1. Discuss the URL and URLConnection class with their use in network programming.

The **URLConnection** class is used to connect to a website or resource on a network and get all the details of resource on a website.

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

http://www.darshan.ac.in

A **URL** contains many information:

- **Protocol:** In this case, http is the protocol.
- **Server name or IP Address:** In this case, www.darshan.ac.in is the server name.
- **Port Number:** It is an optional attribute. If we write http//ww.darshan.ac.in:80/index/ 80 is the port number. If port number is not mentioned in the URL, it returns -1.
- File Name or directory name: In this case, index.jsp is the file name.

Using OpenConnection() method of the URL object, we can contact with the resource on a website is established :

```
import java.io.*;
import java.net.*;
import java.util.*;

class Details {
        public static void main(String args[ ]) throws Exception        {
                //pass the site url to URL object
                URL obj = new URL("http://www.microsoft.com");

                //open a connection with the site on Internet
                URLConnection conn = obj.openConnection();

                //display the date
                System.out.println("Date: "+ new Date(conn.getDate()));

                //display how many bytes the index.html page has
                int l = conn.getContentLength();
                System.out.println("Length of content: "+ l);
                if(l == 0)                {
                        System.out.println("Content not available");
                        return;
                }
                else  {
                        int ch;
                        InputStream in = conn.getInputStream();

                        //display the content of the index.html page
                        int i = 0;
                        while((ch = in.read())!= -1)
                                System.out.print((char)ch);
                }
        }
}
```

2. Explain Socket, ServerSocket, InetAddress classes. Write a java program to find an IP address of the machine on which the program runs.

Socket :

For implementing client side sockets this class is useful. It accepts two arguments ipaddress and port no.

Socket s = new Socket("localhost",777)

ServerSocket :

For implementing server side sockets this class is useful. It accepts one argument port no.

ServerSocket ss = new ServerSocket(777)

InetAddress :

The java.net package provides an **InetAddress** class that allows you to retrieve the IP Address, some of the methods are :

getLocalHost() method will return the object of InetAddress Class
getHostAddress() method then will return the IP Address of the Local System.

Example : GetIPAddress.java :

```
import java.net.*;
import java.io.*;
public class GetIPAddress  {
        public static void main(String [] args) {
                try {
                        InetAddress thisIp =InetAddress.getLocalHost();
                System.out.println("IP Address of this System:"+thisIp.getHostAddress());
            }
            catch(Exception e) {
                e.printStackTrace();
            }
        }
}
```

3. One-way Communication using Connection-Oriented Protocol (TCP)

Example : TCPServerOneWay.Java

```
import java.io.*;
import java.net.*;
class TCPServerOneWay {
        public static void main(String args[ ]) throws Exception {
            //Create a server socket with some port number
            ServerSocket ss = new ServerSocket(777);

            //let the server wait till a client accepts connection
            Socket s = ss.accept();
            System.out.println("Connection established");

            //attach output stream to the server socket
            OutputStream obj = s.getOutputStream();

            //attach print stream to send data to the socket
            PrintStream ps = new PrintStream(obj);

            //send 2 strings to the client
            String str = "Hello client";
            ps.println(str);
            ps.println("Bye");

            //close connection by closing the streams and sockets
            ps.close();
            ss.close();
            s.close();
        }
}
```

Example : TCPClientOneWay.Java

```
import java.io.*;
import java.net.*;
class TCPClientOneWay {
        public static void main(String args[ ]) throws Exception {
                //create client socket with same port number
                Socket s = new Socket("localhost", 777);

                //to read data coming from server, attach InputStream to the socket
                InputStream obj = s.getInputStream();

                //to read data from the socket into the client, use BufferedReader
                BufferedReader br = new BufferedReader(new InputStreamReader(obj));

                //receive strings
                String str;
                while((str = br.readLine()) != null)
                        System.out.println("From server: "+str);

                //close connection by closing the streams and sockets
                br.close();
                s.close();
        }
}
```

4. Two-way Communication using Connection-Oriented Protocol (TCP)

Example : TCPServerTwoWay.Java

```
import java.io.*;
import java.net.*;
class Server2{
        public static void main(String args[ ]) throws Exception {

//Create server socket
                ServerSocket ss = new ServerSocket(888);
```

```
                    //Create server socket
                    ServerSocket ss = new ServerSocket(888);

                    //connect it to client socket
                    Socket s = ss.accept();
                    System.out.println("Connection established");

                    //to send data to the client
                    PrintStream ps = new PrintStream(s.getOutputStream());

                    //to read data coming from the client
                    BufferedReader br = new BufferedReader(new
 InputStreamReader(s.getInputStream()));

                    //to read data from the key board
                    BufferedReader kb = new BufferedReader(new
 InputStreamReader(System.in));
//server executes continuously
                    while(true) {
                              String str,str1;

                              //repeat as long as client does not send null string
                              while((str = br.readLine()) != null)  {
                                        System.out.println(str);
                                        str1 = kb.readLine();
                                        ps.println(str1); //send to client
                              }
                              //close connection
                              ps.close();
                              br.close();
                              kb.close();
                              ss.close();
                              s.close();
                              System.exit(0); //terminate application
                    } //end of while
          }
}
```

5. One-way Communication using Connection-Less Protocol (UDP)

Example : UDPServerEx.Java

```
//A server that sends a messages to the client
import java.net.*;

class UDPServerEx{
	public static DatagramSocket mySocket;
	public static byte myBuffer[]=new byte[2000];
	public static void serverMethod() throws Exception{
		int position=0;
		while(true){
			int charData=System.in.read();
			switch(charData){
				case -1:
					System.out.println("The execution of the server has been
					terminated");
					return;
				case '\r':
					break;
				case '\n':
					mySocket.send(new
					DatagramPacket(myBuffer,position,InetAddress.getLocalH
					ost(),777));
					position=0;
					break;
				default:
					myBuffer[position++]=(byte) charData;
			}
		}
	}

	public static void main(String args[]) throws Exception {
		System.out.println("Please enter some text here");
		mySocket=new DatagramSocket(888);
		serverMethod();
	}
}
```

Example : UDPClient.Java

```
//UDPClient - receives and displays messages sent from the server
import java.net.*;
class UDPClient {
	public static DatagramSocket mySocket;
	public static byte myBuffer[]=new byte[2000];

	public static void clientMethod() throws Exception {
		while(true) {
			DatagramPacket dataPacket=new
			DatagramPacket(myBuffer,myBuffer.length);

			mySocket.receive(dataPacket);
			System.out.println("Message Recieved :");

			System.out.println(new
			String(dataPacket.getData(),0,dataPacket.getLength()));
		}
	}

	public static void main(String args[]) throws Exception {
		System.out.println("You need to press CTRL+C in order to quit");
		mySocket=new DatagramSocket(777);
		clientMethod();
	}
}
```

6. Protocol Handler & Content Handler

- When Java Interacts with Network Programming, then it is categorized into two distinct domains called :
  - Protocol Handler
  - Content Handler
- **Protocol Handler** is required to take care of certain issues: - Generating client request, on receipt of data, acknowledging that the data has been received.
- **Content Handler** is required to :- Convert the raw data in the format which Java can understand
- Working of these two categories is totally different and independent. That means protocol does not care about the type of downloading data. Similarly data getting downloaded does not have to worry about the protocol.

Protocol Handler :

- Protocol Handler can be defined as a collection of classes which know how to handle protocols.
- Various protocols supported by JAVA are http,ftp,file.mailto etc.
- java.net is the package for all this magic.
- In java.net package there are four important classes defiend
    - URL – it is the main class
    - URLStreamHandler – it is an abstract class
    - URLConnection – it is an abstract class (getInputStream, getOutputStream)
    - URLStreamHandlerFactory – it is an interface

Content Handler :

- What is the Content ?
- When we request for viewing some file on the web browser and if it is a simple HTML file then that particular file will be displayed in the browser.
- Sometimes we can request for viewing some gif and jpg file and the browser displays it.
- Sometimes we want to see some pdf or ppt file, then browser takes help of some external program.
- Actually the items which get displayed by your browser are reffered to as contents.
- Each time when browser wants to handle different contents, plug-ins are used. These plug-ins that are required to handle various contents on web browser are called content-handler.

1. Explain various types of JDBC drivers and comment on selection of driver.

JDBC Drivers Types:

- JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates.
- The different types of drivers available in JDBC are listed in following table:

JDBC driver type	**Description**
Type-1 (JDBC-ODBC Bridge)	Refers to the bridge driver(jdbc-odbc diver)
Type-2 Driver (Native)	Refers to partly java and partly native code driver(native partly java driver)
Type-3 Driver (Middleware)	Refers to a pure java driver that uses a middleware driver connect to a database(pure java driver for database middleware)
Type-3 Driver (Pure Java)	Refers to a pure java driver(pure).which is directly connect to a database

Table 2.1 : JDBC Driver Type

Type-1 Driver (JDBC-ODBC Bridge) :

- Type-1 driver act as a bridge between jdbc and other connecting mechanisms i.e.sun JDBC-ODBC bridge driver.
- This driver also included in the java2 SDK within the sun.jdbc.odbc package.
- This driver converts jdbc calls into odbc calls and redirect the request to odbc driver.

Advantages :

- It is represent the single driver
- Allow to communicate with all database supported by odbc driver
- It is vendor independent driver

Disadvantages:

- Decrease the execution speed
- Dependent on the odbc driver
- Require odbc binary code or odbc client library
- Uses java native interface to make odbc call

Type-2 Driver (Native) :

- JDBC call can be converted into the database vendor specific native call with the help of type-2 driver
- Example of type-2 driver is weblogic driver implemented by BEA weblogic
- Its not recommended to use type-2 driver with client side application

Advantage:

- The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.

Disadvantage :

- Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
- Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
- If we change the Database we have to change the native api as it is specific to a database.
- Mostly obsolete now
- Usually not thread safe.

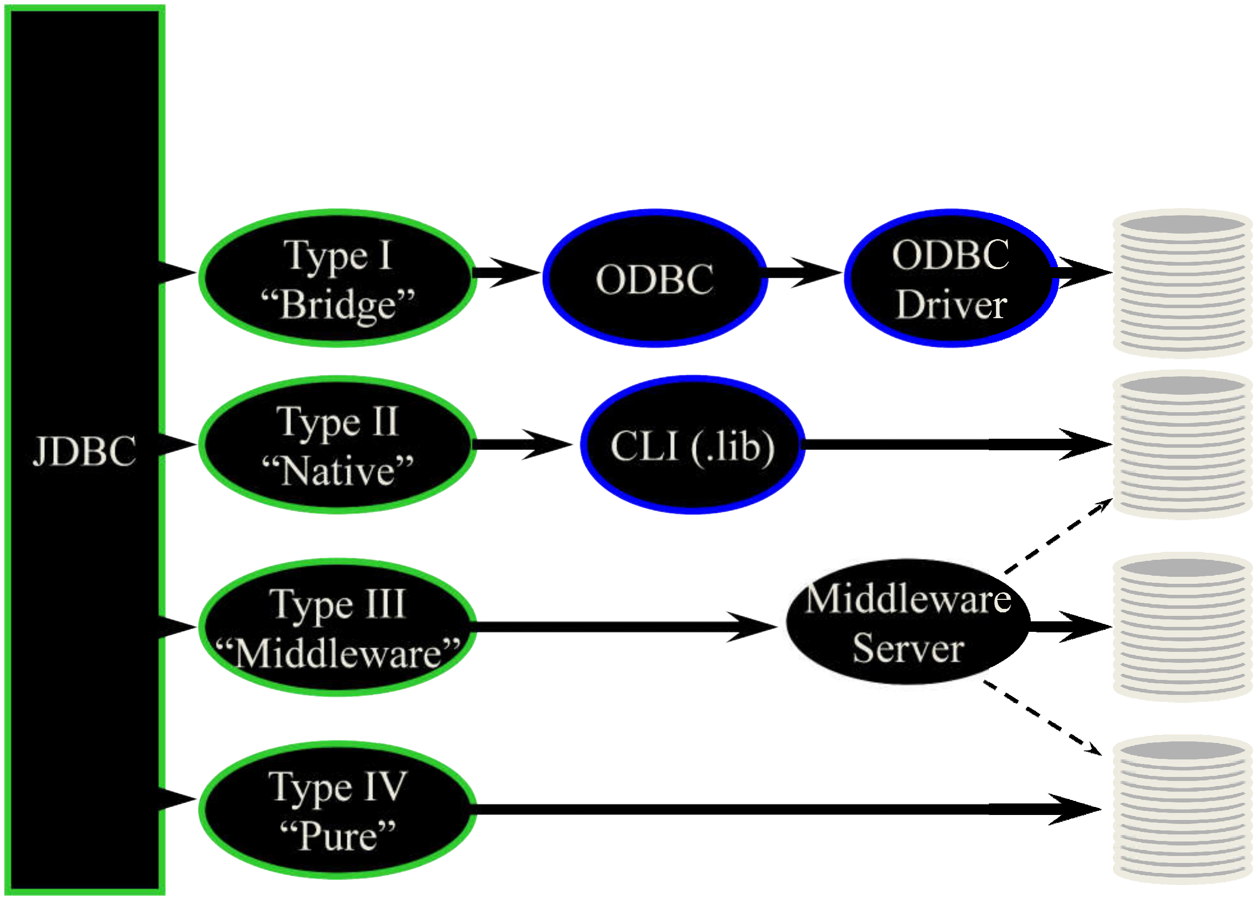


Fig. 2.1 : JDBC Driver Types

Type-3 Driver (Middleware):

- This driver translate the jdbc calls into a database server independent and middleware server specific calls.

- With the help of the middleware server, the translated jdbc calls further translated into database server specific calls.
- This type of driver also known as net-protocol fully java technology-enabled driver.
- Type-3 driver is recommended to be used with applets. its auto-downloadable.

Advantages:

- Serve as all java driver
- Does not require any native library to be installed on client machine
- Ensure database independency
- Does not provide database detail
- Provide the facility to switch over from one database to other

Disadvantage:

- Its perform the task slowly due to increased number of network calls as compared to type-2 driver

Type-4 Driver (Pure Java):

- It is pure java driver
- This type of driver does not require any native database library to retrieve the record the records from database
- This type of driver is lightweight and generally known as thin driver.
- You can use this driver when you want an auto downloadable option the client side application
- i.e. thin driver for oracle from oracle corporation, weblogic and ms sqlserver4 for ms sql server from BEA system

Advantages:

- Serves as pure java driver
- Does not require any native library to install in client machine
- Uses server specific protocol
- Does not require middleware server

Disadvantages:

- Its uses the database specific proprietary protocol

2. Explain the use of Statement, PreparedStatement & CallableStatement with example.

- Once a connection is obtained we can interact with the database. The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.
- They also define methods that help bridge data type differences between Java and SQL data types used in a database.
- Following table provides a summary of each interface's purpose.

Interfaces	**Recommended Use**
Statement	Use 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 when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Table 2.2 : Types of Statements

Statement:

- The Statement interface defines a standard abstraction to execute the SQL statements requested by a user and return the results by using the ResultSet object.
- The Statement interface is created after the connection to the specified database is made.
- The object is created using the createStatement() method of the Connection interface, as shown in following code snippet:

```
Connection con = DriverManager.getConnection(url, “username”, “password”);
Statement stmt = con.createStatement();
```

- **Example of Statement : Select and Display Record From Table**

```
import java.io.*;
import java.sql.*;

class StatementDemo {
    public static void main(String[] args) {
      try {
            // Load and register the driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // Establish the connection to the database server
            Connection cn = DriverManager.getConnection("jdbc:odbc:exm");

            // Create a statement
            Statement st = cn.createStatement();

            // Execute the statement
            ResultSet rs = st.executeQuery("select * from emp”);

            // Retrieve the results
            while(rs.next())
               System.out.println(rs.getInt(1)+" "+rs.getString(2));

            // Close the statement and connection
            st.close();
            cn.close();
           }
           catch(SQLException e) {
            System.out.println(e.getMessage());
           }
      }
  }
```

PreparedStatement:

- The PreparedStatement interface is subclass of the Statement interface, can be used to represent a precompiled query, which can be executed multiple times.
- The object is created using the prepareStatement() method of Connection interface, as shown in following snippet:

```
Connection con = DriverManager.getConnection(url, “username”, “password”);
String query = “insert into emp values(? ,?)”;
PreparedStatement ps = con.prepareStatement(query);
ps.setInt(1,5);
ps.setString(2,”New Employee”);
int n = ps.executeUpdate();
```

Advantages:

- Improves the performance of an application as compared to the Statement object that executes the same query multiple times.
- Inserts or updates the SQL 99 data types columns, such as BLOB, CLOB, or OBJECT, with the help of setXXX() methods.
- Provides the programmatic approach to set the values.

Disadvantages:
The main disadvantage of PreparedStatement is that it can represent only one SQL statement at a time.

- **Example of PreparedStatement : Insert Records into the Table**

```
import java.io.*;
import java.sql.*;

class PreparedStatementDemo {
  public static void main(String[] args) {
   try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    String q = "insert into emp(emp_no,emp_name) values (?,?)";

    Connection con = DriverManager.getConnection("jdbc:odbc:exm");

    PreparedStatement pst = con.prepareStatement(q);

    pst.setString(2,"Employee Name");
    pst.setInt(1,10);
    pst.executeUpdate();

    pst.setString(2,"New Employee1");
    pst.setInt(1,11);
    pst.executeUpdate();

    pst.setString(2,"New Employee2");
    pst.setInt(1,12);
    pst.executeUpdate();

    pst.setString(2,"New Employee3");
    pst.setInt(1,13);
    pst.executeUpdate();

    pst.setString(2,"New Employee4");
    pst.setInt(1,14);
    pst.executeUpdate();
   }
   catch(Exception e) { System.out.println(e); }
  }
}
```

CallableStatement:

- In java the CallableStatement interface is used to call the stored procedure and functions.
- Therefore, the stored procedure can be called by using an object of the CallableStatement interface.
- The object is created using the prepareCall() method of Connection interface, as shown in following snippet:

```
        Connection cn = DriverManager.getConnection("jdbc:odbc:exm");
        CallableStatement cs = cn.prepareCall("{call proc1()}");
        ResultSet rs= cs.executeQuery();
```

- **Example of CallableStatement : Select Records from the Table**

```
import java.io.*;
import java.sql.*;
class CallableStatement {
        public static void main(String[] args)  {
                try {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        Connection cn = DriverManager.getConnection("jdbc:odbc:exm");

                        //call the procedure
                        CallableStatement cs = cn.prepareCall("{call proc1()}");

                        //display record selected  by the procedure
                        ResultSet rs= cs.executeQuery();
                        while(rs.next())
                                System.out.println(rs.getInt(1)+" "+rs.getString(2));

                }
                catch(Exception e) {
                        System.out.println(e.getMessage());
                }
    }
}
```

3. Explain following in brief: JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Manager connection. Some of the important classes and interface defined in JDBC API are as follows.
  - Driver Manager
  - Driver
  - Connection
  - Statement
  - PreparedStatement
  - CallableStatement
  - ResultSet
  - DatabaseMetaData
  - ResultSetMetaData
  - SqlData
  - Blob
  - Clob
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

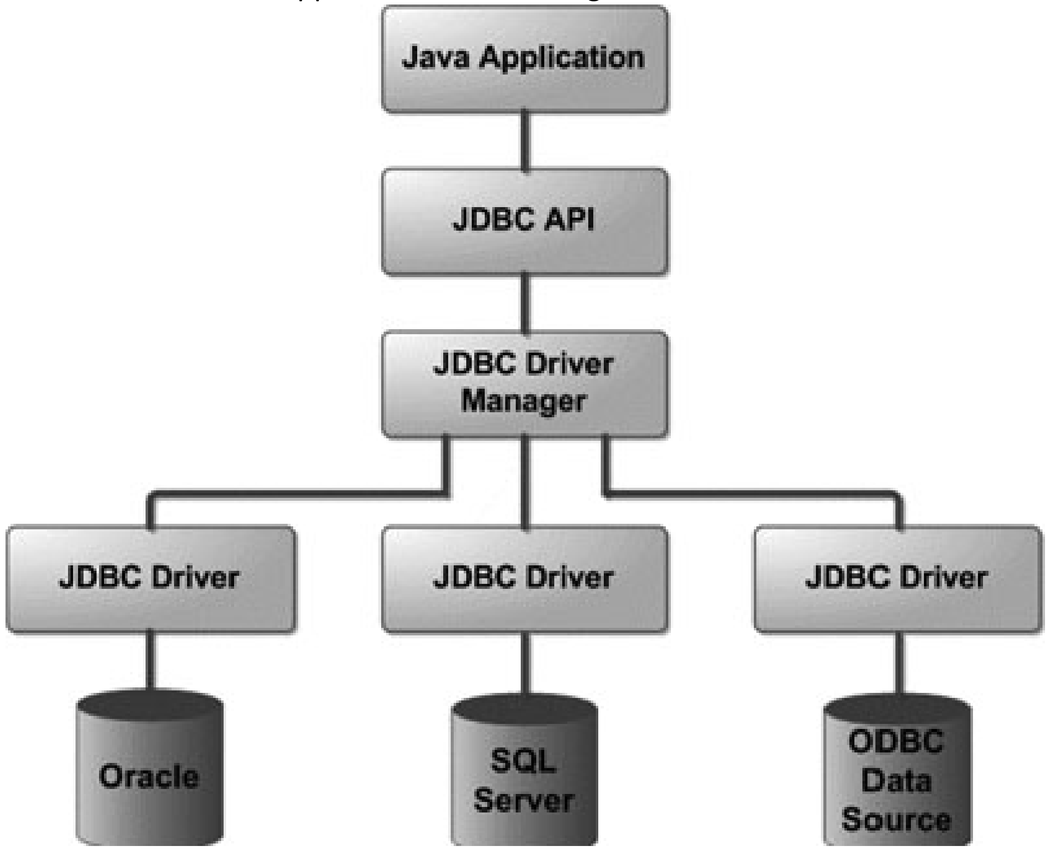


Fig. 2.2 : JDBC Architecture

- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

4. Difference Between executeQuery(), executeUpdate() and execute().

- **executeQuery() : -** executeQuery() can be used to execute selection group sql queries to fetch the data from database.

  public ResultSet executeQuery(String sqlquery)

  When we use selection group sql query with executeQuery() then JVM will send that sql query to the database engine, database engine will execute it, by this database engine(DBE) will fetch the data from database and send back to the java application.

- **executeUpdate() : -** executeUpdate() can be used to execute updation group sql query to update the database.

  public int executeUpdate(String sqlquery)

  When we provide updation group sql query as a parameter to executeUpdate(), then JVM will send that sql query to DBE, here DBE will execute it and perform updations on the database

- **execute() : -** execute() can be used to execute either selection group sql queries or updation group queries.

  public boolean execute(String sqlquery)

  When we use selection group sql query with the execute() then we will get ResultSet object at heap memory with the fetched data. But execute() will return “true” as a Boolean value. When we use updation group sql query with execute() then we will get “ records updated count value” at jdbc application. But execute() will return “false” as a Boolean value.

5. Explain ResultsetMetaData Interface with Example

- The metadata means data about data i.e. we can get further information from the data.
- If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

```
import java.io.*;
import java.sql.*;

class ResultSetMetaData {

        public static void main(String[] args) {

          try {
      // Load and register the driver
           try {
             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
           }
           catch(Exception e) {}

           //Driver myDriver = new sun.jdbc.odbc.JdbcOdbcDriver();
           //DriverManager.registerDriver(myDriver);


           // Establish the connection to the database server
           Connection cn = DriverManager.getConnection("jdbc:odbc:example");
// Create a statement
           Statement st = cn.createStatement();

           // Execute the statement
           ResultSet rs = st.executeQuery("select * from emp");

           // Create Meta Data object
           ResultSetMetaData rsmd = rs.getMetaData();

   // Get Column Details
           for(int i=1;i<=rsmd.getColumnCount();i++)
            System.out.println("Column"+i+"                    "+rsmd.getColumnName(i)+"
"+rsmd.getPrecision(i)+" "+rsmd.getColumnTypeName(i));
```

```
            // Retrieve the results
             while(rs.next())
                System.out.println(rs.getInt("emp_no")+" "+rs.getString(2));

             // Close the statement and connection
             st.close();
             cn.close();
            }
            catch(SQLException e) {
             System.out.println(e.getMessage());
            }
        }
    }
```

1. Explain Servlet Life Cycle

In the life cycle of servlet there are three important methods. These methods are

1. Init()
2. Service()
3. Destroy()

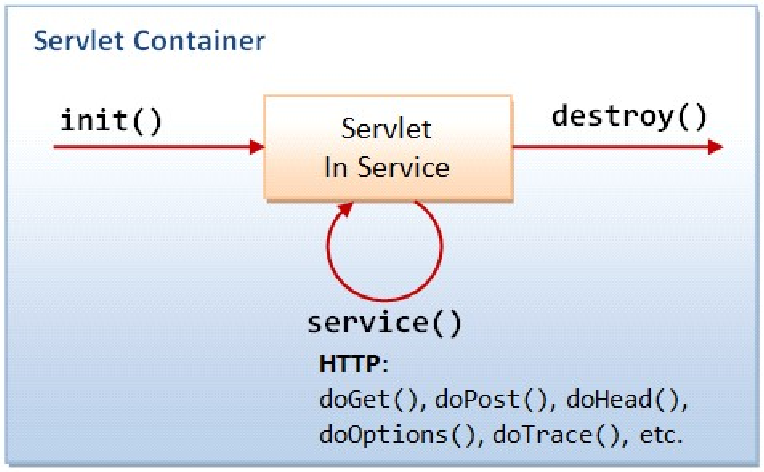


Fig. 1 : Servlet Life Cycle

- The client enters the URL in the web browser and makes a request. The browser then generates the HTTP request and sends it to the Web server.
- Web server maps this request to the corresponding servlet.
- The server basically invokes the **init()** method of servlet. This method is called only when the servlet is loaded in the memory for the first time. Using this method initialization parameters can also be passed to the servlet in order to configure itself.
- Server can invoke the service for particular HTTP request using **service()** method. The servlets can then read the data provided by the HTTP request with the help of **service()** method.
- Finally server unloads the servlet from the memory using the **destroy()** method.

Following is a simple servlet in which these three methods viz. init(), service() and destroy() are used.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class LifeCycle extends GenericServlet {
        public void init(ServletConfig config) throws ServletException {
                System.out.println("init");
        }
        public void service(ServletRequest request, ServletResponse response) throws
        ServletException, IOException {
                System.out.println("from service");
                PrintWriter out = response.getWriter();
                out.println("welcome to servlet");
                out.println("This is my first servlet");
        }

        public void destroy() {
                 System.out.println("destroy");
        }
}
```

ServletContext:

- ServletContext object is used to get configuration information from Deployment Descriptor(web.xml) which will be available to any Servlet or JSPs that are part of the web application.
- There is one context per "web application" per Java Virtual Machine.
- The ServletContext object is contained within the ServletConfig object. That is, the ServletContext can be accessed using the ServletConfig object within a servlet.
- You can specify param-value pairs for ServletContext object in <context-param> tags in web.xml file.
- Example:

```
<context-param>
<param-name>Parameter Name</param-name>
<param-value>Parameter Value</param-value>
</context-param>
```

- Diffrence between ServletContext & ServletConfig.

Servlet Config	**Servlet Context**
Servlet config object represent single servlet	It represent whole web application running on particular JVM and common for all the servlet
Its like local parameter associated with particular servlet	Its like global parameter associated with whole application
It's a name value pair defined inside the servlet section of web.xml file so it has servlet wide scope	ServletContext has application wide scope so define outside of servlet tag in web.xml file.

getServletConfig() method is used to get the config object	getServletContext() method is used to get the context object.
for example shopping cart of a user is a specific to particular user so here we can use servlet config	To get the MIME type of a file or application session related information is stored using servlet context object.

2. Explain Request Dispatcher in Java Servlet with example.

- The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration.
- There are two methods defined in the RequestDispatcher interface.

1. **public void forward(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:** Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. **public void include(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:** Includes the content of a resource (servlet, JSP page, or HTML file) in the response.

- **Example**

Index.html

```
<form method="post" action="http://localhost:8080/Validate">
Name:<input type="text" name="user" /><br/>
Password:<input type="password" name="pass" ><br/>
<input type="submit" value="submit">
</form>
```

Validate.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Validate extends HttpServlet {
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
     response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    try {
      String name = request.getParameter("user");
      String password = request.getParameter("pass");

       if(password.equals("darshaninst"))
       {
          RequestDispatcher rd = request.getRequestDispatcher("Welcome");
          rd.forward(request, response);
       }
       else
        {
        out.println("<font color='red'><b>You have entered incorrect password</b></font>");
           }
       }finally {
           out.close();
       }

    }
}
```

Welcome.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Welcome extends HttpServlet {

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       try {

           out.println("<h2>Welcome user</h2>");
       } finally {
           out.close();
       }
   }
}
```

web.xml

```
<web-app>
    <servlet>
        <servlet-name>Validate</servlet-name>
        <servlet-class>Validate</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Validate</servlet-name>
        <url-pattern>/Validate</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>Welcome</servlet-name>
        <servlet-class>Welcome</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Welcome</servlet-name>
        <url-pattern>/Welcome</url-pattern>
    </servlet-mapping>

</web-app>
```

3 What is filter? What is its use? List the different filter interfaces with their important methods.

- A **filter** is an object that is invoked at the preprocessing and postprocessing of a request.
- It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc.
- The **servlet filter is pluggable**, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the servlet.
- So maintenance cost will be less.

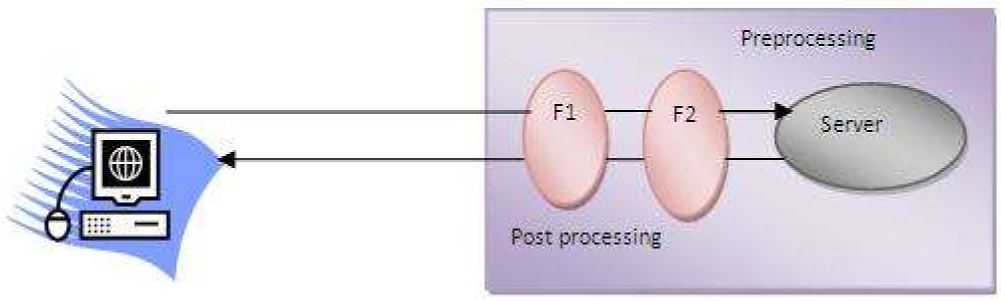


Fig 1 : Filter

Usage of Filter

- recording all incoming requests
- logs the IP addresses of the computers from which the requests originate
- conversion
- data compression
- encryption and decryption
- input validation etc.

Advantage of Filter

- Filter is pluggable.
- One filter don't have dependency onto another resource.
- Less Maintenance

Filter API :

Like servlet filter have its own API. The javax.servlet package contains the three interfaces of Filter API.

1. Filter
2. FilterChain
3. FilterConfig

1) Filter interface

For creating any filter, you must implement the Filter interface. Filter interface provides the life cycle methods for a filter.

Method	Description
public void init(FilterConfig config)	init() method is invoked only once. It is used to initialize the filter.
public void doFilter(HttpServletRequest request,HttpServletResponse response, FilterChain chain)	doFilter() method is invoked every time when user request to any resource, to which the filter is mapped.It is used to perform filtering tasks.
public void destroy()	This is invoked only once when filter is taken out of the service.

2) FilterChain interface

The object of FilterChain is responsible to invoke the next filter or resource in the chain.This object is passed in the doFilter method of Filter interface.The FilterChain interface contains only one method:

1. **public void doFilter(HttpServletRequest request, HttpServletResponse response):** it passes the control to the next filter or resource.

Example

MyFilter.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;

public class MyFilter implements Filter{
    public void init(FilterConfig arg0) throws ServletException {}
    public void doFilter(ServletRequest req, ServletResponse resp,  FilterChain chain)
        throws IOException, ServletException {
                PrintWriter out=resp.getWriter();
                out.print("filter is invoked before");
                chain.doFilter(req, resp);//sends request to next resource
                out.print("filter is invoked after");
        }
        public void destroy() {}
}
```

HelloServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

     response.setContentType("text/html");
     PrintWriter out = response.getWriter();

     out.print("<br>welcome to servlet<br>");

   }
}
```

Web.xml

```
<web-app>
    <servlet>
        <servlet-name>s1</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>s1</servlet-name>
        <url-pattern>/servlet1</url-pattern>
    </servlet-mapping>
```

```
<filter>
    <filter-name>f1</filter-name>
    <filter-class>MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>f1</filter-name>
    <url-pattern>/servlet1</url-pattern>
</filter-mapping>
</web-app>
```

4 Explain Request and Response object in servlet.

- True job of a Servlet is to handle client request.
- Servlet API provides two important interfaces **javax.servlet.ServletRequest** and **javax.servlet.ServletResponse** to encapsulate client request and response.
- Implementation of these interfaces provide important information about client request to a servlet.

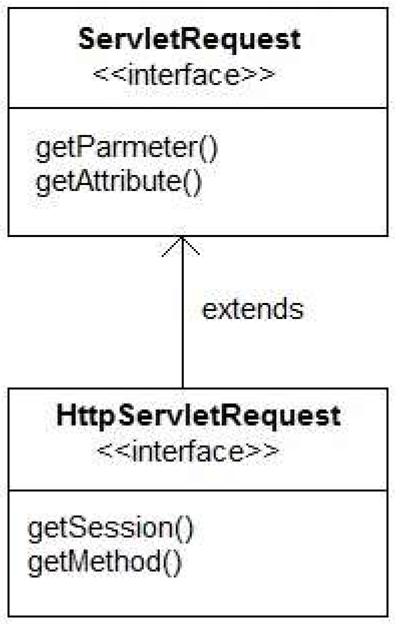


Fig : Servlet Request

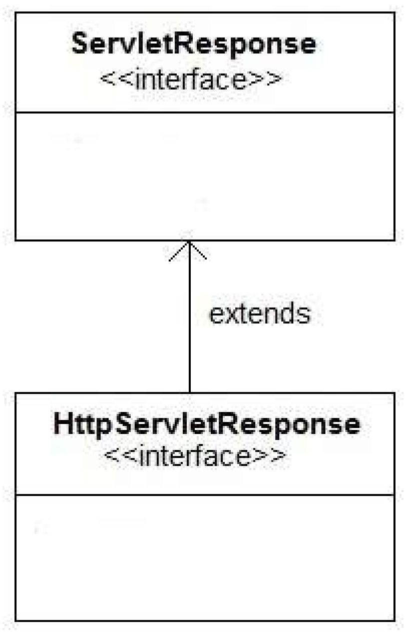


Fig : Servlet Response

Index.html

```
<form method="post" action="check">
        Name <input type="text" name="user" >
        <input type="submit" value="submit">
</form>
```

Web.Xml

```
<servlet>
     <servlet-name>check</servlet-name>
     <servlet-class>MyServlet</servlet-class>
</servlet>
<servlet-mapping>
     <servlet-name>check</servlet-name>
     <url-pattern>/check</url-pattern>
</servlet-mapping>
```

MyServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
```

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     try {

        String user=request.getParameter("user");
        out.println("<h2> Welcome "+user+"</h2>");
     } finally {
        out.close();
     }
  }
}
```

5 Session in Servlet, Explain with Example.

- HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.
- Still there are following three ways to maintain session between web client and web server:
- **Cookies:**
    - A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the recieved cookie.
    - This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.
- **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">
    - This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. 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.

Session Tracking Example:

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// Extend HttpServlet class
public class SessionTrack extends HttpServlet {

  public void doGet(HttpServletRequest request,
                HttpServletResponse response)
        throws ServletException, IOException
  {
     // Create a session object if it is already not  created.
     HttpSession session = request.getSession(true);
     // Get session creation time.
     Date createTime = new Date(session.getCreationTime());
     // Get last access time of this web page.
     Date lastAccessTime =
                  new Date(session.getLastAccessedTime());

     String title = "Welcome Back to my website";
     Integer visitCount = new Integer(0);
     String visitCountKey = new String("visitCount");
     String userIDKey = new String("userID");
     String userID = new String("ABCD");

     // Check if this is new comer on your web page.
     if (session.isNew()){
        title = "Welcome to my website";
        session.setAttribute(userIDKey, userID);
     } else {
        visitCount = (Integer)session.getAttribute(visitCountKey);
        visitCount = visitCount + 1;
        userID = (String)session.getAttribute(userIDKey);
     }
     session.setAttribute(visitCountKey,  visitCount);
```

```
      // Set response content type
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
              "<html>\n" +
              "<head><title>" + title + "</title></head>\n" +
              "<body bgcolor=\"#f0f0f0\">\n" +
              "<h1 align=\"center\">" + title + "</h1>\n" +
               "<h2 align=\"center\">Session Infomation</h2>\n" +
              "<table border=\"1\" align=\"center\">\n" +
              "<tr bgcolor=\"#949494\">\n" +
              "  <th>Session info</th><th>value</th></tr>\n" +
              "<tr>\n" +
              "  <td>id</td>\n" +
              "  <td>" + session.getId() + "</td></tr>\n" +
              "<tr>\n" +
              "  <td>Creation Time</td>\n" +
              "  <td>" + createTime +
              "  </td></tr>\n" +
              "<tr>\n" +
              "  <td>Time of Last Access</td>\n" +
              "  <td>" + lastAccessTime +
              "  </td></tr>\n" +
              "<tr>\n" +
              "  <td>User ID</td>\n" +
              "  <td>" + userID +
              "  </td></tr>\n" +
              "<tr>\n" +
              "  <td>Number of visits</td>\n" +
              "  <td>" + visitCount + "</td></tr>\n" +
              "</table>\n" +
              "</body></html>");
  }
}
```

1. Explain JSP unified Expression Language (EL) .

- EL is a language that allows JSP programmers to fetch application data stored in JavaBeans component.
- The following methods are used to call the java code from a JSP page:
    1. Placing the entire Java code in a JSP page
    2. Defining separate helper classes that encapsulate the entire Java code and calling these helper classes from a JSP page
    3. Using JavaBeans and JSP page action tags
    4. Using JSP EL
    5. Create tag handler classes to embed the Java code
- The incorporation of EL to the JSP has helped reduce the use of scriptles in JSP page.
- EL expressions provide short hand notations to retrieve, present, and manipulate Web application data.
- EL expressions are enclosed between the ${ and} characters.
- The most general example of an EL expression is ${object.data}

  Where, object can be any Java object representing different scope, such as request,session.
- EL expressiongs can be categorized into the following types:
    1. Immediate and deferred expressions
    2. Value expressions
    3. Method expressions

1) Immediate and deferred expressions
    - Two constructs are used to represent EL expressions,
        - ${ expr } used for expressions that need to be evaluated immediately, and
        - #{ expr } used for expressions that need to be evaluated at later time.
    - An expression that uses ${ expr } is called immediate expression and
    - The expression that uses #{ expr } is called deferred expression.
2) Value Expressions
    - Value epressions are used to refer to objects such as JavaBeans, Collections, enumerations and implicit objects and their properties.
    - An object is referred to by using the value expression containing the name of the object
    - Suppose the ${ employee } expression is used in a JSP page, where employee refers to the name of a JavaBean.
    - Which searches for the employee JavaBeab in the request, session and application scopes
    - If the employee JavaBean does not exist, a null value is returned.
    - Syntax of value expression to access properties and elements of a collection in java uses two operators, (.) and ([]).
    - For example,

      The name property of the employee JavaBean can be referred as ${employee.name} or ${ employee["name"] } expression.

To access a specific element of a collection, such as list or an array inside a JavaBean we can use ${ employee.name[0] }, ${ employee.name[1] }, etc…

- The value expressions are of two types:
  - o rvalue

    rvalue expression can only read data, and not write data.
  - o lvalue

    lvalue expression can read as well as write data.

3) Method Expressions

- Method expressions are used to call public methods, which returns a value or object.
- Such expressions are usually deferred expressions.
- Tags usually use method expressions to call functions that perform operations such as validating a UI component or handling the events generated on a UI component.
- The following code snippet shows the use of method expressions in JSF page:

```
<h:form>
    <h:inputText
                    id=”email”
                    value=”#{employee.email}
                    validator=”#{employee.validateEmail}”  />
    <h:commandButton
                    id=”submit”
                    action=”#{customer.submit}” />
</h:form>
```

- The various elements shown in the preceding code snippet can be briefly described as follows:
  - o The **inputText** tag : shows the UIInput component in the form of a text field on a webpage
  - o The **validator** attribute : calls the validateEmail method of the employee JavaBean
  - o The **action** attribute of the **commandButton** tag : calls the submit method, which carries out processing after submitting the webpage
  - o The **validateEmail** method : Refers to the method that is called during the validation phase of the JSP life cycle
  - o The **submit** method : Refers to the method that is called during the invoke application phase of the JSP life cycle

2 Enlist and explain the use of action tags in JSP.

- Action tags are specific to a JSP page. When a JSP container encounters an action tag while translating a JSP page into a servlet, it generates a Java code corresponding to the task to be performed by the action tag.
- The following are some important action tags available in JSP page:
  1. <jsp:include>
     - The <jsp:include> action tag facilitates Java programmers in including a static or dynamic resource, such as an HTML or JSP page in the current JSP page.
     - If the resource to be included is static then its content is directly included.
     - If the resource is dynamic then first it is processed and result is included in page.
     - Example
       <jsp:include page=”test.jsp” />
  2. <jsp:forward>
     - The <jsp:forward> tag forwards a JSP request to another resource which can be either static or dynamic.
     - If the request is forwarded to a dynamic resource, a <jsp:param> tag can be used to pass a name and value of a parameter to a resource.
     - Example:
       <jsp:forward page=”/header.html” />
  3. <jsp:param>
     - The <jsp:param> tag allows Java programmers to pass a name and value of a parameter to a dynamic resource, while including it in a JSP page or forwarding a request to another JSP page.
     - Example:
       <jsp:param name=”uname” value=”PM”/>
  4. <jsp:useBean>
     - To separate the business logic from the presentation logic, it is often a good idea to encapsulate the business login in Java object, and then instantiate and use this Java object within a JSP page, <jsp:useBean> tag helps in such task.
     - The <jsp:useBean> action tag has certain attributes that add extra characteristics to it, some attributes specific to <jsp:useBean> are:
       - **Id** : represents the name assigned to a JavaBean, which is later used as a variable to access the JavaBean
       - **Class** : takes the qualified class name to create a JavaBean instance if the JavaBean instance is not found in the given scope.
       - **beanName** :
         - takes the qualified class name or expression of a JavaBean to create a JavaBean.
         - Class & beanName can not be used together.
         - We can not use expressions to create instance of JavaBean from class attribute, for that we have to use beanName attribute.

- **Scope** : represents the scope in which a JavaBean instance has to be created.
  - **Page scope** : indicates that a JavaBean can be used where <jsp:useBean> action tag is used.
  - **Request scope** : indicates that a JavaBean can be used from any JSP page, which processes the same request until a response is sent to a client by the JSP page.
  - **Session scope** : indicates that a JavaBean can be used from any JSP page invoked in the same session as the JSP page that created the JavaBean.
  - **Application scope** : indicates that a JavaBean can be used from any JSP page in the same application as the JSP page that created the JavaBean.
- Example :

```
<jsp:useBean id=”myBean” class=”MyBeanClass” scope=”session”>
        …………
</jsp:useBean>
```

5. <jsp:setProperty>
   - The <jsp:setProperty> action tag sets the value of a property by using the setter method of a JavaBean.
   - Before using the <jsp:setProperty> action tag, the JavaBean must be instantiated.
   - Example:

```
<jsp:setProperty name=”beanName” property=”uName” value=”PM” />
```

6. <jsp:getProperty>
   - The <jsp:setProperty> action tag retrives the value of a property by using the getter method of a JavaBean and writes the value of the current JspWriter.
   - Example:

```
<jsp:getProperty name=”myBean” property=”uName” />
```

7. <jsp:plugin>
   - The <jsp:plugin> action tag provides support for including a Java applet or JavaBean in a client Web browser, by using built-in or downloaded Java plug-in.
   - The <jsp:plugin> action tag perform the following operations:
     - Specify whether the component added in the <object> tag is a JavaBean or an applet.
     - Locate the code that needs to be run.
     - Position an object in the browser window.
     - Specify a URL from which the plug-in software is to be downloaded.
     - Pass parameter names and values to an object
   - Example:

```
<jsp:plugin attributes>
    ………
</jsp:plugin>
```

8. <jsp:params>
   - The <jsp:params> action tag sends the parameters that you want to pass to an

applet or JavaBean.

- Example:

```
<jsp:params>
        <!- one or more jsp:param tags -->
</jsp:params>
```

9. <jsp:fallback>
   - The <jsp:fallback> action tag is allows you to specify a text message that is displayed if the required plug-in cannot run.
   - This tag must be used as a child tag with <jsp:plugin> action tag.
   - Example:

```
<jsp:fallback>
        Text message that has to be displayed if the plugin cannot be started
</jsp:fallback>
```

10. <jsp:attribute>
    - The <jsp:attribute> action tag is used to specify the value of a standard or custom action attribute.
    - Example:

```
<jsp:setProperty name=”myBean”>
        <jsp:attribute name=”property”>uName</jsp:attribute>
        <jsp:attribute name=”value”>PM</jsp:attribute>
</jsp:setProperty>
```

11. <jsp:body>
    - The <jsp:body> action tag is used to specify the content (or body) of a standard or custom action tag.
    - Example:

```
<jsp:userBean id=”myBean”>
        ……..
        <jsp:body>  …………….   </jsp:body>
</jsp:myBean>
```

12. <jsp:element>
    - The <jsp:element> action tag is used to dynamically define the value of the tag of an XML element.
    - Example:

```
<jsp:element name=”mytag”>
        <jsp:attribute name=”myatt”>My Value </jsp:attribute>
</jsp:element>
```

13. <jsp:text>
    - A <jsp:text> tag is used to enclose template data in an XML tag.
    - The content of the <jsp:text> body is passed to the implicit object *out*.
    - Example:

- <jsp:text> Hello world from the action tag </jsp:text>

3 Explain JSP standard tag library (JSTL) with examples.

- Initially, web designers used scriplets in JSP pages to generate dynamic content.
- This resulted in readability issues and also made it difficult to maintain the JSP page.
- Custom tags were introduced to overcome the problems faced in using scriplets, they had some limitations too.
- Web designers had to spend a lot of time in coding, packaging, and testing these tags before using them.
- This meant that web designers were often left with little time to concentrate on the designing of web pages.
- The introduction of JSTL has helped web designers overcome the shortcomings of custom tags, by encapsulating the common functionalities included the use of tag libraries, such as core, SQL, and XML.
- The main features of JSTL are:
    - Provides support for conditional processing and Uniform Resource Locator (URL)-related actions.
    - Provides the XML tag library, which helps you to manipulate XML documents.
    - Enables Web applications to be accessed globally by providing the internationalization tag library.
    - Enables interaction with relational databases by using various SQL commands.
    - Provides series of functions to perform manipulations.
- Tag Libraries in JSTL:

Tag Library	Function	URI	prefix
Core tag library	Variable support Flow Control Iterator URL management Miscellaneous	http://java.sun.com/jsp/jstl/core	c
XML tag library	Flow control Transformation Locale	http://java.sun.com/jsp/jstl/xml	x
Internationalization tag library	Message formatting Number and date formatting	http://java.sun.com/jsp/jstl/fmt	fmt
SQL tag library	Database manipulation	http://java.sun.com/jsp/jstl/sql	sql

Functions Library	Collection length String manipulation	http://java.sun.com/jsp/jstl/functions	fn

- Example of Core tag library:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title><c:choose> Tag Example</title>
</head>
<body>
<c:set var="salary" scope="session" value="${2000*2}"/>
<p>Your salary is : <c:out value="${salary}"/></p>
<c:choose>
   <c:when test="${salary <= 0}">
     Salary is very low to survive.
   </c:when>
   <c:when test="${salary > 1000}">
      Salary is very good.
   </c:when>
   <c:otherwise>
      No comment...
   </c:otherwise>
</c:choose>
</body>
</html>
```

4 What is XML tag library? Explain the XML core tags and show their use.

- The JSTL XML tags provide a JSP-centric way of creating and manipulating XML documents. Following is the syntax to include JSTL XML library in your JSP.
- The JSTL XML tag library has custom tags for interacting with XML data. This includes parsing XML, transforming XML data, and flow control based on XPath expressions.

```
<%@ taglib prefix="x"
        uri="http://java.sun.com/jsp/jstl/xml" %>
```

- Before you proceed with the examples, you would need to copy following two XML and XPath related libraries into your <Tomcat Installation Directory>\lib:

- **XercesImpl.jar:** Download it from http://www.apache.org/dist/xerces/j/
- **xalan.jar:** Download it from http://xml.apache.org/xalan-j/index.html

Following is the list of XML JSTL Tags:

Tag	Description
<x:out>	Like <%= ... >, but for XPath expressions.
<x:parse>	Use to parse XML data specified either via an attribute or in the tag body.
<x:set >	Sets a variable to the value of an XPath expression.
<x:if >	Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
<x:forEach>	To loop over nodes in an XML document.
<x:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<x:when >	Subtag of <choose> that includes its body if its expression evalutes to 'true'
<x:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'
<x:transform >	Applies an XSL transformation on a XML document
<x:param >	Use along with the transform tag to set a parameter in the XSLT stylesheet

5 Explain SQL Tab Libraries.

- The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, mySQL, or Microsoft SQL Server.
- Following is the syntax to include JSTL SQL library in your JSP:

```
<%@ taglib prefix="sql"
      uri="http://java.sun.com/jsp/jstl/sql" %>
```

Following is the list of SQL JSTL Tags:

Tag	Description
<sql:setDataSource>	Creates a simple DataSource suitable only for prototyping
<sql:query>	Executes the SQL query defined in its body or through the sql attribute.
<sql:update>	Executes the SQL update defined in its body or through the sql attribute.
<sql:param>	Sets a parameter in an SQL statement to the specified value.
<sql:dateParam>	Sets a parameter in an SQL statement to the specified java.util.Date
<sql:transaction >	Provides nested database action elements with a shared Connection.

6 Explain Core JSTL Tags

- The core group of tags are the most frequently used JSTL tags. Following is the syntax to include JSTL Core library in your JSP:
```
<%@ taglib prefix="c"
        uri="http://java.sun.com/jsp/jstl/core" %>
```
- There are following Core JSTL Tags:

Tag	Description
<c:out >	Like <%= ... >, but for expressions.
<c:set >	Sets the result of an expression evaluation in a 'scope'
<c:remove >	Removes a scoped variable (from a particular scope, if specified).
<c:catch>	Catches any Throwable that occurs in its body and optionally exposes it.
<c:if>	Simple conditional tag which evalutes its body if the supplied condition is true.
<c:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<c:when>	Subtag of <choose> that includes its body if its condition evalutes to 'true'.
<c:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'.
<c:import>	Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'.
<c:forEach >	The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality .
<c:forTokens>	Iterates over tokens, separated by the supplied delimeters.
<c:param>	Adds a parameter to a containing 'import' tag's URL.
<c:redirect >	Redirects to a new URL.
<c:url>	Creates a URL with optional query parameters

7 Explain Session Handling with example.

- HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.
- Still there are following three ways to maintain session between web client and web server:

Cookies:

- A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.
- This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

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">
- This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. 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://tutorialspoint.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.

Session Tracking Example :

Index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

Welcome.jsp

```
<html>
<body>
<%

String name=request.getParameter("uname");
out.print("Welcome "+name);

session.setAttribute("user",name);

<a href="second.jsp">second jsp page</a>

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

Second.jsp

```
<html>
<body>
<%
String name=(String)session.getAttribute("user");
out.print("Hello "+name);
%>
</body>
</html>
```

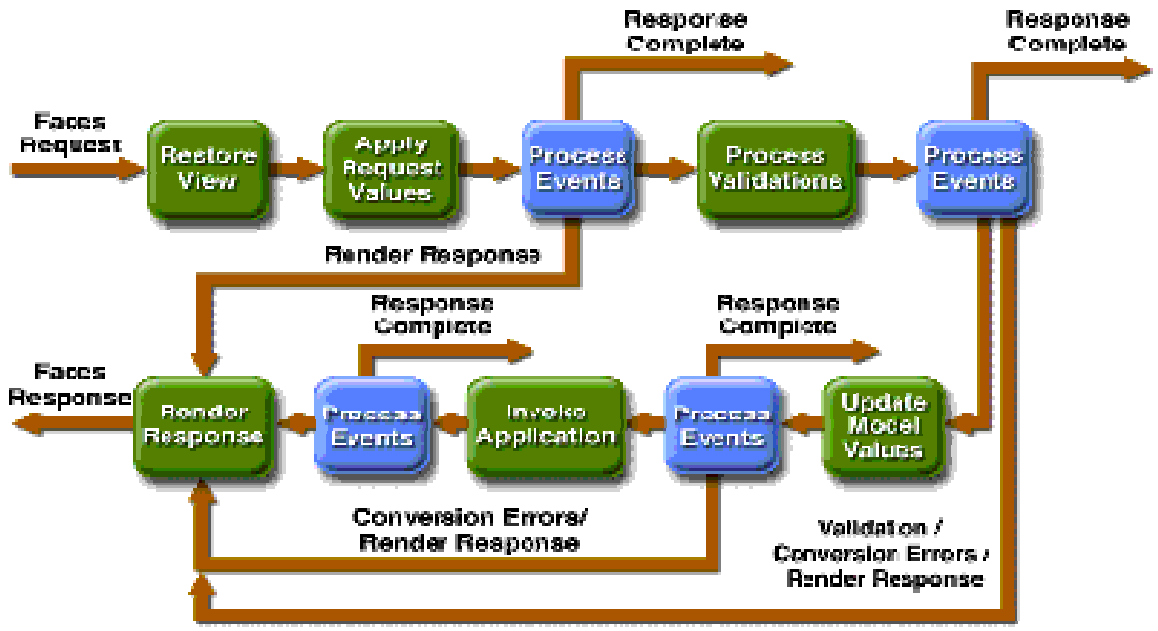
8 Write a program to insert records in a table using JSP.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Example of Java Server Page with JDBC</title>
   </head>
   <body>
<%
```

```
String u=request.getParameter("userid");
String p=request.getParameter("password");
String n=request.getParameter("sname");
String e=request.getParameter("eid");
String a=request.getParameter("addr");
try
{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con=DriverManager.getConnection("jdbc:odbc:sgc","","");
/* Passing argument through the question mark */
            PreparedStatement ps=con.prepareStatement("insert into login values(?,?,?,?,?)") ;
            ps.setString(2,p);
            ps.setString(3,n);
            ps.setString(4,a);
            ps.setString(5,e);
            int i=ps.executeUpdate(); /*Set the Update query command */
            if(i!=0)
            {
                        response.sendRedirect("index.jsp?msg=Thank You for registering with us in
Mrbool !");
             }
            else
             {
                        response.sendRedirect("registerinsert.jsp?msg=Insertion Failed!! Please try
again!!!   ");
            }
      con.close();
   }
   catch(Exception ex)
   {
      out.println(ex);
    }
%>
  </body>
</html>
```

1. Explain JSF Request Processing Life Cycle.

- JSF application lifecycle consist of six phases which are as follows :
  - Restore view phase
  - Apply request values phase; process events
  - Process validations phase; process events
  - Update model values phase; process events
  - Invoke application phase; process events
  - Render response phase



- The six phases show the order in which JSF processes a form. The list shows the phases in their likely order of execution with event processing at each phase.

Phase 1: Restore view

- JSF begins the restore view phase as soon as a link or a button is clicked and JSF receives a request.
- During this phase, the JSF builds the view, wires event handlers and validators to UI components and saves the view in the FacesContext instance. The FacesContext instance will now contains all the information required to process a request.

Phase 2: Apply request values

- After the component tree is created/restored, each component in component tree uses decode method to extract its new value from the request parameters. Component stores this value. If the conversion fails, an error message is generated and queued on FacesContext. This message will be displayed during the render response phase, along with any validation errors.
- If any decode methods / event listeners called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.

Phase 3: Process validation

- During this phase, the JSF processes all validators registered on component tree. It examines the component attribute rules for the validation and compares these rules to the local value stored for the component.
- If the local value is invalid, the JSF adds an error message to the FacesContext instance, and the life cycle advances to the render response phase and display the same page again with the error message.

Phase 4: Update model values

- After the JSF checks that the data is valid, it walks over the component tree and set the corresponding server-side object properties to the components' local values. The JSF will update the bean properties corresponding to input component's value attribute.
- If any updateModels methods called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.

Phase 5: Invoke application

- During this phase, the JSF handles any application-level events, such as submitting a form / linking to another page.

Phase 6: Render response

- During this phase, the JSF asks container/application server to render the page if the application is using JSP pages. For initial request, the components represented on the page will be added to the component tree as the JSP container executes the page. If this is not an initial request, the component tree is already built so components need not to be added again. In either case, the components will render themselves as the JSP container/Application server traverses the tags in the page.
- After the content of the view is rendered, the response state is saved so that subsequent requests can access it and it is available to the restore view phase.

2. Explain JSF Expression Language with Example.

- JSF provides a rich expression language. We can write normal operations using #{operation-expression} notation. Some of the advantages of JSF Expression languages are following.
  - Provides easy access to elements of a collection which can be a list, map or an array.
  - Provides easy access to predefined objects such as request.
  - Arithmetic, logical, relational operations can be done using expression language.
  - Automatic type conversion.
  - Shows missing values as empty strings instead of NullPointerException.
- Let us understand it with example :

UserData.java

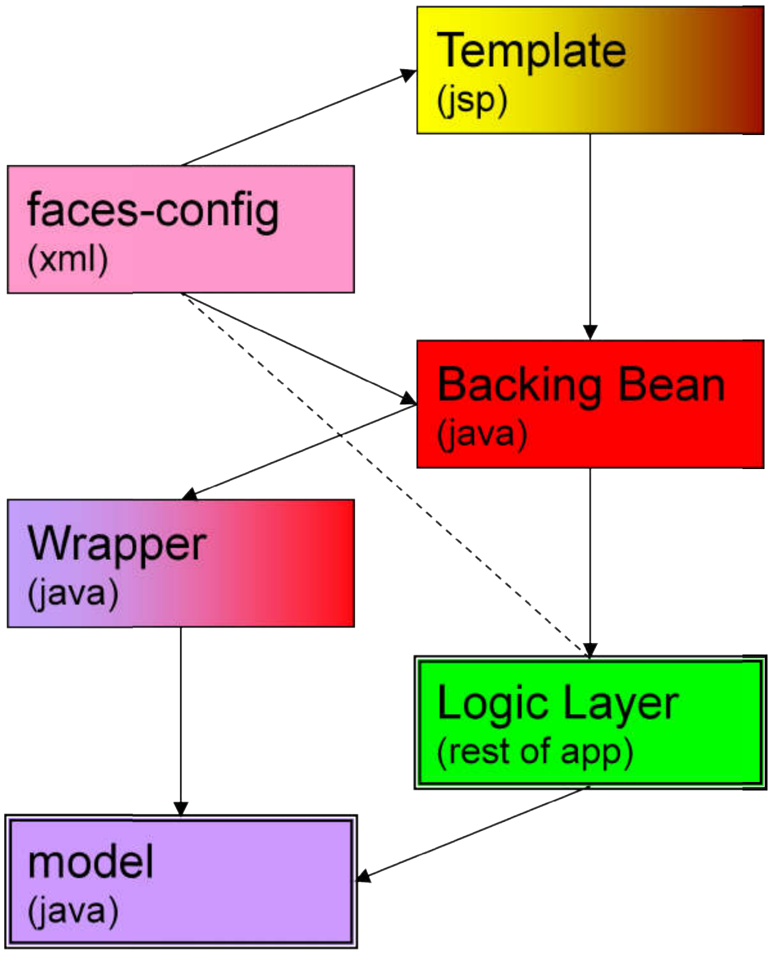
```
import java.io.Serializable;
import java.util.Date;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "userData", eager = true)
@SessionScoped
public class UserData implements Serializable {

private static final long serialVersionUID = 1L;

  private Date createTime = new Date();
  private String message = "Hello World!";

  public Date getCreateTime() {
    return(createTime);
  }
  public String getMessage() {
    return(message);
  }
}
```

home.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>JSF Tutorial!</title>
  </h:head>
  <h2>Expression Language Example</h2>
  Creation time:
  <h:outputText value="#{userData.createTime}"/>
  <br/><br/>Message:
  <h:outputText value="#{userData.message}"/>
  </h:body>
</html>
```

3. Explain JSF Structure/Architecture.



- Similar to most of the popular Web Application Frameworks, JSF implements MVC design pattern. The implementation of MVC design pattern helps to design different components separately, and each of these components implements different types of logic.
- For example, you can crate view component for presentation logic and model component for business logic implementation. A controller is another component, which controls the execution of various model methods and navigates from one view to another in a Web Application.
- In JSF, you have a single front controller, which handles all JSF requests mapped to it.
- The controller part of the JSF architecture consists of a Controller Servlet, that is, FacesServlet, a centralized configuration file, faces-config.xml, and a set of event handlers for the Web Application.
- The Model in JSF architecture is a set of server-side JavaBeans that retrieves the values from the model components such as the database and defines methods on the basis of these values.
- The UI Components are rendered in different ways according to the type of the client. The view delegates this task to separate the renderers , each taking care of one specific output type, such as HTML.
- JSF Structure/Architecture is consists of following components:
    - The template (most commonly jsp) defines the interface
    - The faces-config defines the navigation and the backing beans
    - Backing beans handle action processing, navigation processing, and connections to the logic (business) layer
    - Wrapper bean wraps the data POJOs for JSF handling
    - Logic layer beans can be injected as defined in the faces-config
    - Model is basic data POJO (Plain Old Java Object)

JSP Template

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<f:view>
    <html><head><title>Items</title></head><body>
    <h:form id="items">
    <h:dataTable id="itemlist" value="#{JsfBean.allItems}" var="entry">
    <h:column>
```

```
        <f:facet name="header">
                <h:outputText value=""/>
        </f:facet>
        <h:selectBooleanCheckbox id="itemSelect" value="#{entry.selected}"
        rendered="#{entry.canDelete}"/>
        <h:outputText value="" rendered="#{not entry.canDelete}"/>
        </h:column>
        </h:form>
        </body></html>
    </f:view>
```

faces-config.xml

- Defines the backing beans
    - Syntax not like Spring
    - Name used in EL in template
    - Scope controlled (request, session, etc.)
- Defines navigation rules
    - Based on views
        - Where from (view)
        - Which outcome (id)
        - Where to go (view)
- Can match outcomes using wildcards

JSF backing beans

- Typical bean with getters and setters and additional methods to handle actions
    - Store data needed for processing the user actions using setters
    - Retrieve data using getters and methods
    - Process actions using methods
- Often includes code to wrap data objects
- Connects to the rest of the application
    - Typically via injection

JSF wrapper bean

- Wraps basic data POJO
- Required to avoid putting UI information in the data POJO
- Often includes basic setters and getters for the UI related properties
  - Can be methods as well
- Also includes a setter and getter for the data POJO

4. JSF Standard Components

- Following are important *Basic Tags* in JSF 2.0:
- h:inputText : Renders a HTML input of type="text", text box.
- h:inputSecret : Renders a HTML input of type="password", text box.
- h:inputTextarea : Renders a HTML textarea field.
- h:inputHidden : Renders a HTML input of type="hidden".
- h:selectBooleanCheckbox : Renders a single HTML check box.
- h:selectManyCheckbox : Renders a group of HTML check boxes.
- h:selectOneRadio : Renders a single HTML radio button.
- h:selectOneListbox : Renders a HTML single list box.
- h:selectManyListbox : Renders a HTML multiple list box.
- h:selectOneMenu : Renders a HTML combo box.
- h:outputText : Renders a HTML text.

5. JSF Facelet Tags

- JSF provides special tags to create common layout for a web application called facelets tags. These tags gives flexibility to manage common parts of a multiple pages at one place.
- **Templates**
  - <ui:insert>
  - <ui:define>
  - <ui:include>
  - <ui:define>
- **Parameters**
  - <ui:param>
- **Custom**
- **Remove**

6. JSF Validation Tags

- **f:validateLength :** Validates length of a string
- **f:validateLongRange :** Validates range of numeric value
- **f:validateDoubleRange :** Validates range of float value
- **f:validateRegex :** Validate JSF component with a given regular expression.
- **Custom Validator :** Creating a custom validator.

7. JSF Database Access.

- We can easily integrate JDBC with JSF for Database Access, let's understand that with example.

UserData.java

```
import java.io.Serializable;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.event.ComponentSystemEvent;

@ManagedBean(name = "userData", eager = true)
@SessionScoped
public class UserData implements Serializable {

  private static final long serialVersionUID = 1L;

  public List<Author> getAuthors(){
    ResultSet rs = null;
    PreparedStatement pst = null;
    Connection con = getConnection();
    String stm = "Select * from authors";
    List<Author> records = new ArrayList<Author>();
    try {
      pst = con.prepareStatement(stm);
```

```
            pst.execute();
            rs = pst.getResultSet();

            while(rs.next()){
              Author author = new Author();
              author.setId(rs.getInt(1));
              author.setName(rs.getString(2));
              records.add(author);
            }
          } catch (SQLException e) {
            e.printStackTrace();
          }
          return records;
        }

        public Connection getConnection(){
          Connection con = null;

          String url = "jdbc:postgresql://localhost/testdb";
          String user = "user1";
          String password = "user1";
          try {
            con = DriverManager.getConnection(url, user, password);
            System.out.println("Connection completed.");
          } catch (SQLException ex) {
            System.out.println(ex.getMessage());
          }
          finally{
          }
          return con;
        }
      }
```

Home.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>JSF Tutorial!</title>
    <h:outputStylesheet library="css" name="styles.css"  />
  </h:head>
  <h2>JDBC Integration Example</h2>
  <h:dataTable value="#{userData.authors}" var="c"
    styleClass="authorTable"
    headerClass="authorTableHeader"
    rowClasses="authorTableOddRow,authorTableEvenRow">
    <h:column><f:facet name="header">Author ID</f:facet>
      #{c.id}
    </h:column>
    <h:column><f:facet name="header">Name</f:facet>
      #{c.name}
    </h:column>
  </h:dataTable>
  </h:body>
</html>
```

1. Write a note on Hibernate with its Architecture and Discuss each part in brief.

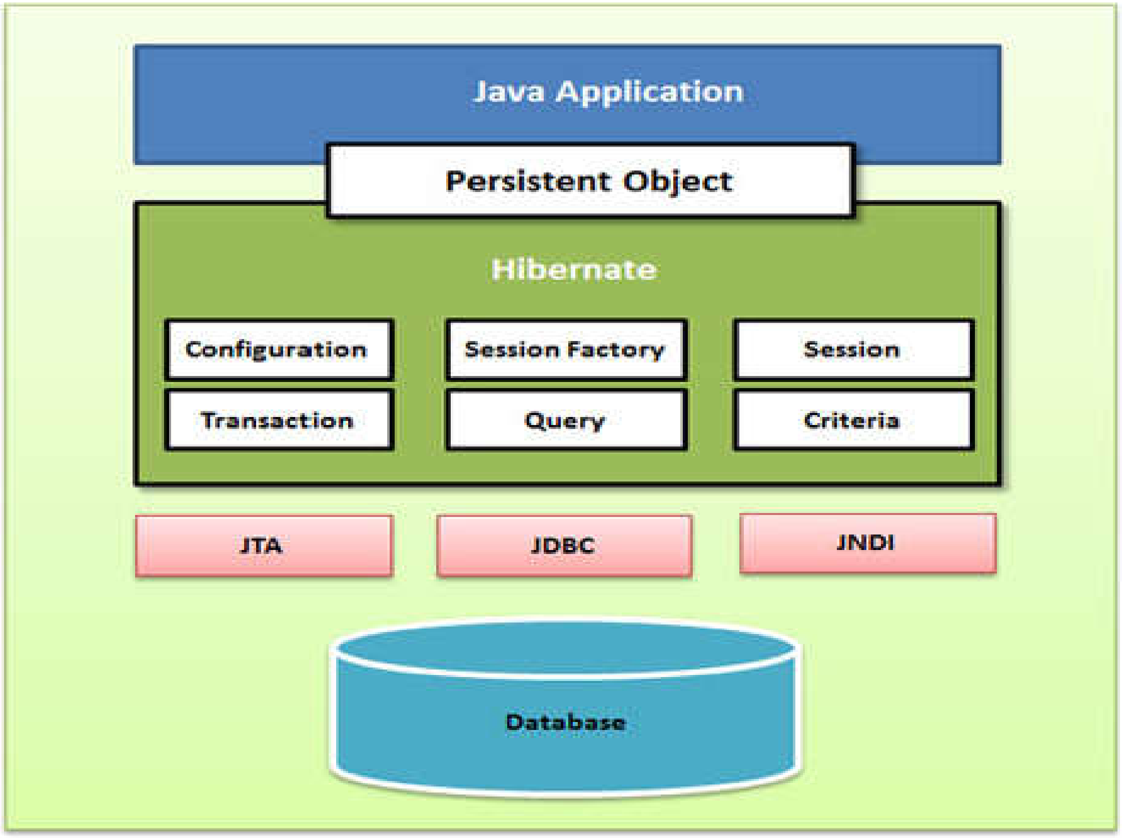


Fig. 1 : Architecture of Hibernate

- In order to persist data to a database, Hibernate create an instance of entity class (Java class mapped with database table).
- This object is called Transient object as they are not yet associated with the session or not yet persisted to a database.
- To persist the object to database, the instance of **SessionFactory** interface is created.
- **SessionFactory** is a singleton instance which implements Factory design pattern. SessionFactory loads hibernate.cfg.xml file and with the help of **TransactionFactory** and **ConnectionProvider** implements all the configuration settings on a database.
- Each database connection in Hibernate is created by creating an instance of Session interface.
- Session represents a single connection with database.
- Session objects are created from **SessionFactory** object.
- Each transaction represents a single atomic unit of work. One Session can span through multiple transactions.
- Here are some of the part of the hibernate architecture

1. **SessionFactory (org.hibernate.SessionFactory)**
   - A thread-safe, immutable cache of compiled mappings for a single database. A factory for org.hibernate.Session instances.
   - A client of org.hibernate.connection.ConnectionProvider.
   - Optionally maintains a second level cache of data that is reusable between transactions at a process or cluster level.
2. **Session (org.hibernate.Session)**
   - A single-threaded, short-lived object representing a conversation between the application and the persistent store.
   - Wraps a JDBC java.sql.Connection.
   - Factory for org.hibernate.Transaction.
   - Maintains a first level cache of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.
3. **Persistent objects and collections**
   - Short-lived, single threaded objects containing persistent state and business function.
   - These can be ordinary JavaBeans/POJOs.
   - They are associated with exactly one org.hibernate.Session.
   - Once the org.hibernate.Session is closed, they will be detached and free to use in any application layer.
4. **Transient and detached objects and collections**
   - Instances of persistent classes that are not currently associated with a org.hibernate.Session.
   - They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed org.hibernate.Session.
5. **Transaction (org.hibernate.Transaction)**
   - A single-threaded, short-lived object used by the application to specify atomic units of work.
   - It abstracts the application from the underlying JDBC, JTA or CORBA transaction.
6. **ConnectionProvider (org.hibernate.connection.ConnectionProvider)**
   - A factory for, and pool of, JDBC connections.
   - It abstracts the application from underlying javax.sql.DataSource or java.sql.DriverManager.
7. **TransactionFactory (org.hibernate.TransactionFactory)**
   - A factory for org.hibernate.Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

2. What is HQL? How does it differ from SQL? Give its advantages.

- Hibernate Query Language HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- The Hibernate Query Language (HQL) is an easy-to-learn and powerful query language designed as an object-oriented extension to SQL.
- HQL queries are case insensitive; however, the names of java classes and properties are case sensitive.If HQL is used in an application define a query for a database, the Hibernate framework automatically generates the SQL query and execute it.
- HQL can also be used to retrieve objects from database through O/R mapping by performing the following tasks:
  - Apply restrictions to properties of objects
  - Arrange the results returned by a query by using the order by clause
  - Paginate the results
  - Aggregate the records by using group by and having clauses
  - Use Joins
  - Create user-defined functions
  - Execute subqueries
- Following are some of the reasons why HQL is preferred over SQL:
  - Provides full support for relation operations
  - Returns results as objects
  - Support polymorphic queries
  - Easy to learn and use
  - Supports for advanced features
  - Provides database independency
- HQL consists of the following elements:
  - Clauses
    - From : from object [as object_alias]
    - Select : select [object.property]
    - Where : where condition
    - Order by : order by obj1.property1 [asc|desc] [obj2.property2]
    - Group by : Group by obj1.property1,obj1.property2
  - Association and Joins
    - Inner join
    - Left outer join
    - Right outer join
    - Full join

- Aggregate functions
  - avg(…)
  - sum(…)
  - min(…)
  - max(…)
  - count(…)
- Expressions
  - Mathematical operators : +,-,*,/
  - Binary comparison operators : =,>=,<=,<>,!=
  - Logical operators : and,or,not
  - String concatenation : ||
- Subqueries
- A query within query is known as subquery and is surrounded by parentheses.

- **FROM Clause :**
  You will use FROM clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list()
```

- **SELECT Clause :**
  The SELECT clause provides more control over the result set than the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause.

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

- **ORDER BY Clause :**
  To sort your HQL query's results, you will need to use the ORDER BY clause.

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

- **USING NAMED Parameters :**
  Hibernate supports named parameters in its HQL queries.This makes writing HQL queries that accept input from the user.

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql); query.setParameter("employee_id",10);
List results = query.list();
```

- **UPDATE Clause :**
  The UPDATE clause can be used to update one or more properties of an one or more objects.

```
String hql = "UPDATE Employee set salary = :salary WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

- **Pagination Using Query :**
  Query setFirstResult - This method takes an integer that represents the first row in your result set, starting with row 0.

  Query setMaxResult - This method tells Hibernate to retrieve a fixed number maxResults of objects.

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

3. Explain the OR Mapping in Hibernate.

- ORM is technique to map object-oriented data with relational data.
- In other words, it is used to convert the datatype supported in object-oriented programming language to a datatype supported by a database.
- ORM is also known as O/RM & O/R mapping.

- Mapping should be in the format that can define the mapping of the
    - classes with tables
    - properties with columns
    - foreign key with associations
    - and SQL types with Java types.
- Mapping can be done with Oracle, DB2, MySql, Sybase and any other relational databases.
- Mapping will be done in the form of XML Document.
- In complex applications, a class can be inherited by other classes, which may lead to the problem of class mismatching. To overcome such problem, the following approach can be used.
    - **Table-per-class-hierarchy:** Removes polymorphism and inheritance relationship completely from the relational model
    - **Table-per-subclass(normalized mapping):** De-Normalizes the relational model and enables polymorphism.
    - **Table-per-concrete-class:** Represents inheritance relationship as foreign key relationships.
- Multiple objects can be mapped to a single row. Polymorphic associations for the 3 strategies are as follows,
    - **Many-to-one:** Serves as an association in which an object reference is mapped to a foreign key association.
    - **One-to-many:** Serves as an association in which a collection of objects is mapped to a foreign key association.
    - **Many-to-many:** Serves as an association in which a collection of objects is transparently mapped to a match table.
- The important elements of the Hibernate mapping file are as follows.
    - **DOCTYPE:** Refers to the Hibernate mapping Document Type Declaration (DTD) that should be declared in every mapping file for syntactic validation of the XML.
    - **<hibernate-mapping> element:** Refers as the first or root element of Hibernate, inside <hibernate-mapping> tag any number of class may be present.
    - **<class> element:** Maps a class object with its corresponding entity in the database.
    - **<id> element :** Serves as a unique identifier used to map primary key column of the database table.
    - **<generator> element:** Helps in generation of the primary key for the new record, following are some of the commonly used generators.
        - Increment , Sequence, Assigned, Identity, Hilo, Native
    - **<property> element:** Defines standard Java attributes and their mapping into database schema.

Example :

- **Employee.java**

```
public class Employee {
            private int employeeId;
            private String name,email;
            private Address address;
            //setters and getters
}
```

- **Address.java**

```
public class Address {
            private int addressId;
            private String addressLine1,city,state,country;
            private int pincode;
            private Employee employee;
//setters and getters
}
```

- **Employee.hbm.xml**

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
         "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Employee" table="emp211">
<id name="employeeId">
<generator class="increment"></generator>
</id>
<property name="name"></property>
<property name="email"></property>
<many-to-one name="address" unique="true" cascade="all"></many-to-one>
</class>
</hibernate-mapping>
```

- **Address.hbm.xml**

```
    <?xml version='1.0' encoding='UTF-8'?>
    <!DOCTYPE hibernate-mapping PUBLIC
            "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
            "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
  <hibernate-mapping>
  <class name="com.javatpoint.Address" table="address211">
  <id name="addressId">
  <generator class="increment"></generator>
  </id>
  <property name="addressLine1"></property>
  <property name="city"></property>
  <property name="state"></property>
  <property name="country"></property>
  </class>
  </hibernate-mapping>
```

- **Hibernate.cfg.xml**

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hbm2ddl.auto">update</property>
      <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
      <property name="connection.username">system</property>
      <property name="connection.password">oracle</property>
      <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
   <mapping resource="employee.hbm.xml"/>
   <mapping resource="address.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

- **StoreData.java**

```
import org.hibernate.cfg.*;
import org.hibernate.*;

public class Store {
        public static void main(String[] args) {
            Configuration cfg=new Configuration();
            cfg.configure("hibernate.cfg.xml");
            SessionFactory sf=cfg.buildSessionFactory();
            Session session=sf.openSession();
            Transaction tx=session.beginTransaction();

            Employee e1=new Employee();
            e1.setName("Ravi Malik");
            e1.setEmail("ravi@gmail.com");

            Address address1=new Address();
            address1.setAddressLine1("G-21,Lohia nagar");
            address1.setCity("Ghaziabad");
            address1.setState("UP");
            address1.setCountry("India");
            address1.setPincode(201301);

            e1.setAddress(address1);
            address1.setEmployee(e1);

            session.save(e1);
            session.save(address1);
            tx.commit();

            session.close();
        }
}
```

- **FetchData.java**

```
public class Fetch {
        public static void main(String[] args) {
            Configuration cfg=new Configuration();
            cfg.configure("hibernate.cfg.xml");
            SessionFactory sf=cfg.buildSessionFactory();
            Session session=sf.openSession();
            Query query=session.createQuery("from Employee e");
            List<Employee> list=query.list();
```

```
                Query query=session.createQuery("from Employee e");
                List<Employee> list=query.list();
                Iterator<Employee> itr=list.iterator();
                while(itr.hasNext()){
                        Employee emp=itr.next();
                        System.out.println(emp.getEmployeeId()+" "+emp.getName()+" "+emp
                    .getEmail());
                        Address address=emp.getAddress();
                        System.out.println(address.getAddressLine1()+" "+address.getCity()+" "
                    +
                          address.getState()+" "+address.getCountry());
                }
                session.close();
                System.out.println("success");
            }
        }
```

4 Explain Hibernate Annotation.

- Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping.
- Consider we are going to use following EMPLOYEE table to store our objects:

```
create table EMPLOYEE (
id INT NOT NULL auto_increment,
first_name VARCHAR(20) default NULL,
 last_name VARCHAR(20) default NULL,
salary INT default NULL,
PRIMARY KEY (id) );
```

- Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table:

```
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
@Id @GeneratedValue
@Column(name = "id")
private int id;
@Column(name = "first_name")
private String firstName;
@Column(name = "last_name")
private String lastName;
@Column(name = "salary")
private int salary;

public Employee() {}
public int getId() {
    return id;
    }
public void setId( int id ) {
    this.id = id;
    }
public String getFirstName() {
    return firstName;
    }
public void setFirstName( String first_name ) {
    this.firstName = first_name;
    }
public String getLastName() {
    return lastName;
    }
public void setLastName( String last_name ) {
    this.lastName = last_name;
    }
public int getSalary() {
    return salary;
    }
public void setSalary( int salary ) {
    this.salary = salary;
    }
}
```

- @Entity Annotation:
    - ☐ Employee class which marks this class as an entity bean
- @Table Annotation:
    - ☐ The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.
- @Id and @GeneratedValue Annotations:
    - ☐ Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation
    - ☐ **@GeneratedValue** is same as Auto Increment

- @Column Annotation:
  - ☐ The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:
  - ☐ **name** attribute permits the name of the column to be explicitly specified.
  - ☐ **length** attribute permits the size of the column used to map a value particularly for a String value.
  - ☐ **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.
  - ☐ **unique** attribute permits the column to be marked as containing only unique values.

1. Explain Spring MVC Architecture

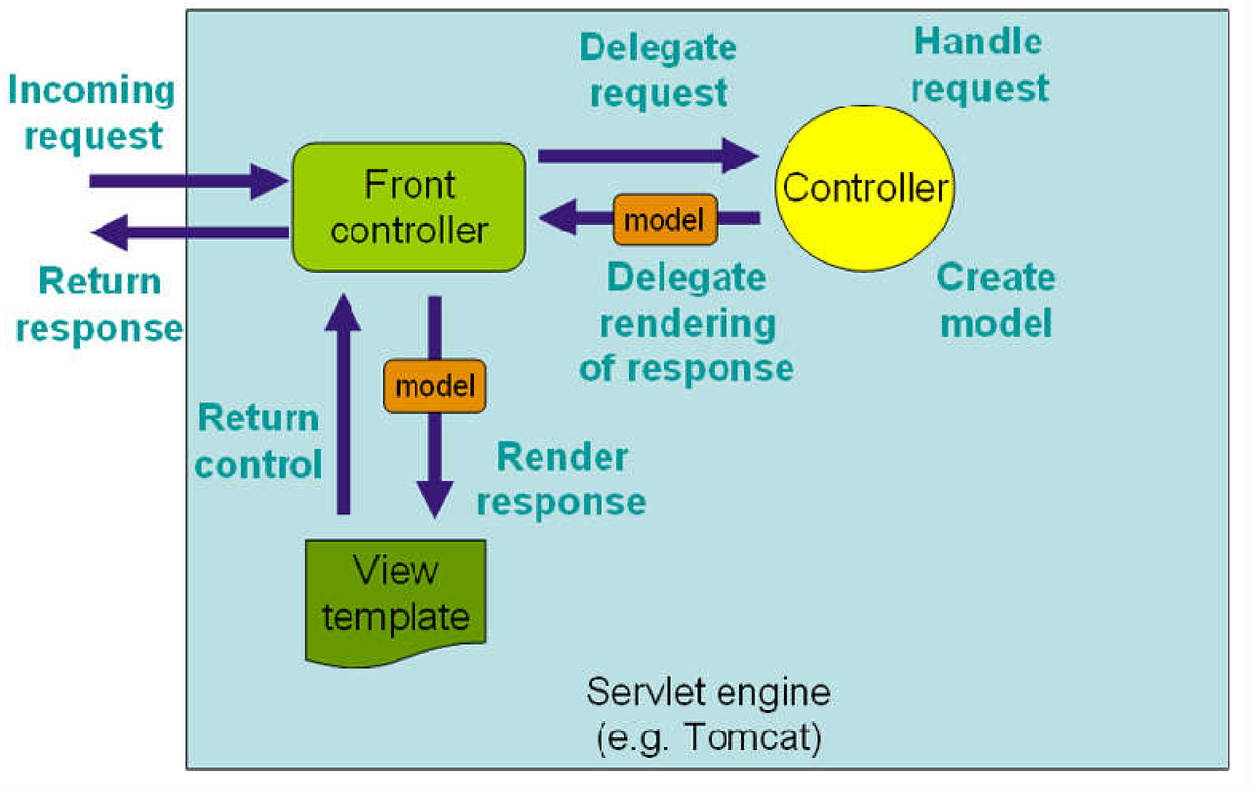


Fig. 1 : Spring MVC Architecture

- Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications.
- Spring's DispatcherServlet is completely integrated with Spring IoC container and allows us to use every other feature of Spring.
- Following is the Request process lifecycle of Spring 3.0 MVC:
  1. The client sends a request to web container in the form of http request.
  2. This incoming request is intercepted by Front controller (DispatcherServlet) and it will then tries to find out appropriate Handler Mappings.
  3. With the help of Handler Mappings, the DispatcherServlet will dispatch the request to appropriate Controller.
  4. The Controller tries to process the request and returns the Model and View object in form of ModelAndView instance to the Front Controller.
  5. The Front Controller then tries to resolve the View (which can be JSP, Freemarker, Velocity etc) by consulting the View Resolver object.
  6. The selected view is then rendered back to client.

Features of Spring 3.0 :

- Spring 3.0 framework supports Java 5. It provides annotation based configuration support. Java 5 features such as generics, annotations, varargs etc can be used in Spring.
- A new expression language Spring Expression Language SpEL is being introduced. The Spring Expression Language can be used while defining the XML and Annotation based bean definition.
- Spring 3.0 framework supports REST web services.
- Data formatting can never be so easy. Spring 3.0 supports annotation based formatting. We can now use the @DateFimeFormat(iso=ISO.DATE) and @NumberFormat(style=Style.CURRENCY) annotations to convert the date and currency formats.
- Spring 3.0 has started support to JPA 2.0.

Configuring Spring 3.0 MVC :

- The entry point of Spring 3.0 MVC is the DispatcherServlet. DispatcherServlet is a normal servlet class which implements HttpServlet base class. Thus we need to configure it in web.xml.

```
<web-app>
  <servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>example</servlet-name>
    <url-pattern>*.html</url-pattern>
  </servlet-mapping>
</web-app>
```

- In above code snippet, we have configure DispatcherServlet in web.xml. Note that we have mapped *.html url pattern with example DispatcherServlet. Thus any url with *.html pattern will call Spring MVC Front controller.
- Once the DispatcherServlet is initialized, it will looks for a file names [servlet-name]-servlet.xml in WEB-INF folder of web application. In above example, the framework will look for file called example-servlet.xml.

2. Explain Bean Life Cycle.

- The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.
- Though, there is lists of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, but this chapter will discuss only two important bean lifecycle callback methods which are required at the time of bean initialization and its destruction.
- To define setup and teardown for a bean, we simply declare the <bean> with init-method and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.
- Here is the content of **HelloWorld.java** file:

```
public class HelloWorld {
  private String message;

  public void setMessage(String message){
    this.message  = message;
  }
  public void getMessage(){
    System.out.println("Your Message : " + message);
  }
  public void init(){
    System.out.println("Bean is going through init.");
  }
  public void destroy(){
    System.out.println("Bean will destroy now.");
  }
}
```

- Following is the content of the **MainApp.java** file. Here you need to register a shutdown hook **registerShutdownHook()** method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.

```
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {

      AbstractApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

- Following is the configuration file Beans.xml required for init and destroy methods:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
      <bean id="helloWorld"
          class="com.HelloWorld"
          init-method="init" destroy-method="destroy">
          <property name="message" value="Hello World!"/>
      </bean>
</beans>
```

3. Explain Transaction Management in Spring MVC.

- A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency. The concept of transactions can be described with following four key properties described as **ACID:**
  - **Atomicity:** A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.
  - **Consistency:** This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

  - **Isolation:** There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.
  - **Durability:** Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.
- Spring supports two types of transaction management:
  - **Programmatic transaction management:** This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
  - **Declarative transaction management:** This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.
- Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code.

Spring Transaction Abstractions

- The key to the Spring transaction abstraction is defined by the org.springframework.transaction.PlatformTransactionManager interface, which is as follows:

```
public interface PlatformTransactionManager {
        TransactionStatus getTransaction(TransactionDefinition definition);
          throws TransactionException;
        void commit(TransactionStatus status) throws TransactionException;
        void rollback(TransactionStatus status) throws TransactionException;
}
```

Sr. No.	Method & Description
1	**TransactionStatus getTransaction(TransactionDefinition definition)** This method returns a currently active transaction or create a new one, according to the specified propagation behavior.
2	**void commit(TransactionStatus status)** This method commits the given transaction, with regard to its status.
3	**void rollback(TransactionStatus status)** This method performs a rollback of the given transaction.