UNIT - III

INTRODUCTION TO SERVLETS

Servlet technology is used to create web application (resides at server side and generates dynamic web page).

* Servlet technology is robust and scalable because of java language. Before servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there are many disadvantages for this technology.

* There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

What is a Servlet ?

Servlet is described in many ways, depending on the context.

* Servlet is a technology i.e., used to create web application.

* Servlet is an API that provides many interfaces and classes including documents.

* Servlet is an interface that must be implemented for creating any servlet.

* Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any types of requests

* Common Gateway Interface (CGI):

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

Disadvantages of CGI:

There are many problems in CGI technology:

1) If number of clients increases, it takes more time for sending response.

2) For each request, it starts a process and web server is limited to start processes.

3) It uses platform dependent language. e.g., C, C++, perl.

Advantage of Servlet:

There are many advantages of servlet over CGI. The web container creates threads for handling the multiple requests to the servlet. Threads have a lot of benefits over the processes such as they share a common memory area, light weight, cost of communication between the threads are low. The basic benefits of servlet are as follows:

a) Better performance:
because it creates a thread for each request not process.

b) Portability:

because it gives java language.

c) Robust:

Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.

d) Secure:

because it uses java language.

* Life-Cycle Of a Servlet:

Three methods are central to the life cycle of a servlet. These are init(), service() and destroy(). They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

* First, assume that a user enters a Uniform Resource Locator (URL) to a Web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

* Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

* Third, the server invokes the 'init()' method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

* Fourth, the server invokes the service() method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

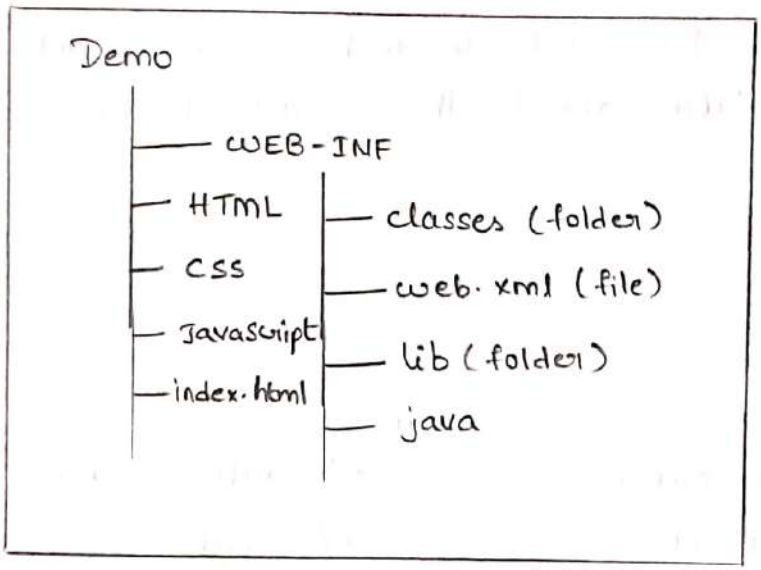
* The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The service() method is called for each HTTP request.

* Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the destroy() method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistant store. The memory allocated for the servlet and its objects can then be garbage collected.

Steps to create Servlet Program:

1) create a directory structure
2) create a servlet
3) compile the servlet
4) create a deployment descriptor
5) start the server and deploy the project.
6) Access the servlet.

1) create a directory structure:



structure of servlet

2) create a servlet:

servlet can be created in 3 different ways.

a) By implementing servlet interface.
b) By inheriting the GenericServlet class
c) By inheriting the HttpServlet class.

3) Compile the servlet :

Servlet can be compiled by using jar files. jar files are required to be loaded.

servlet-api.jar

↓

we can set the classpath or it can be copied manually into the server.

4) create a deployment descriptor :

Deployment descriptor is an XML file with a name web.xml.

* From this xml file web container gets the information about the servlet to be invoked.

web.xml

```
<web-app>
  <servlet>
    <servlet-name> filename </servlet-name>
    <servlet-class> classname </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name> filename </servlet-name>
    <url-pattern> replica </url-pattern>
  </servlet-mapping>
</web-app>
```

5) Start the servlet and deploy the project:

There are two ways to deploy the project

a) hard deployment

b) soft deployment

a) soft deployment:

Copy the demo folder into server manually.

In htdocs (i.e., server) copy & paste the complete folder of the project.

In appache web-apps is the server.

c://xampp/Tomcat/web-apps

↑

copy complete folder.

b) soft deployment:

For soft deployment first create a war file.

By compressing all files (i.e., zipping) a war file is generated.

* A war file is created with Jar commands.

localhost:8080/manager/html.

D:/>Demo/Jar cvf Demo.war;

6) Access the servlet:

http://localhost/Demo →

if filename is not specified then it directly access index.html.

* The Servlet API :

Two packages contain the classes and interfaces that are required to build servlets. They are :

a) javax.servlet

b) javax.servlet.http

a) javax.servlet package :

Interface	Description
Servlet	Declares lifecycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters
ServletContext	Enables servlets to log events and access information about their environment
ServletRequest	Used to read data from client request.
ServletResponse	Used to write data to a client response.
SingleThreadModel	Indicates that the servlet is thread safe.

class	Description
Generic Servlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	provides an input stream for reading requests from a client.
ServletOutputStream	provides an output stream for writing responses to a client.
ServletException	indicates a servlet error occured.
UnavailableException	indicates a servlet is unavailable.

The Servlet Interface

All servlets must implement the Servlet Interface. It declares the init(), service() and destroy() methods that are called by the server during the life cycle of a servlet. The methods defined by servlet are shown in table below.

Method	Description
void destroy()	called when the servlet is unloaded.
ServletConfig getServletConfig()	returns a ServletConfig object that contains any initialization parameters
String getServletInfo()	returns a string describing the servlet.

void init(ServletConfig sc) throws ServletException	called when the servlet is initialized. Initialization parameters for the servlet can be obtained from sc. An UnavailableException should be thrown if a servlet cannot be initialized.
void service(ServletRequest req, ServletResponse res) throws ServletException, IOException	called to process a request from a client. The request from the client can be read from req. The response to the client can be written to res. An exception is generated if a servlet or IO problem occurs.

The ServletConfig Interface

The ServletConfig interface is implemented by the server. It allows the servlet to obtain configaration data when it is loaded. The methods declared by this interface are summarized here:

Method	Description
ServletContext getServletContext()	returns the context for this servlet.
String getInitParameter (String param)	returns the value of initialization parameter named Param
Enumeration getInitParameterNames()	returns an enumeration of all initialization parameter names.
String getServletName()	returns the name of the invoking servlet.

The ServletContext Interface

The ServletContext interface is implemented by the server. It enables the servlets to obtain information about their environment. Several of its methods are summarized in table below:

Method	Description
Object getAttribute (String attr)	returns the value of the server attribute named attr
String getMimeType (String file)	returns the MIME type of file.
String getRealPath (String vpath)	returns the real path that corresponds to the virtual path vpath.
String getServerInfo()	returns information about the server.
void log(String s)	writes s to the servlet log
void log(String s, Throwable e)	write s and the stack trace for e to the servlet log
void setAttribute (String attr, Object val)	sets the attribute specified by attr to the value passed in val

The ServletRequest Interface

The ServletRequest interface is implemented by the server. It enables a servlet to obtain information about the client request. Several of its methods are summarized in table below.

Method	Description
Object getAttribute (String attr)	returns the value of the attribute named attr.
String getCharacterEncoding ()	returns the character encoding of the request
int getContentLength()	returns the size of the request. The value -1 is returned if the size is unavailable.
String getContentType()	returns the type of the request. A null value is returned if the type cannot be determined.
ServletInputStream getInputStream() throws IOException	returns the ServletInputStream that can be used to read binary data from the request. An IllegalStateException is thrown if getReader() has already been invoked for this request.
String getParameter (String pname)	returns the value of the parameter named pname.
Enumeration getParameterNames ()	returns an enumeration of the parameter names for this request.
String[] getParameterValues (String name)	returns an array containing values associated with the parameter specified by name
String getProtocol()	returns a description of the protocol.

BufferedReader getReader() throws IOException	returns a buffered reader that can be used to read text from the request. An IllegalStateException is thrown if getInputStream() has already been invoked for this request.
String getRemoteAddr()	returns the string equivalent of the client IP address.
String getRemoteHost()	returns the string equivalent of the client host name.
String getScheme()	returns the transmission scheme of the URL used for the request.
String getServerName()	returns the name of the server.
int getServerPort()	returns the port number.

The ServletResponse Interface

The ServletResponse interface is implemented by the server. It enables a servlet to formulate a response for a client. Several of its methods are summarized in table below.

Method	Description
String getCharacterEncoding()	returns the character encoding for the response
ServletOutputStream getOutputStream() throws IOException	returns a ServletOutputStream that can be used to write binary data to the response. An IllegalStateException is thrown if getWriter() has already been invoked for this request.

PrintWriter getWriter() throws IOException	returns a PrintWriter that can be used to write character data to the response. An IllegalStateException is thrown if getOutputStream() has already been invoked for this request.
void setContentLength (int size)	sets the content length for the response to size.
void setContentType (String type)	sets the content type for the response to type

The SingleThreadModel Interface

This interface is used to indicate that only a single thread will execute the service() method of a servlet at a given time. It defines no constants and declares no methods. If a servlet implements this interface, the server has two options. First, it can create several instances of the servlet. When a client request arrives, it is sent to an available instance of the servlet. Second, it can synchronize access to the servlet.

The GenericServlet Class

The GenericServlet class provides implementations of the basic life cycle methods for a servlet and is typically subclassed by servlet developers. GenericServlet implements the Servlet and ServletConfig interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

void log(String s)
void log(String s, Throwable e)

Here, s is the string to be appended to the log, and e is an exception that occured.

The ServletInputStream Class

The ServletInputStream class extends InputStream. It is implemented by server and provides an input stream that a servlet developer can use to read the data from the client request. It defines the default constructor. In addition, a method is provided to read bytes from the stream. Its signature is shown here:

int readLine(byte[] buffer, int offset, int size)
throws IOException

* Here, array buffer is the array into which size bytes are placed starting at offset. The method returns the actual number of bytes or -1 if the end-of-the stream condition is encountered.

The ServletOutputStream class

The ServletOutputStream class extends OutputStream. It is implemented by the server and provides an output stream that a servlet developer can be used to write data to a client response. A default constructor is defined. It also defines print() and println() methods, which output data to the stream

* Handling Http Request & Responses

The HttpServlet class provides specialized methods that handle the various types of Http requests. A servlet developer typically overrides one of these methods. These methods are doDelete(), doGet(), doHead(), doOptions(), doPost(), doPut() and doTrace(). A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling from input. Therefore, this section presents examples of these cases.

* Handling HTTP GET Requests Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a Web page is submitted. The example contains two files. A Web page is defined in ColorGet.htm and a servlet is defined in ColorGetServlet.java. The HTML source code for ColorGet.htm. is shown in following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
  <body>
    <center>
    <form name = "Form1"
        action = "http://localhost:8080/examples/
                                servlet/ColorGetServlet">
    <B> Color </B>
    <select name = "Color" size = "1">
    <option value = "Red"> Red </option>
    <option value = "Green"> Green </option>
    <option value = "Blue"> Blue </option>
    </select>
    <br><br>
    <input type = submit value = "Submit">
    </form>
  </body>
</html>
```

* The source code for ColorGetServlet.java is shown in the following listing. The doGet() method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the getParameter() method of HttpServletRequest to obtain this selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet
{
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                  throws ServletException, IOException
   {
      String color = request.getParameter("color");
      response.setContentType("text/html");
      PrintWriter pw = response.getWriter();
      pw.println("<B>The selected color is: ");
      pw.println(color);
      pw.close();
   }
}
```

Compile the servlet and perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the Web page in a browser.
3. Select a color.
4. Submit the Web page.

* After completing these steps, the browser will display the response that is dynamically generated by the servlet. One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the Web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

http://localhost:8080/examples/servlet/ColorGetServlet?

Color = Red. The characters to the right of the question mark are known as the query string.

Handling HTTP POST Requests.

* Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a Web page is submitted. The example contains two files. A web page is defined in ColorPost.htm ~~and a servlet is define~~ is shown in the following listing. It is identical to ColorGet.htm except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>
  <body>
   <center>
   <form name = "form1" method = "post"
         action = "http://localhost:8080/examples/
                              servlet/ColorPostServlet">
    <B> Color: </B>
    <select name = "color" size = "1">
    <option value = "Red"> Red </option>
    <option value = "Green"> Green </option>
    <option value = "Blue"> Blue </option>
    </select>
    <br> <br>
    <input type = submit value = "Submit">
   </form>
  </body>
</html>
```

(11)

The source code for ColorPostServlet.java is shown in the following listing. The doPost() method is overridden to process any HTTP POST request to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
Public class ColorPostServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                      throws ServletException, IOException
    {
        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is:");
        pw.println(color);
        pw.close();
    }
}
```

Compile the Servlet and perform the same steps as described in the previous section to test it.

Note:

Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is:

http://localhost:8080/examples/servlet/ColorGetServlet

The parameter names and values are sent in the body of the HTTP request.

* Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a Web page is submitted. The example contains three files as summarized here:

File Description AddCookie.htm Allows a user to specify a value for the cookie named MyCookie.

* AddCookieServlet.java Processes the submission of AddCookie.htm

* GetCookiesServlet.java Displays cookie values.

* The HTML source code for AddCookie.htm is shown in the following listing.

* This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to AddCookie Servlet via an HTTP POST request.

```
<html>
  <body>
    <center>
    <form name = "Form1" method = "post"
        action = "http://localhost:8080/examples/
                              servlet/AddCookieServlet">
    <B> Enter the value for MyCookie: </B>
    <input type=textbox name="data" size=25
                                  value = " ">
    <input type= submit value = "submit">
    </form>
```

```
</body>
</html>
```

The source code for AddCookieServlet.java is shown in the following listing. It gets the value of the parameter named "data". It then creates a Cookie object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the addCookie() method. A feedback message is then written to the browser.

```
import java.io.*
import javax.servlet.*;
import javax.servlet.http.*;
public class AddCookieServlet extends HttpServlet
{
   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                throws ServletException, IOException
   {
      // get parameter from HTTP request
      String data = request.getParameter("data");
      // create cookie
      Cookie cookie = new Cookie("MyCookie", data);
      // add cookie to HTTP response
      response.addCookie(cookie);
      // write output to browser
      response.setContentType("text/html");
      PrintWriter pw = response.getWriter();
      pw.println("<B> MyCookie has been set to");
```

```
        pw.println(data);
        pw.close();
    }
}
```

* The source code for GetCookiesServlet.java is shown in the following listing. It invokes the getCookies() method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the getName() and getValue() methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookiesServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException,IOException
    {
        // Get cookies from header of HTTP request
        Cookie[] cookies= request.getCookies();
        // Display these cookies.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i=0; i< cookies.length ; i++)
        {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = "+name+" value = "+
                                        value);
    }
```

```
        pw.close();
    }
}
```

Compile the servlet and perform these steps:

1. Start Tomcat, if it is not already running.
2. Display AddCookie.htm in a browser.
3. Enter the value for MyCookie
4. Submit the web page.

After completing these steps you will observe that a feedback message is displayed by the browser. Next, request the following URL via the browser

http://localhost:8080/examples/servlet/GetCookieServlet

Observe that the name and value of the cookie are displayed in the browser.

* In this example, an expiration date is not explicitly assigned to the cookie via the setMaxAge() method of Cookie. Therefore, the cookie expires when the browser session ends. You can experiment by using setMaxAge() and observe that the cookie is then saved to the disk on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

* A session can be created via the getSession() method of HttpServletRequest. An HttpSession object is returned. This object can store a set of bindings that associate names with objects. The setAttribute(), getAttribute(), getAttributeNames() and removeAttribute() methods of HttpSession manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.

* The following servlet illustrates how to use session state. The getSession() method gets the current session. A new session is created if one does not already exist. The getAttribute() method is called to obtain the object that is bound to the name "date". That object is a Date object that encapsulates the date and time when this page was last accessed. A Date object encapsulating the current date and time is then created. The setAttribute() method is called to bind the name "date" to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DateServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
                      throws ServletException, IOException.
    {
        // Get the HttpSession object
        HttpSession hs = request.getSession (true);
        // Get Writer
        response.setContentType ("text/html");
        PrintWriter pw = response.getWriter ();
        pw.print ("<B>");
        // Display date/time of last access
        Date date = (Date) hs.getAttribute ("date");
        if (date != null)
        {
            pw.print ("Last access:" + date + "<br>");
        }
        // Display current date/time
        date = new Date ();
        hs.getAttribute ("date", date);
        pw.println ("Current date :" + date);
    }
}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

* Introduction to JDBC :-

→ In today's scenario, many enterprise level applications need to intract with databases for storing information.

→ For this purpose, we used an API (Application programming Interface) i.e ODBC (open Database connectivity).

→ The ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e platform dependent and unsecured).

→ That is why java has defined its own API, called JDBC (Java Database connectivity). That uses JDBC drivers (Written in Java language).

→ The JDBC drivers are more compatible with Java Applns to provide database communication.

→ JDBC is a Java API to connect and execute query with the database. JDBC API uses 'jdbc drivers to connect with the database.

→ JDBC supports a wide level of portability and JDBC is simple and easy to use.

→ In JDBC API, a programmer needs a specific driver to connect to specific database.

RDBMS	Driver
Oracle	oracle.jdbc.driver.OracleDriver
MySQL	com.mysql.jdbc.Driver
SyBase	com.sybase.jdbc.SybDriver
SQLServer	com.microsoft.jdbc.SqlServer
DB2	com.ibm.db2.jdbc.net.DB2Driver

* List of some popular Drivers.*

* JDBC Architecture :-

The main function of the JDBC is to provide a standard abstraction for java applications to communication with databases.

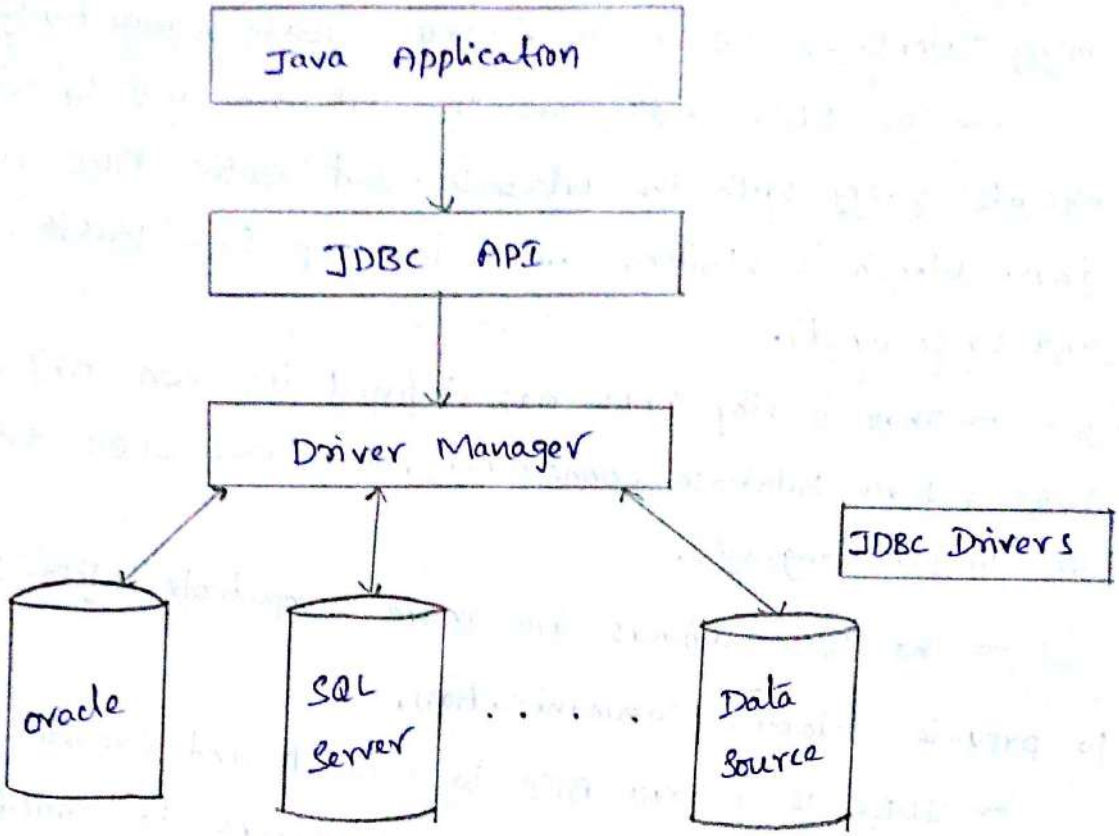


Fig:- The JDBC Architecture

As shown in figure, The Java application that wants to communicate with a database has to be programmed using JDBC API.

The JDBC Driver is required to process the SQL requests and generate the results.

The JDBC driver has to be plays an important role in the JDBC architecture. The Driver Manager uses some specific drivers to effectively connect with specific databases.

→ JDBC Driver is a software component that enables Java application to intract with the database.

There are 4 types of JDBC drivers, those are

→ Type -1 Driver (JDBC-ODBC bridge driver)
→ Type - 2 Driver (partial JDBC driver)
→ Type - 3 Driver (pure java driver for middleware)
→ Type - 4 Driver (pure java driver with direct database connection)

* Type-1 Driver (JDBC-ODBC bridge driver) :-

The type -1 driver acts as a bridge between JDBC and other database connectivity mechanisms such as ODBC.

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 method calls.

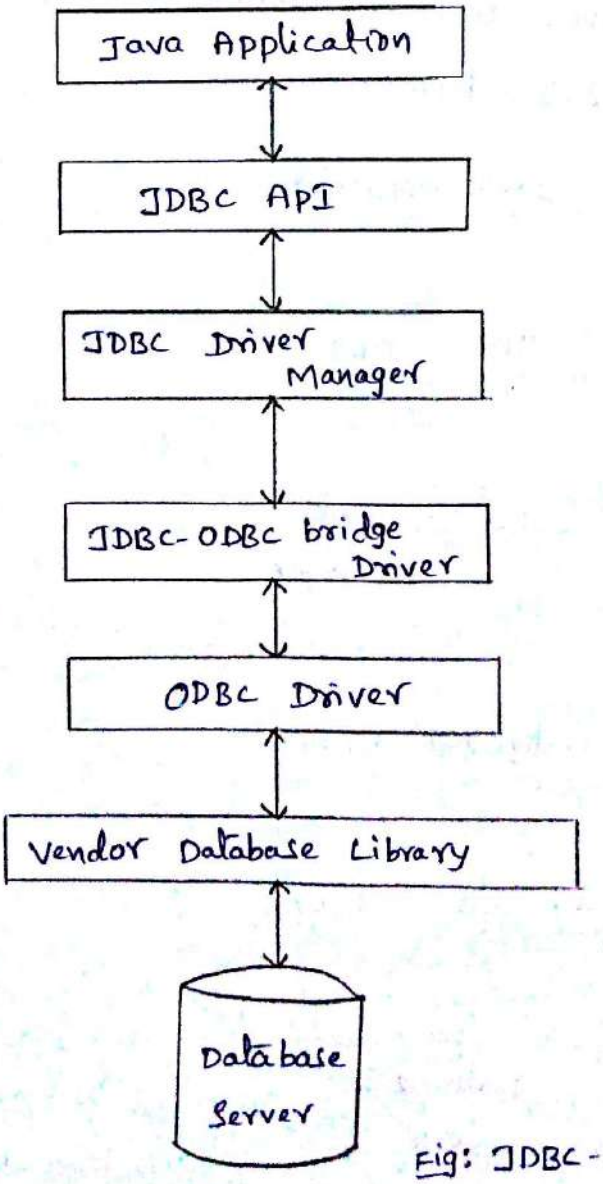


Fig: JDBC-ODBC Bridge Driver

Advantages:

* Easy to use.
* Can be easily connected to any database.

Disadvantages:

* performance degraded because large number of trans-lations (i.e JDBC calls to ODBC calls).
* The ODBC driver needs to be installed on the client machine.

* Type-2 Driver (partial JDBC driver):-

The type-2 driver uses the client-side libraries of the database. so This driver is also called as Native-API driver.

This driver converts JDBC method calls into native calls of the database API. It is not written entirely in Java, so it is called as partial JDBC driver.

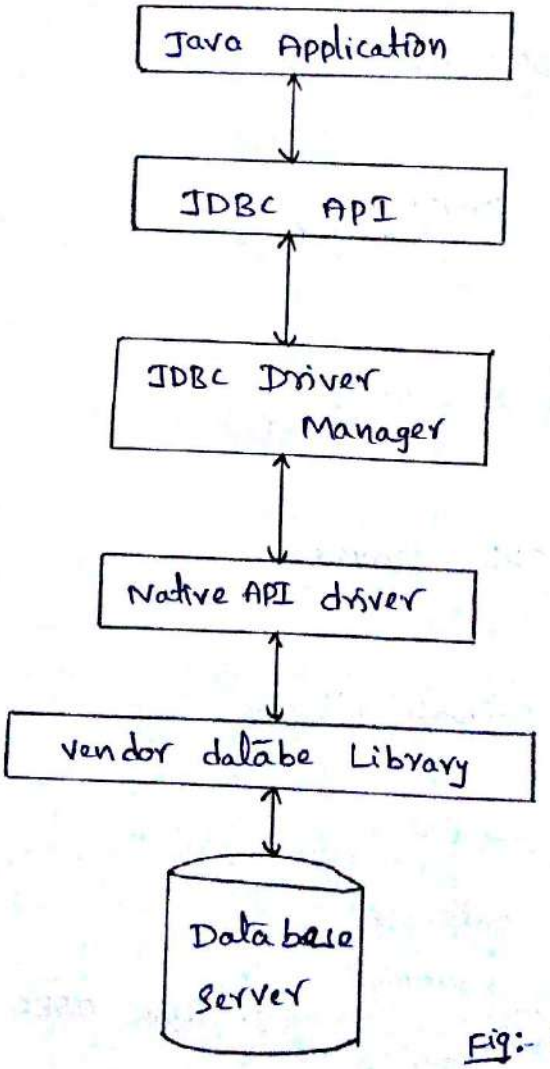


Fig:- Native API driver

Advantages:

* performance upgraded than JDBC-ODBC bridge driver.
* suitable to use with server-side applications.

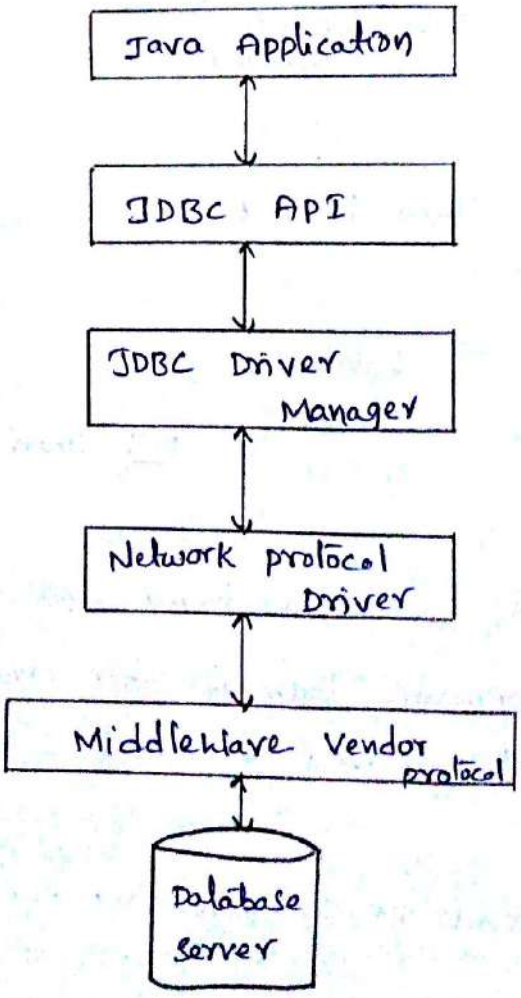
Disadvantages:

* This Native driver needs to installed on the each client machine
* The vendor client library needs to be installed on client machine.
* It may increase the cost of the application if the application needs to run on different platforms.

* Type-3 Driver (pure Java driver for middleware):-

The type-3 driver is completely implemented in Java, hence it is a pure Java JDBC driver.

The type-3 driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. so it is called as Network protocol driver.



Advantages:

* No client side library is required on client side.
* pure Java drivers and auto downloadable.

Disadvantages:

* Network support is required on client machine.
* This driver is costly compared to other drivers.

* Type-4 Driver (pure Java driver with direct database connection):-

The type-4 driver is a pure Java driver, which converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver.

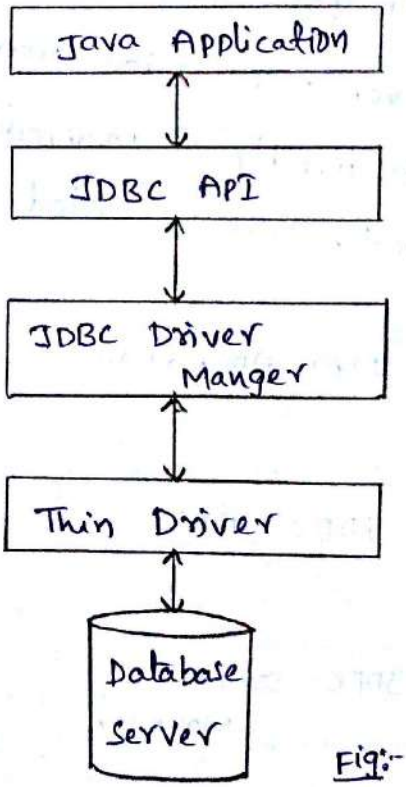


Fig:- Thin Driver.

Advantages:

* This driver is pure Java driver and auto downloadable.
* Better performance than all other drivers
* No software is required at client side or server side.

Disadvantages:

* Drivers depends on the Database.

* Database programming using JDBC :-

JDBC APIs are used by a Java application to communi-cate with a database.

In otherwords, we use JDBC connectivity code in Java application to communicate with a database.

There are 5 steps to connect any Java application with the database in java using JDBC. They are as follows:

Step 1 : Register the driver class

Step 2 : Creating connection

Step 3 : Creating statement

Step 4 : Executing SQL statements

Step 5 : closing connection.

* Step 1 :- (Register the driver class)

In this step, we register the driver class with driver Manager by using forName() method of class class.

Syntax: Class.forName(Driver class Name)

Example: Class.forName("oracle.jdbc.driver.OracleDriver");

* Step 2 :- (Creating connection)

In this step, we can create a connection with database Server by using getConnection() method of DriverManager class.

Syntax: getConnection(string url, string name, string pwd)

Example:

```
Connection con = DriverManager.getConnection(
                    "jdbc:oracle:thin:@localhost:1521:xe",
                                "system", "admin");
```

* Step 3 :- (Creating statement)

After the connection made, we need to create the statement object to execute the SQL statements.

The createStatement() method of Connection interface is used to create statement. This statement object is responsible to execute SQL statements with the database.

Syntax: createStatement()

Example:

```
Statement stmt = con.Statement();
```

* Step 4:- (Executing SQL Statements)

After the statement object is created, it can be used to execute the SQL statements by using executeUpdate() (or) executeQuery() method of statement interface.

The executeQuery() method is only used to execute SELECT statements.

The executeUpdate() method is used to execute all SQL statements except SELECT statements.

Syntax: executeQuery(String query)
executeUpdate(String query)

Example:

```
// using executeQuery()
String query = "Select * from emp";
ResultSet rs = stmt.executeQuery(query);
// using executeUpdate()
String query = "insert into emp values(504,'Madhu',29);
stmt.executeUpdate(query);
```

* Step 5:- (closing the connection)

After executing all the SQL statements and obtaing the results, we need to close the connection and release the session.

The close() method of Connection interface is used to close the connection.

Syntax:- close()

Example:- con.close();

* Example :- (connectivity with oracle database)

For connecting java application with the oracle database, we need to know following information to perform database connectivity with oracle.

In this example we are using oracle 10g as the database, so we need to know following information for the oracle database.

* Driver class : The driver class for oracle database is "oracle.jdbc.driver.OracleDriver".
* Connection URL : The connection URL for the oracle 10G database is "jdbc:oracle:thin:@localhost:1521:xe".

Where jdbc is the API, oracle is the database, thin is the driver, localhost is the servername on which oracle is running, 1521 is the port number and XE is the oracle service name.

* username : The default username for the oracle database is "system".
* password : password is given by the user at the time of installing the oracle database.

→ To connect java application with the oracle database ojdbc14.jar file is required to be loaded.

→ There are two ways to load the ojdbc14.jar file, we need to follow any one of two ways.

1. paste the ojdbc14.jar file in "java/jre/lib/ext" folder
2. set classpath

Firstly, search the ojdbc14.jar file then go to "java/jre/lib/ext" folder and paste the jar file here.

(or)

Set class path : To set classpath, goto environment variable then click on new tab. In variable name write classpath and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as

"c:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;".

* Example:

Let's first create a table and insert two or more records in oracle database.

```
SQL> create table emp(id number(10), name varchar2(40),
                                    age number(3));

SQL> insert into emp values(501, 'Madhu', 30);
SQL> insert into emp values(502, 'Hari', 32);
SQL> insert into emp values(503, 'Satti', 33);
```

* program: connect java application with oracle database for selecting or retriving data.

SelectData.java

```
import java.sql.*;
import java.util.*;
class SelectData
{
public static void main(String args[])
{
try
{
// Step 1: load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");
// Step 2: create the connection object
Connection con = DriverManager.getConnection(
   "jdbc:oracle:thin:@localhost:1521:xe", "system", "admin");
// Step 3: create the statement object
Statement stmt = con.createStatement();
// Step 4: execute query
ResultSet rs = stmt.executeQuery("select * from emp");
while(rs.next())
{
System.out.println(rs.getInt(1) + "    " + rs.getString(2) + "    "
                   + rs.getString(3));
}
// Step 5: close the connection object
con.close();
}catch(Exception e) { System.out.println(e);}
}}
```

output:

```
D:\> javac SelectData.java
D:\> java Select Data
501     Madhu     30
502     Hari      32
503     Satti     33
```

* program: connect Java application with oracle database for inserting data.

InsertData.java

```
import java.sql.*;
import java.util.*;
class InsertData
{
public static void main (String args[])
{
 try
 {
 Class.forName ("oracle.jdbc.driver.OracleDriver");
 Connection con = DriverManager.getConnection ("jdbc : oracle : thin
                  :@localhost : 1521 : xe", "system", "admin");

 Statement stmt = con.createStatement();
 stmt.executeUpdate ("insert into emp values (504, 'Ganesh', 28)");
 System.out.println ("Inserted ...");
 con.close();
 }
 catch (Exception e)
  {
   System.out.println (e);
  }
}
}
```

output:

```
D:\> javac InsertData.java
D:\> java InsertData
Inserted...
```

* program: java application with oracle database for update data.

Updatedata.java

```
import java.sql.*;
import java.util.*;
class Updatedata
{
public static void main(String args[])
{
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection("jdbc:oracle:
        thin:@localhost:1521:xe", "system", "admin");
Statement stmt = con.createStatement();
stmt.executeUpdate("update emp set age=38 where id=503");
System.out.println("updated....");
con.close();
}
catch(Exception e)
{
  System.out.println("Exception is:" +e);
}
}
}
```

output:

```
D:/> javac Updatedata.java
D:/> java Updatedata
Updated....
```

* DriverManager class :-

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are avaliable and handles establishing a connection between a database and the appropri-ate driver.