**Module-IV**

**Introduction to Servlets**

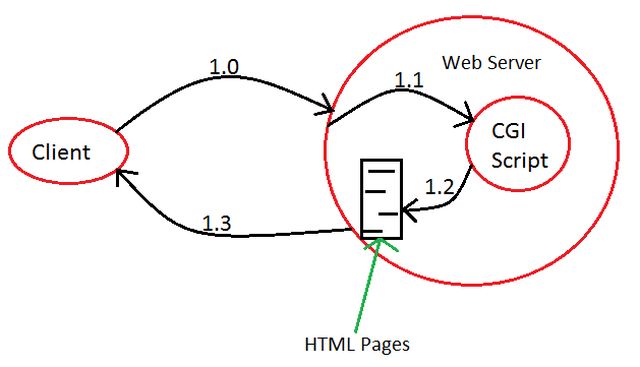
Servlet is a class to extend the properties of a server, It is used to design web-based applications.

**Why we need servlets**

Before servlets there was another API called Common Gateway Interface (CGI) for developing web-based applications.

**What Common Gateway Interface (CGI) :**

It is a protocol that was developed to facilitate communication between web server and a request processing program. In CGI based web applications the request process program was developed in C/C++ or Perl. The programs are called CGI scripts.



**1.0:**The client sends a request for some dynamic content to the server.

**1.1:**On receiving the request **a process** is submitted by the server to create a

GUI and a Script.

**1.2:**The request is processed by a CGI script and dynamic generated contents are

provided to the server.

**1.3:**Dynamically generated content is sent as a response by the server.

**CGI based applications have the following drawbacks:**

1. Execution of CGI scripts are process-based. Process-based execution creates more overhead and limits the scalability.
2. CGI scripts were platform-dependent.

Due to those drawbacks, a new technology is required; (i.e)Servlets.

**Servlets**

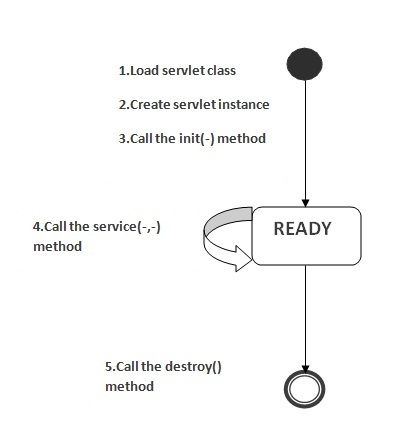
Technology that eliminates both of those drawbacks as in the following:

1. The first drawback is removed by providing a thread-based processing model.
2. The second drawback is removed by facilitating the development by request processing programs in Java.

**Life Cycle of a Servlet (Servlet Life Cycle)**

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end.

The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state.

In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

### 1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

### 2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class.The servlet instance is created only once in the servlet life cycle.

### 3) init() method is invoked

|  |
| --- |
| The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface.  **Syntax of the init() method is given below:** |
|  |

**public** **void** init(ServletConfig config) **throws** ServletException

### 4) service() method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method.

If servlet is initialized, it calls the service method. Notice that servlet is initialized only once.

**The syntax of the service method of the Servlet interface is given below:**

**public** **void** service(ServletRequest request, ServletResponse response)

**throws** ServletException, IOException

### 5) destroy() method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc.

**The syntax of the destroy method of the Servlet interface is given below:**

**public** **void** destroy()

### Servlet API

Servlet API consists of two important packages that encapsulates all the important classes and interfaces, namely :

* **javax.servlet**
* **javax.servlet.http**

#### Some Important Classes and Interfaces of javax.servlet

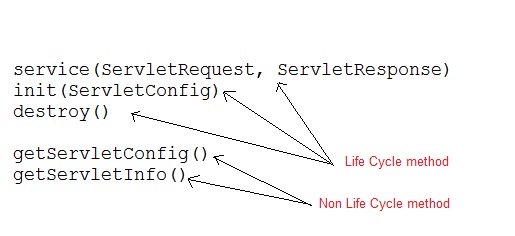
|  |  |
| --- | --- |
| **INTERFACES** | **CLASSES** |
| Servlet | ServletInputStream |
| ServletContext | ServletOutputStream |
| ServletConfig | ServletRequestWrapper |
| ServletRequest | ServletResponseWrapper |
| ServletResponse | ServletRequestEvent |
| ServletContextListener | ServletContextEvent |
| RequestDispatcher | ServletRequestAttributeEvent |
| SingleThreadModel | ServletContextAttributeEvent |
| Filter | ServletException |
| FilterConfig | UnavailableException |
| FilterChain | GenericServlet |
| ServletRequestListener |  |

#### Some Important Classes and Interface of javax.servlet.http

|  |  |
| --- | --- |
| **CLASSES INTERFACES** | |
| HttpServlet | HttpServletRequest |
| HttpServletResponse | HttpSessionAttributeListener |
| HttpSession | HttpSessionListener |
| Cookie | HttpSessionEvent |

#### Servlet Interface

Servlet Interface provides five methods. Out of these five methods, three methods are **Servlet life cycle** methods and rest two are non life cycle methods.



#### GenericServlet Class

GenericServlet is an abstract class that provides implementation of most of the basic servlet methods. This is a very important class.

**Methods of GenericServlet class**

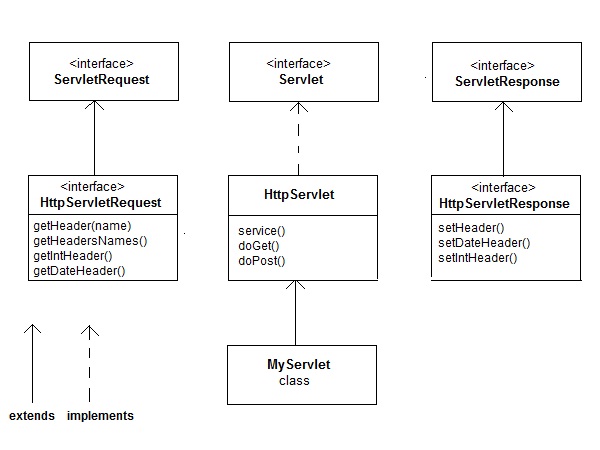
* public void init(ServletConfig)
* public abstract void service(ServletRequest request,ServletResposne response)
* public void destroy()
* public ServletConfig getServletConfig()
* public String getServletInfo()
* public ServletContext getServletContext()
* public String getInitParameter(String name)
* public Enumeration getInitParameterNames()
* public String getServletName()
* public void log(String msg)
* public void log(String msg, Throwable t)

#### HttpServlet class

HttpServlet is also an abstract class. This class gives implementation of various service() methods of **Servlet** interface.

To create a servlet, we should create a class that extends **HttpServlet** abstract class. The Servlet class that we will create, must not override service() method. Our servlet class will override only the doGet() and/or doPost() methods.

The service() method of **HttpServlet** class listens to the Http methods (GET, POST etc) from request stream and invokes doGet() or doPost() methods based on Http Method type.



**Handling HttpRequest and HttpResponses**

* HttpServlet gets request from a client (browser) on the HttpServletRequest interface and can return response to the client through the HttpServletResponse interface in all the doXXXX methods.

### Here is some of most used HttpServletRequest Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public Enumeration getParameterNames() | Names of the parameters contained in this request. |
| public String[] getParameterValues(String name) | Used when the named parameter may have multiple values. |
| public Map getParameterMap() | Returns all the parameters stored in a Map object. |
| public BufferedReader getReader() | This will retrieve the body of a request as characters into a BufferedReader. |
| public ServletInputStream getInputStream() | This will retrieve the body of a request as binary data into a ServletInputStream. |
| public Enumeration getHeaderNames() | Returns an enumeration of all the header names this request contains. |
| public String getQueryString() | This will return the query string that is contained in the request URL after the path. |
| public RequestDispatcher getRequestDispatcher(String path) | A RequestDispatcher object can be used to forward a request to the resource or to include the resource in a response. |

* We have to use either the getReader() method or the getInputStream(). You cannot combine to use these for the same request.

### Here is some of most used HttpServletResponse Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public PrintWriter getWriter() | This will return a PrintWriter object that can send character text to the client. |
| public ServletOutputStream getOutputStream() | This will return a ServletOutputStream suitable for writing binary data in the response. |
| public addHeader(String name, String value) | This will add a property, which is a String name with a String value to the response header. |
| public addDateHeader(String name, long date) | This will add a property, which is a String name with a long date value to the response header. |
| public void sendRedirect(String location) | This will send a temporary redirect response to the client using the specified redirect location URL. |
| public Enumeration getHeaderNames() | Returns an enumeration of all the header names this request contains. |

* We have to use either the getWriter() method or the getOutputStream(). You cannot combine to use these for the same response

## Example of Servlet handling a html form input.

import java.io.\*;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class FormHandlerServlet extends HttpServlet

{

public void doPost(HttpServletRequest request, HttpServletResponse response)

{

String enteredValue;

// gets all the selected options from the client browser

String[] selectedOptions = request.getParameterValues("options");

// gets the enteredValue fields value

enteredValue = request.getParameter("enteredValue");

response.setContentType("text/html");

PrintWriter printWriter;

try

{

// get a printwriter from the HttpServletResponse objects ref.

printWriter = response.getWriter();

// return on the HttpServletResponse objects ref. requested values

printWriter.println("<p>");

printWriter.print("You entered: ");

printWriter.print(enteredValue);

printWriter.print("</p>");

printWriter.println("<p>");

printWriter.print("The following options were selected:");

printWriter.println("<br/>");

if (selectedOptions != null)

{

for (String option : selectedOptions)

{

printWriter.print(option); printWriter.println("<br/>");

}

}

else

{

printWriter.println("None");

}

printWriter.println("</p>");

}

catch (IOException e)

{

e.printStackTrace();

}

}

}

•In this servlet we expect a POST Http request and we fetch all the values we

know about from a html form (read the comments).

•We set the Content type to "text/html" for the response to the client (browser)

and use a PrintWriter on the response object to return html codes and text back

to the client.

**Dataentry.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="windows-1252">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Data Entry Page</title>

</head>

<body>

<form method="post" action="FormHandlerServlet"

style="width: 310px; border: 1px solid black;" >

<table >

<tr>

<td>Please enter some text:</td>

<td><input type="text" name="enteredValue" />

</td>

</tr>

<tr>

<td colspan="2">

<input name="options" type="checkbox" value="option1" />

Option 1</td>

</tr>

<tr>

<td colspan="2">

<input name="options" type="checkbox" value="option2" />

Option 2</td>

</tr>

<tr>

<td colspan="2">

<input name="options" type="checkbox" value="option3" />

Option 3</td>

</tr>

<tr>

<td colspan="2" style="text-align: right;" >

<input type="submit" value="Submit">

</td>

</tr>

</table>

</form>

</body>

</html>

# Cookies in Servlet

A **cookie** is a small piece of information stored in text file that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

### How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet.

So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by[ default. Thus, we recognize the user as the old user.



### Types of Cookies

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie

### Non-persistent Cookie

It is **valid for single session** only. It is removed each time when user closes the browser.

### Persistent cookie

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

### Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

### Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

**Example to represent Servlets with Cookies**

### Index.html

### <html>

<body>

### <form action=”http://localhost:8080/examples/servlets/servlet/FirstServlet"

method="post">

Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

</body>

</html>

### FirstServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet

{

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response)

{

**try**

{

      response.setContentType("text/html");

     PrintWriter out = response.getWriter();

     String n=request.getParameter("userName");

     out.print("Welcome "+n);

      Cookie ck=**new** Cookie("username",n);//creating cookie object

     response.addCookie(ck);//adding cookie in the response

    //creating submit button

    out.print("<form action='<http://localhost:8080/examples/servlets/servlet/SecondServlet>'method=’GET’>");

   out.print("<input type='submit' value='go'>");

    out.print("</form>");

    out.close();

        }

**catch**(Exception e)

{

System.out.println(e);

}

  }

}

### SecondServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response){

**try**{

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    Cookie ck[]=request.getCookies();

    out.print("Hello "+ck[0].getValue());

    out.close();

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

    }

}

### web.xml

<web-app>

<servlet>

<servlet-name> FirstServlet </servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>FirstServlet</servlet-name>

<url-pattern>/servlets/servlet/FirstServlet</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>SecondServlet</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

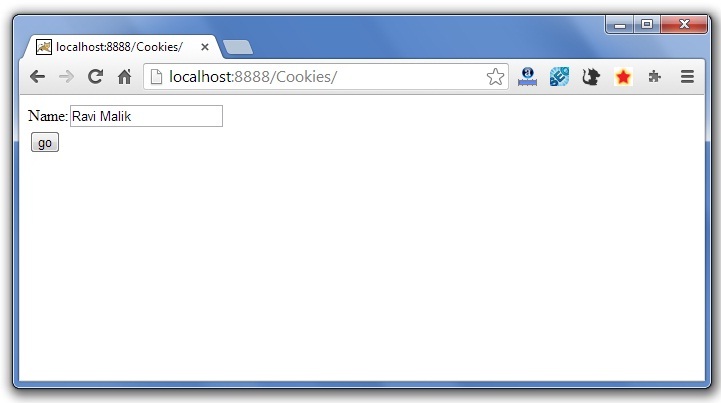
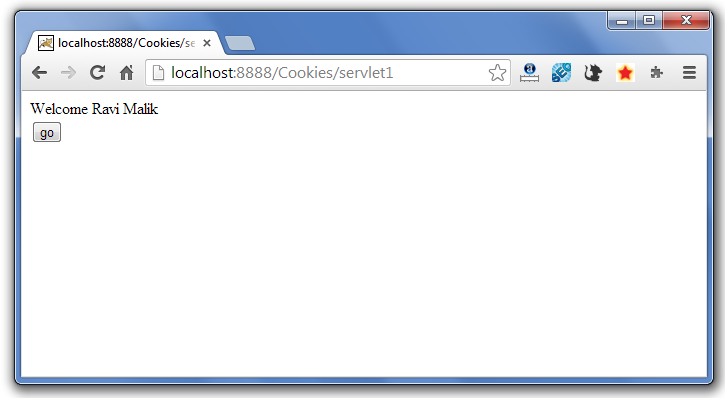
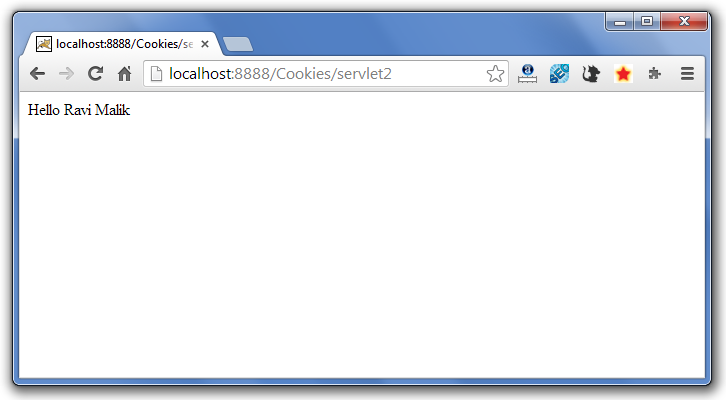
<servlet-name>SecondServlet</servlet-name>

<url-pattern>/servlets/servlet/SecondServlet</url-pattern>

</servlet-mapping>

</web-app>

#### Output

**Session Tracking in Servlets**

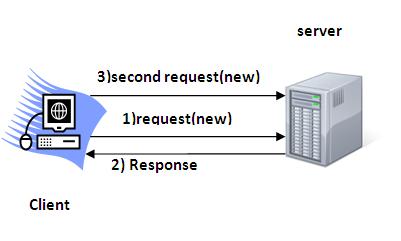
**Session** simply means a particular interval of time.**Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:

### Need of Session Tracking

**Session Tracking** is used to recognize the particular user.



### Session Tracking Techniques

There are four techniques used in Session tracking:

1. **Cookies(Given above)**
2. **Hidden Form Field**
3. **URL Rewriting**
4. **HttpSession**

# 2) Hidden Form Field

In case of Hidden Form Field **a hidden (invisible) textfield** is used for maintaining the state of an user.

In such case, we store the information in the hidden field and get it from another servlet. **This approach is better if we have to submit form in all the pages and we don't want to depend on the browser**.

The code to store value in hidden field.

1. <input type="hidden" name="uname" value="Vimal Jaiswal">

Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

### Real application of hidden form field

It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

### Advantage of Hidden Form Field

1. It will always work whether cookie is disabled or not.

### Disadvantage of Hidden Form Field:

1. It is maintained at server side.
2. Extra form submission is required on each pages.
3. Only textual information can be used.

### (3) Example of using URL Rewriting

In this example, we are maintaning the state of the user using link. For this purpose, **we are appending the name of the user in the query string and getting the value from the query string in another page**.

### Index.html

<form action="servlet1">

Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

**FirstServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet

{

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

{

**try**

{

         response.setContentType("text/html");

        PrintWriter out = response.getWriter();

   String n=request.getParameter("userName");

   out.print("Welcome "+n);

    //appending the username in the query string

     out.print("<a href=' http://localhost:8080/examples/servlets/servlet/SecondServlet?uname="+n+"'>visit</a>");

       out.close();

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

    }

}

### SecondServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet

{

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

{

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        //getting value from the query string

        String n=request.getParameter("uname");

        out.print("Hello "+n);

        out.close();

              }

**catch**(Exception e)

{

System.out.println(e);

}

   }

  }

### web.xml

<web-app>

<servlet>

<servlet-name> FirstServlet </servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>FirstServlet</servlet-name>

<url-pattern>/servlets/servlet/FirstServlet</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>SecondServlet</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>SecondServlet</servlet-name>

<url-pattern>/servlets/servlet/SecondServlet</url-pattern>

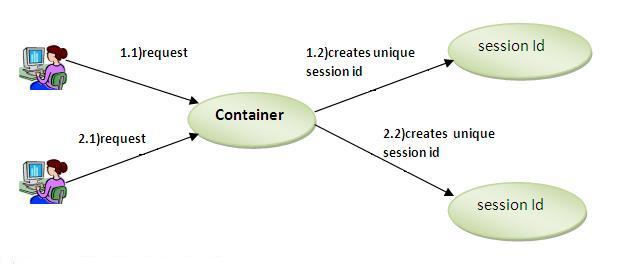
</servlet-mapping>

</web-app>

# 4) HttpSession interface

In this case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform **two tasks**:

1. bind objects
2. **view and manipulate information** about a session, such as the s**ession identifier, creation time, and last accessed time**.



### Getting the HttpSession object

The **HttpServletRequest interface provides two method**s to **get the object of HttpSession**:

1. **public HttpSession getSession():**Returns the current session associated with this request, or if the request does not have a session, creates one.
2. **public HttpSession getSession(boolean create):**Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

### Commonly used methods of HttpSession interface

1. **public String getId():**Returns a string containing the unique identifier value.
2. **public long getCreationTime():**Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
3. **public long getLastAccessedTime():**Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
4. **public void invalidate():**Invalidates this session then unbinds any objects bound to it.

### Example of using HttpSession

In this example, we are **setting the attribute in the session scope in one servle**t and **getting that value from the session scope in another servlet**. To set the attribute in the session scope, we have used the setAttribute() method of HttpSession interface and to get the attribute, we have used the getAttribute method.

### Index.html

### <form action=”http://localhost:8080/examples/servlets/servlet/FirstServlet"

method="post">

User Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

### FirstServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response){

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");

        out.print("Welcome "+n);

        HttpSession session=request.getSession();

        session.setAttribute("uname",n);

        out.print("<a href=' http://localhost:8080/examples/servlets/servlet/SecondServlet '>visit</a>");

        out.close();

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

    }

}

**SecondServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        HttpSession session=request.getSession(**false**);

        String n=(String)session.getAttribute("uname");

        out.print("Hello "+n);

        out.close();

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

    }

}

**web.xml**

<web-app>

 <servlet>

<servlet-name> FirstServlet </servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>FirstServlet</servlet-name>

<url-pattern>/servlets/servlet/FirstServlet</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>SecondServlet</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>SecondServlet</servlet-name>

<url-pattern>/servlets/servlet/SecondServlet</url-pattern>

</servlet-mapping>

  </web-app>

### Differences between Session and Cookie

|  |  |
| --- | --- |
| **Session** | **Cookie** |
| Data on server-side | data on client side |
| unlimited side of data as per as server capability | limited support for data data handling |
| It can store any type of data | only text |
| age of data is not fixed . | Fixed |
| destroy after session timeout or logout | remains on client machine |
| less data traveling over the network | All cookie need to travel each time client sends request to server. |
| More secure mechanism to session tracking | less secure |

# Java JDBC

Java JDBC is **a java API** to **connect and execute query with the database**. JDBC API **uses jdbc drivers** to **connect with the databa**se.

JDBC (Java Database Connectivity) 

### Need of JDBC

Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

# JDBC Drivers

|  |
| --- |
| JDBC Driver is **a software component** that **enables java application to interact with the database.**  ->There are 4 types of JDBC drivers:   1. JDBC-ODBC bridge driver 2. Native-API driver (partially java driver) 3. Network Protocol driver (fully java driver) 4. Thin driver (fully java driver) |
|  |

### 1) JDBC-ODBC bridge driver

|  |
| --- |
| The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver **converts JDBC method calls into the ODBC function calls**. This is now **discouraged because of thin driver**. |
|  |



### Advantages:

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

### Disadvantages:

* Performance degraded because **JDBC method call is converted into the ODBC** function calls.
* The **ODBC driver** needs **to be installed** on the client machine.

### 2) Native-API driver

|  |
| --- |
| The **Native API driver uses the client-side libraries of the database.** The driver **converts JDBC method calls into native calls of the database API**. It is not written entirely in java. |



### Advantages:

* Performance upgraded than JDBC-ODBC bridge driver.

### Disadvantages:

* The Native driver needs to be installed on the each client machine.
* The Vendor client library needs to be installed on client machine.

### 3) Network Protocol driver

The Network Protocol driver u**ses middleware (application server) that converts JDBC calls directly** **or indirectly** into the **vendor-specific database protocol.** It is fully written in java.



### Advantage:

* No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

### Disadvantages:

* Network support is required on client machine.
* Requires database-specific coding to be done in the middle tier.
* Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

### 4) Thin driver

|  |
| --- |
| The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language. |
|  |



### Advantages:

* Better performance than all other drivers.
* No software is required at client side or server side.

### Disadvantages:

* Drivers depends on the Database.

# 5 Steps to connect to the database in java

|  |
| --- |
| There are 5 steps to connect any java application with the database in java using JDBC.  They are as follows:   * Register the driver class * Creating connection * Creating statement * Executing queries * Closing connection |

### I) Register the driver class

|  |
| --- |
| The **forName() method** of **Class** class is used to register the driver class. This method is used to dynamically load the driver class. |

### Syntax of forName() method

**public** **static** **void** forName(String className)**throws** ClassNotFoundException

### Example:

### (1)To register the OracleDriver class

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

**(2)** To register the MySqlDriver class

Class.forName("com.mysql.jdbc.Driver");

### II) Create the connection object

|  |
| --- |
| The **getConnection() method of DriverManager class** is used to establish connection with the database. |
|  |

### Syntax of getConnection() method

1. **public** **static** Connection getConnection(String url)**throws** SQLExce

ption

1. **public** **static** Connection getConnection(String url,String name,Strin

g password)**throws** SQLException

### Example:

### 1)To establish connection with the Oracle database

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","password");

### 2) To establish connection with the MySql database

  Connection con=DriverManager.getConnection(

"jdbc:mysql://localhost:3306/test","root","root");

**III)Create the Statement object**

|  |
| --- |
| The **createStatement() method of Connection** **interface** is used to create statement. The object of statement is responsible to execute queries with the database. |

### Syntax of createStatement() method

1. **public** Statement createStatement()**throws** SQLException

### Example to create the statement object

1. Statement stmt=con.createStatement();

**IV)Execute the query**

|  |
| --- |
| **The executeQuery() method of Statement interface** is used to execute queries to the database. This **method returns** the **object of ResultSet** that can be used to get all the records of a table. |

### Syntax of executeQuery() method

**public** ResultSet executeQuery(String sql)**throws** SQLException

### Example to execute query

ResultSet rs=stmt.executeQuery("select \* from Emp");

**while**(rs.next()){

System.out.println(rs.getInt(1)+" "+rs.getString(2));

}

### V) Close the connection object

|  |
| --- |
| By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection. |

### Syntax of close() method

**public** **void** close()**throws** SQLException

### Example to close connection

con.close();

Example: Servlet Program to store Employee details in Database using DataBase

Connectivity.

**EmpDetails.html**

<html>

<body background-color:"pink">

<h1>Storage of Employee Details in DataBase<h1>

<form action="http://localhost:8080/examples/servlets/servlet/RegisterServlet" method="post">

Employee No :<input type="text" name="Eno"/><br/><br/>

Employee Name:<input type="text" name="Ename"/><br/><br/>

Salary :<input type="text" name="Salary" size="20"/><br/><br/>

<br/><br/>

<input type="submit" value="Submit"/>

</form>

</body>

</html>

**RegisterServlet.java**

import java.io.\*;

import java.sql.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public final class RegisterServlet extends HttpServlet {

String DB\_CONNECTION\_URL = "jdbc:mysql://localhost:3306/test";

String DB\_USERNAME = "root";

String DB\_PASSWORD = "root";

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException

{

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String Eno = request.getParameter("Eno");

String Ename = request.getParameter("Ename");

String Salary= request.getParameter("Salary");

out.print(Salary+"<br>");

try {

Class.forName("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection(DB\_CONNECTION\_URL, DB\_USERNAME, DB\_PASSWORD);

PreparedStatement ps = con.prepareStatement("insert into Employee(Eno,Ename,Salary) values(?,?,?)");

ps.setString(1, Eno);

ps.setString(2, Ename);

ps.setString(3,Salary);

int result = ps.executeUpdate();

if(result == 1){

out.print("You are successfully Inserted...");

}

else{

out.print("You are successfully not Inserted...");

}

} catch(Exception e) {

e.printStackTrace();

}

}

}

### web.xml

<web-app>

<servlet>

<servlet-name>RegisterServlet</servlet-name>

<servlet-**class**>RegisterServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>RegisterServlet</servlet-name>

<url-pattern>/servlets/servlet/RegisterServlet</url-pattern>

</servlet-mapping>

  <welcome-file-list>

<welcome-file>EmpDetails.html</welcome-file>

</welcome-file-list>

</web-app>

**Chapter-2**

## Introduction about MVC

MVC is an architecture or design pattern that separates business logic, presentation and data. In MVC,

* M stands for Model
* V stands for View
* C stands for controller.

MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

**Components of MVC**

**Model Layer:**

* This is the data layer which consists of the business logic of the system.
* It consists of all the data of the application
* It also represents the state of the application.
* It consists of classes which have the connection to the database.
* The controller connects with model and fetches the data and sends to the view layer.
* The model connects with the database as well and stores the data into a database which is connected to it.

**View Layer:**

* This is a presentation layer.
* It consists of HTML, JSP, etc. into it.
* It normally presents the UI of the application.
* It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.
* This view layer shows the data on UI of the application.

**Controller Layer:**

* It acts as an interface between View and Model Layers.
* It intercepts all the requests which are coming from the view layer.
* It receives the requests from the view layer and processes the requests and does the necessary validation for the request.
* This requests is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.

**The MVC Architecture**



**The advantages of MVC are:**

* Easy to maintain
* Easy to extend
* Easy to test
* Navigation control is centralized

### MVC Example

In this example, we are using servlet as a controller, jsp as a view component, Java Bean class as a model.

In this example, we have created 5 pages:

* **Index.jsp** a page that gets input from the user.
* **ControllerServlet.java** a servlet that acts as a controller.
* **login-success.jsp** and **login-error.jsp** files acts as view components.
* **web.xml** file for mapping the servlet.

**File: Index.jsp**

<form action="ControllerServlet" method="post">

Name:<input type="text" name="name"><br>

Password:<input type="password" name="password"><br>

<input type="submit" value="login">

</form>

**File: ControllerServlet**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** ControllerServlet **extends** HttpServlet {

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out=response.getWriter();

        String name=request.getParameter("name");

        String password=request.getParameter("password");

        LoginBean bean=**new** LoginBean();

        bean.setName(name);

        bean.setPassword(password);

        request.setAttribute("bean",bean);

**boolean** status=bean.validate();

**if**(status){

            RequestDispatcher rd=request.getRequestDispatcher("login-success.jsp");

            rd.forward(request, response);

        }

**else**{

            RequestDispatcher rd=request.getRequestDispatcher("login-error.jsp");

            rd.forward(request, response);

        }

    }

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

        doPost(req, resp);

    }

}

**File: LoginBean.java**

**public** **class** LoginBean

{

**private** String name,password;

**public** String getName()

{

**return** name;

}

**public** **void** setName(String name)

{

**this**.name = name;

}

**public** String getPassword()

{

**return** password;

}

**public** **void** setPassword(String password)

 {

**this**.password = password;

}

**public** **boolean** validate()

{

**if**(password.equals("admin")){

**return** **true**;

    }

**else**{

**return** **false**;

    }

}

}

**File: login-success.jsp**

<p>You are successfully logged in!</p>

**File: login-error.jsp**

<p>Sorry! username or password error</p>

<%@ include file="index.jsp" %>

**File: web.xml**

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

**<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

id="WebApp\_ID" version="3.0"**>**

**<servlet>**

**<servlet-name>**s1**</servlet-name>**

**<servlet-class>**com.java.ControllerServlet**</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>**s1**</servlet-name>**

**<url-pattern>**/ControllerServlet**</url-pattern>**

**</servlet-mapping>**

**</web-app>**

#### mvc output 2 mvc output 3