)
  - Thin driver (fully java driver)

- JDBC-ODBC bridge driver
  - The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.
  - Advantages:
    - easy to use.
    - can be easily connected to any database.
  - Disadvantages:
    - Performance degraded because JDBC method call is converted into the ODBC function calls.
    - The ODBC driver needs to be installed on the client machine.

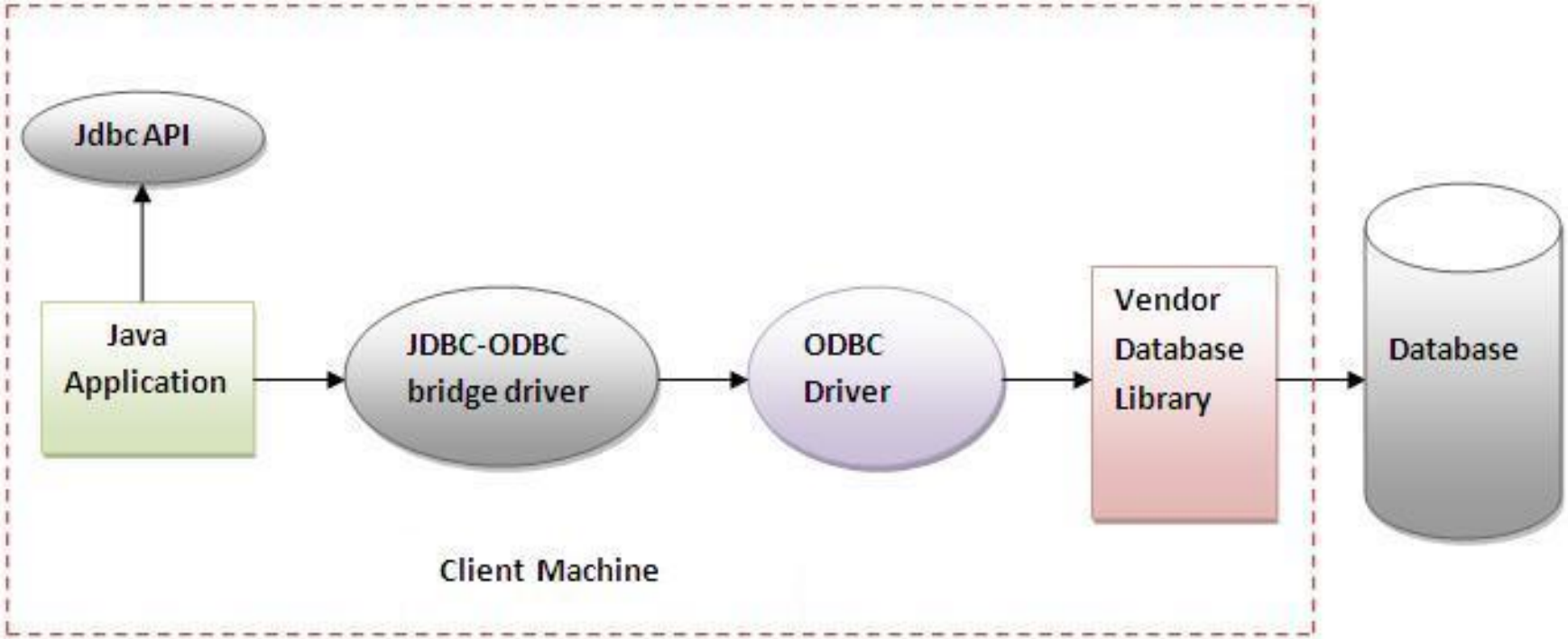


Figure- JDBC-ODBC Bridge Driver

- Native-API driver
  - The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.
  - Advantage:
    - performance upgraded than JDBC-ODBC bridge driver.
  - Disadvantage:
    - The Native driver needs to be installed on the each client machine.
    - The Vendor client library needs to be installed on client machine.

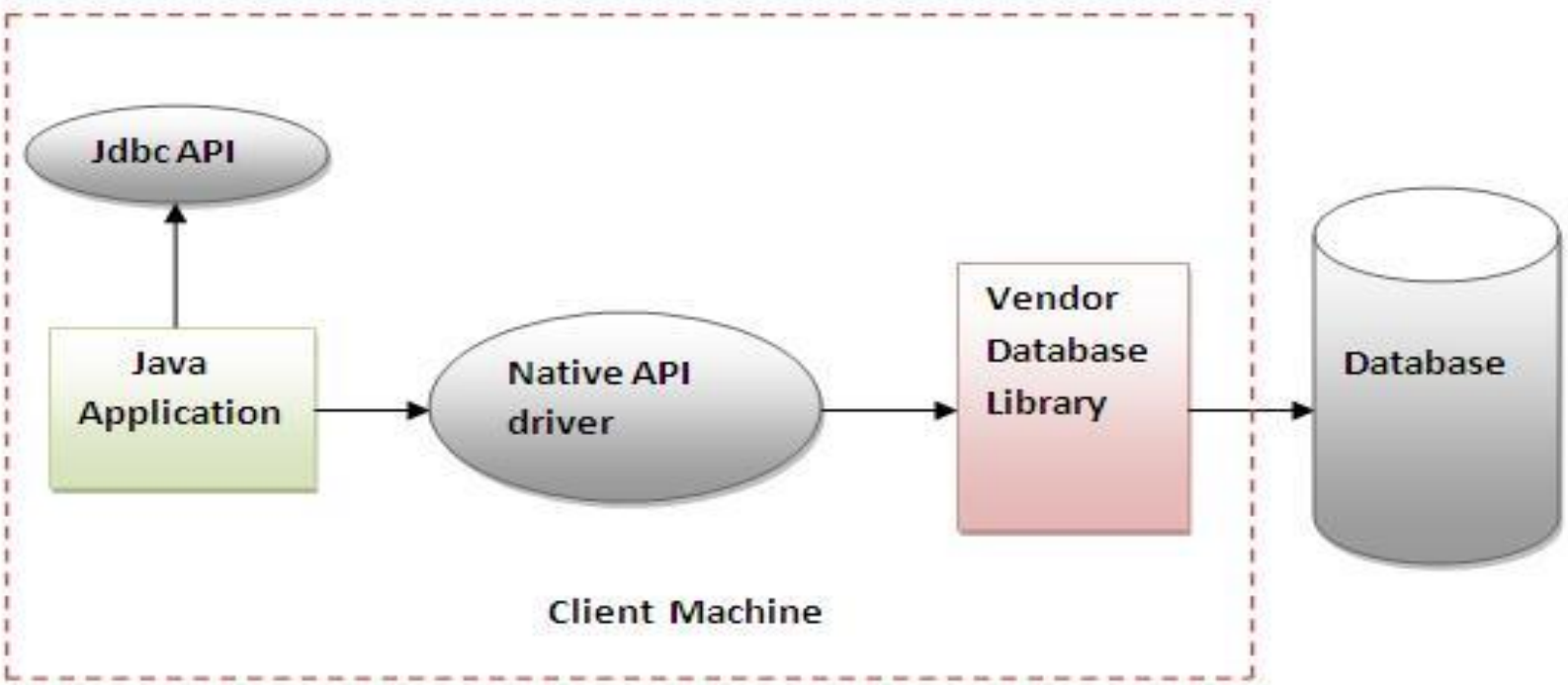


Figure- Native API Driver

- Network Protocol driver
  - The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.
  - Advantage:
    - No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
  - Disadvantages:
    - Network support is required on client machine.
    - Requires database-specific coding to be done in the middle tier.
    - Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

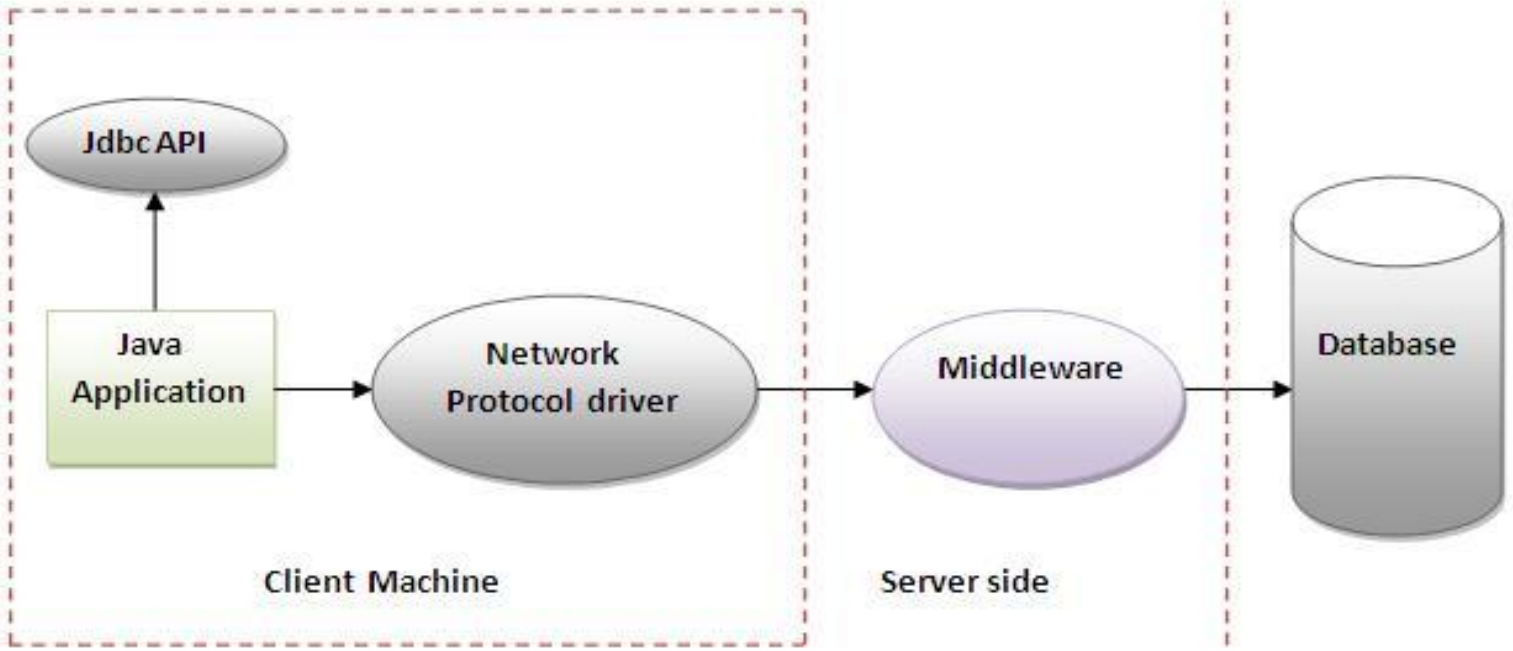


Figure- Network Protocol Driver

- Thin driver
  - The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.
  - Advantage:
    - Better performance than all other drivers.
    - No software is required at client side or server side.
  - Disadvantage:
    - Drivers depends on the Database.

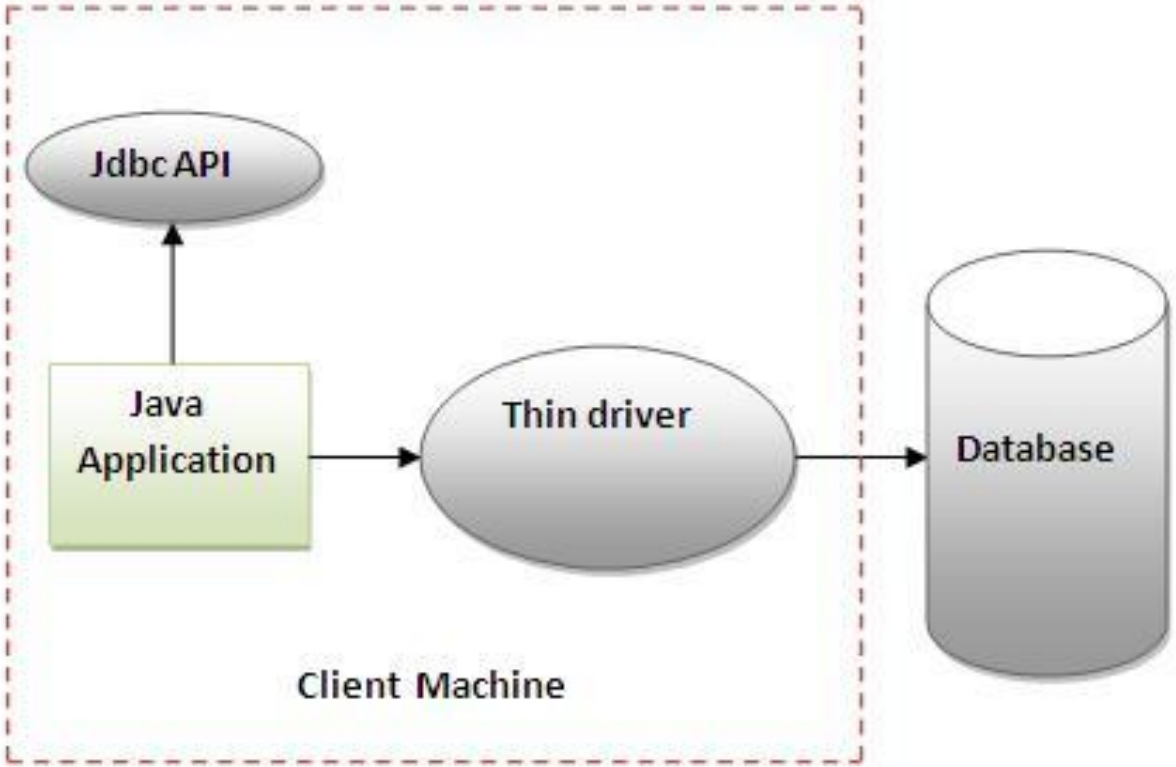


Figure- Thin Driver