UNIT-IV
JDBC AND SERVLETS

Introduction to Servlets:

Servlets are server side components that provide a powerful mechanism for developing server side programs. Servlets provide component-based, platform-independent methods for building Web-based applications. Using Servlets web developers can create fast and efficient server side application which can run on any Servlet enabled web server. Servlets can access the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls; receive all the benefits of the mature java language including portability, performance, reusability, and crash protection. Today Servlets are the popular choice for building interactive web applications. Servlet containers are usually the components of web and application servers, such as BEA Weblogic Application Server, IBM Web Sphere, Sun Java System Web Server, Sun Java System Application Server and others. Servlets are not designed for a specific protocol. It is different thing that they are most commonly used with the HTTP protocols Servlets uses the classes in the java packages javax.servlet and javax.servlet.http. Servlets provides a way of creating the sophisticated server side extensions in a server as they follow the standard framework and use the highly portable java language.

HTTP Servlets Typically Used To:

- Provide dynamic content like getting the results of a database query and returning to the client.
- Process and/or store the data submitted by the HTML.
- Manage information about the state of a stateless HTTP. e.g. an online shopping car manages request for multiple concurrent customers.

Methods of Servlets:

A Generic Servlet contains the following five methods:

- **init():**

  public void init(ServletConfig config) throws ServletException

  The init () method is called only once by the servlet container throughout the life of a Servlet. By this init () method the Servlet get to know that it has been placed into service.

  The Servlet cannot be put into the service if

  - The init () method does not return within a fix time set by the web server.
  - It throws a ServletException

  Parameters - The init () method takes a ServletConfig object that contains the initialization parameters and Servlet's configuration and throws a ServletException if an exception has occurred.

- **service():**

  public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException

  Once the Servlet starts getting the requests, the service() method is called by the Servlet container to respond. The Servlet services the client's request with the help of two objects. These two objects are javax.servlet.ServletRequest and javax.servlet. Servlet Response are passed by the Servlet container. The status code of the response always

SREC

should be set for a Servlet that throws or sends an error. Parameters - The service () method takes the ServletRequest object that contains the client's request and the object ServletResponse contains the Servlet's response. The service() method throws ServletException and IOException exception.

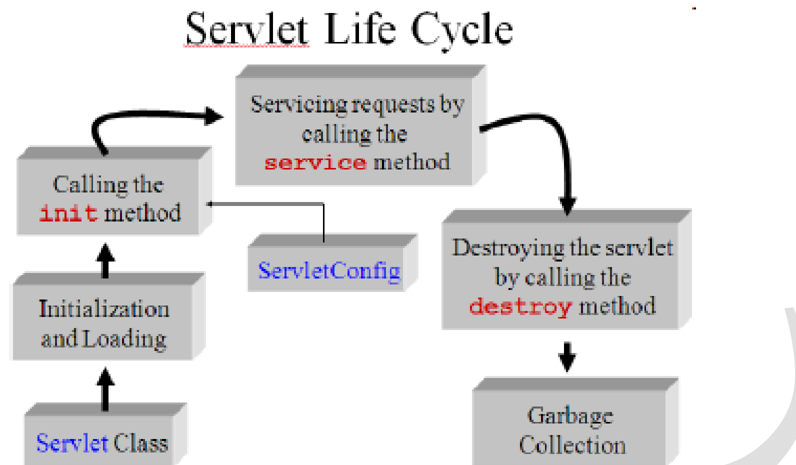
- **getServletConfig():**
  public ServletConfig getServletConfig()
  This method contains parameters for initialization and startup of the Servlet and returns a ServletConfig object. This object is then passed to the init method. When this interface is implemented then it stores the ServletConfig object in order to return it. It is done by the generic class which implements this interface.
  Returns - the ServletConfig object
- **getServletInfo():**
  public String getServletInfo ()
  The information about the Servlet is returned by this method like version, author etc. This method returns a string which should be in the form of plain text and not any kind of markup.
  Returns - a string that contains the information about the Servlet
- **destroy():**
  public void destroy()
  This method is called when we need to close the Servlet. That is before removing a Servlet instance from service, the Servlet container calls the destroy() method. Once the Servlet container calls the destroy() method, no service methods will be then called . That is after the exit of all the threads running in the Servlet, the destroy() method is called. Hence, the Servlet gets a chance to clean up all the resources like memory, threads etc which are being held.

Life cycle of Servlet:

The life cycle of a Servlet can be categorized into four parts:

1. **Loading and Instantiation:** The Servlet container loads the Servlet during startup or when the first request is made. The loading of the Servlet depends on the attribute <load-on-startup> of web.xml file. If the attribute <load-on-startup> has a positive value then the Servlet is load with loading of the container otherwise it load when the first request comes for service. After loading of the Servlet, the container creates the instances of the Servlet.
2. **Initialization:** After creating the instances, the Servlet container calls the init() method and passes the Servlet initialization parameters to the init() method. The init() must be called by the Servlet container before the Servlet can service any request. The initialization parameters persist until the Servlet is destroyed. The init() method is called only once throughout the life cycle of the Servlet. The Servlet will be available for service if it is loaded successfully otherwise the Servlet container unloads the Servlet.
3. **Servicing the Request:** After successfully completing the initialization process, the Servlet will be available for service. Servlet creates separate threads for each request. The Servlet container calls the service() method for servicing any request. The service() method determines the kind of request and calls the appropriate method (doGet () or doPost ()) for handling the request and sends response to the client using the methods of the response object.

4. **Destroying the Servlet:** If the Servlet is no longer needed for servicing any request, the Servlet container calls the destroy() method . Like the init() method this method is also called only once throughout the life cycle of the Servlet. Calling the destroy() method indicates to the Servlet container not to sent the any request for service and the Servlet releases all the resources associated with it. Java Virtual Machine claims for the memory associated with the resources for garbage collection.



The Advantages of Servlets:

1. Portability
2. Powerful
3. Efficiency
4. Safety
5. Integration
6. Extensibilty
7. Inexpensive

Each of the points is defined below:

1. **Portability:** As we know that the Servlets are written in java and follow well known standardized APIs so they are highly portable across operating systems and server implementations. We can develop a Servlet on Windows machine running the tomcat server or any other server and later we can deploy that Servlet effortlessly on any other operating system like UNIX server running on the iPlanet/Netscape Application server. So Servlets are Write Once, Run Anywhere (WORA) program.
2. **Powerful:** We can do several things with the Servlets which were difficult or even impossible to do with CGI, for example the Servlets can talk directly to the web server while the CGI programs can't do. Servlets can share data among each other, they even make the database connection pools easy to implement. They can maintain the session by using the session tracking mechanism which helps them to maintain information from request to request. It can do many other things which are difficult to implement in the CGI programs.
3. **Efficiency:** As compared to CGI the Servlets invocation is highly efficient. When the Servlet get loaded in the server, it remains in the server's memory as a single object instance. However with Servlets there are N threads but only a single copy of the Servlet

class. Multiple concurrent requests are handled by separate threads so we can say that the Servlets are highly scalable.

4. **Safety:** As Servlets are written in java, Servlets inherit the strong type safety of java language. Java's automatic garbage collection and a lack of pointers mean that Servlets are generally safe from memory management problems. In Servlets we can easily handle the errors due to Java's exception handling mechanism. If any exception occurs then it will throw an exception.
5. **Integration:** Servlets are tightly integrated with the server. Servlet can use the server to translate the file paths, perform logging, check authorization, and MIME type mapping etc.
6. **Extensibility:** The Servlet API is designed in such a way that it can be easily extensible. As it stands today, the Servlet API support Http Servlets, but in later date it can be extended for another type of Servlets.
7. **Inexpensive:** There are number of free web servers available for personal use or for commercial purpose. Web servers are relatively expensive. So by using the free available web servers you can add Servlet support to it.

DEPLOYING A SERVLET ON WEB SERVER:

- **Step 1:** First of all you need to install the Apache Tomcat Server and JDK.
  As mentioned earlier, Apache's Tomcat Server is free software available for download @ www.apache.org. You have to download the Tomcat Server 6.0. This Server supports Java Servlets 2.5 and Java Server Pages (JSPs) 2.1 specifications. Important software required for running this server is Sun's JDK (Java Development Kit) and JRE (Java Runtime Environment). The current version of JDK is 1.8. Like Tomcat, JDK is also free and is available for download at www.java.sun.com.
- **Step 2:** Next configure the Tomcat server and JDK by setting up the environment variables for JAVA_HOME variable - You have to set this variable which points to the base installation directory of JDK installation. (e.g. C:\Program Files\Java\jdk1.8.xx\bin). And CATALINA_HOME variable – you have to set this variable which points to the base installation directory of Tomcat installation. (e.g. C:\Program Files\Apache Software Foundation\Tomcatx.x\bin\.
- **Step 3:Write Your Servlet:**
  Here is simple servlet program which can be written in Notepad or EditPlus and saved using .java extension.
  PROGRAM:

```
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 req,ServletResponse res)throws
ServletException,IOException
```

```
        {
                System.out.println("from service");
                PrintWriter out=res.getWriter();
                out.println("LifeCycle\n");
                out.println("III CSE Students");
        }
        public void destroy()
        {
                System.out.println("destroy");
        }
}
```

- **Step 4:** Then set the classpath of the servlet-api.jar file in the variable CLASSPATH inside the environment variable as "C:\Program Files\Apache Tomcat Foundation\Tomcat x.x\lib\servlet-api.jar". Then compile your servlet by using **javac LifeCycle.java.**
- **Step 5:** The next step is to create your web application folder. The name of the folder can be any valid and logical name that represents your application (e.g. bank_apps, airline_tickets_booking, shopping_cart,etc). But the most important criterion is that this folder should be created under webapps folder. The path would be similar or close to this - C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps. For demo purpose, let us create a folder called example programs under the webapps folder.
- **Step 6:** Next create the WEB-INF folder in C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\Example Programs folder.
- **Step 7:** Then create the web.xml file and the classes folder. Ensure that the web.xml and classes folder are created under the WEB-INF folder. The web.xml file contains the following information.
    - The servlet information
    - The mapping information from the server to our web application.
- **Step 8:** Then copy the Servlet class file to the classes folder in order to run the Servlet that we created. All you need to do is copy the Servlet class file (the file we obtained from Step 4) to this folder.
- **Step 9:** Edit web.xml to include Servlet's name and URL pattern. This step involves two actions viz. including the Servlet's name and then mentioning the url- pattern. Let us first see as how to include the Servlet's name in the web.xml file.

```
<web-app>
        <servlet>
                <servlet-name>MyServ</servlet-name>
                <servlet-class>LifeCycle</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>MyServ</servlet-name>
                <url-pattern>/lc</url-pattern>
        </servlet-mapping>
</web-app>
```

Note – The Servlet-name need not be the same as that of the class name. You can give a different name (or alias) to the actual Servlet. This is one of the main reasons as why this tag is used for. Next, include the url pattern using the <servlet-mapping> </servlet-mapping> tag. The url pattern defines as how a user can access the Servlet from the browser.

- **Step 10:** Run Tomcat server and then execute your Servlet by opening any one of your web browser and enter the URL as specified in the web.xml file. The complete URL that needs to be entered in the browser is: http://localhost:8080/scce/lc

Invoking Servlet using HTML:

It is a common practice to invoke servlet using HTML form. This will be achieved by the action attribute of the form tag in HTML. To understand this, we have to create a web page which will invoke above created servlet. In the above method, we have seen that the servlet is invoked from the URL, but here in this example the same servlet will be invoked by clicking the button in the web page.

HTML Program:

```
<html>
<head><title> Life Cycle of Servlet </title>
</head>
<body>
<form name="form1" action="lc">
<b> My Life Class</b>
<input type="submit" value="Go to My Life Cycle">
</form>
</body>
</html>
```

For getting the output:

1. Compile the servlet program using javac compiler.
2. Copy the generated class file into your web application's classes folder.
3. Enter the http://localhost:8080/Example%20Programs/mylifecycle.html

The Servlet API:

Two packages contain the classes and interfaces that are required to build servlets. These are javax.servlet and javax.servlet.http. They constitute the Servlet API. These packages are not part of the java core packages. Instead, they are standard extensions provided by the Tomcat.

The javax.servlet Package:

The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes the core interfaces that are provided in this package. The most significant of these is **Servlet.** All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.

ServletContext	Enables servlets to log events and access info about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the javax.servlet package:

Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

The Servlet Interface:

All servlets must implement the **Servlet** interface. It declares the **init(), service(),** and **destroy()** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by **Servlet** are shown in the following table.

Method	Description
void destroy()	Called when the servlet is unloaded.
ServletConfig getServletConfig()	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo()	Returns a string describing the servlet.
void init(ServletConfig sc) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from sc.
void service(ServletRequest req, ServletResponse) throws IOException, ServletException	Called to process a request from a client. The request from the client can be read from req. The response to the client can be written to res. An exception is generated if a servlet or IO problem occurs.

The ServletConfig Interface:

The **ServletConfig** interface is implemented by the servlet container. It allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are shown here:

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

The ServeltContext Interface:

The **ServletContext** interface is implemented by the servlet container. It enables servlets to obtain information about the environment. The methods of this interface are summarized below:

Method	Description
Object getAttribute(String attr)	Returns the values of the server attribute named attr.
String getServerInfo()	Returns the information about the server.
void setAttributes(String attr, Object val)	Sets the attribute specified by attr to the value passed in val.
void log(String s)	Writes s to the servlet log.

The ServletRequest Interface:

The **ServletRequest** interface is implemented by the servlet container. It enables a servlet to obtain information about a client request. Some of its methods are shown in the following table:

Method	Description
String getParameter(String pname)	Returns the value of the parameter named pname.
Enumeration getParameterNames()	Returns an enumeration of the parameter names for this request.
String[] getParameterValues(String pname)	Returns an array containing values associated with the parameter specified by pname.
String getProtocol()	Returns a description of the protocol.
String getContentType()	Returns the type of the request. A **null** value is returned if the type cannot be determined.
BufferedReader getReader() throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStrream()** has already been invoked for this request.
int getServerPort	Returns the port number.
String getRemoteHost()	Returns the client's host name.

The ServletResponse Interface:

The **ServletResponse** interface is implemented by the servlet container. It enables a servlet to formulate a response for a client. Its methods are summarized below:

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

void setContentSize(int size)	Sets the content length for the response to size.
void setContentType(String type)	Sets the content type for the response to type.

The following are the classes of the javax.servlet package:

The GenericServlet Class:

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

```
void log(String s)
void log(String s, throwable e)
```

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

The ServletInputStream Class:

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

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

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

The ServletOutputStream Class:

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

The Servlet Exception Classes:

javax.servlet defines two exceptions. The first is **ServeltException**, which indicates that a servlet problem has occurred. The second is **UnavailableException,** which extends **ServletException.** It indicates that a servlet is unavailable.

The javax.servlet.http Package:

The **javax.servlet.http** package contains a number of interfaces and classes that are commonly used by the servlet developers. Its functionality makes it easy to build servlets that work with HTTP requests and responses. The following table summarizes the core interfaces that are available in this package:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.

SPEC

HttpSessionBindingListener	Informs an object that it is bound to or unbound from a session.

The following table summarizes the core classes that are provided in this package. The most important of these is **HttpServlet**. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.
HttpSessionBindingEvent	Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

The HttpServletRequest Interface:

The **HttpServletRequest** interface is implemented by the servlet container. It enables a servlet to obtain information about a client request. The following table summarizes the methods implemented by this interface.

Method	Description
Cookie[] getCookies()	Returns an array of the cookies in this request.
String getMethod()	Returns the HTTP method for this request.
String getQueryString()	Returns any query string in the URL.
String getRemoteUser()	Returns the name of the user who issued this request.
String getRequestedSeesionId()	Returns the ID of the session.
String gerServletPath()	Returns the part of the URL that indicates the servlet.

The HttpServeltResponse Interface:

The **HttpServletResponse** interface is implemented the servlet container. It enables a servlet to formulate an HTTP response to a client. Servlet constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. Several important methods of this interface are summarized in the following table:

Method	Description
void addCookie()	Adds cookie to the HttpResponse.
void sendError(int c) throws IOException	Sends the error code c to the client.
void sendError(int c, String s) throws IOException	Sends the error code c and message s to the client.
void sendRedirect(String url) throws IOException	Redirects the client to the given URL.
void setStatus(int code)	Sets the status code for this response to code.
void setHeader(String field, String mesg)	Adds field to the header with value equal to mesg.

The HttpSession Interface:

The **HttpSession** interface is implemented by the servlet container. It enables a servlet to read and write the state information that is associated with an HTTP session. The following table

summarizes the several methods of this class. All of these methods throw an **IllegalStateException** if the session has already been invalidated.

Method	**Descrption**
Object getAttribute(String attr)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
long getCreationTime()	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
String getId()	Returns the session ID.
void invalidate()	Invalidates this session and removes it from the context.
void setAttribute(String s, Object val)	Associates the value passed in *val* with the attribute name passed in *attr*.

The Cookie Class:

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assumes that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store. A servlet can write a cookie to a user's machine via the addCookie() method of the HttpServletResponse interface. The names and values of cookies are stored on the user's machine. Some of the information that is saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and the path of the cookie.

The following table summarizes the several important methods of the **Cookie** class:

String getComment()	Returns the comment
String getDomain()	Returns the domain
Int getMaxAge()	Returns the age
String getName()	Returns the name
String getPath()	Returns the path
Boolean getSecure()	Returns true if the cookie is secure
Int getVersion()	Returns the version
Void setComment(String c)	Sets the comment to c
Void setDomain(String d)	Sets the domain to d
Void setPath(String p)	Sets the path to p
Void setSecure(boolean secure)	Sets the security flag to secure

The HttpServlet Class:

The HttpServlet class extends GenericServlet. It is commonly used when developing servlets that receive and process HTTP requests. Following are the methods used by **HttpServlet** class:

Method	Description
void doDelete(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletExcpetion	Handles an HTTP DELETE.
void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletExcpetion	Handles an HTTP GET.
void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletExcpetion	Handles an HTTP POST.
void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletExcpetion	Handles an HTTP PUT.
void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletExcpetion	Handles an HTTP TRACE.
void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletExcpetion	Called by the server when an HTTP request for this servlet. The arguments provide access to the HTTP request and response, respectively.

Reading Parameters:

The parameters for the servlets can be read in one of the two ways. They are:

1) Initialized Parameters
2) Servlet Parameters

Reading Initialized Parameters:

The initialized parameters are the parameters which are initialized in the web.xml file and they are not changeable in the entire servlet. These parameters are first written into the web.xml file by using <init-param> tag. This tag contains two more sub tags: first, <param-name>, which indicates the name of the parameter. Second tag is <param-value>, which contains the value for the name given. The following example shows how the initialized parameters can be set and get through servlets.

web.xml:

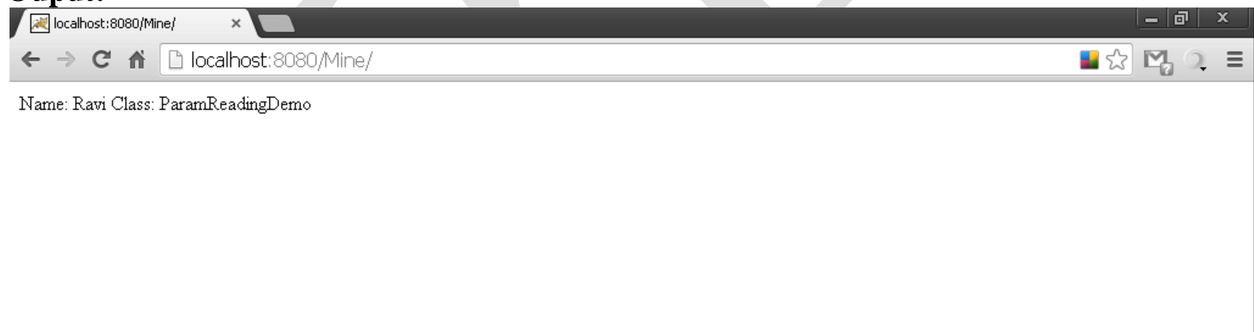
```
<web-app>
	<servlet>
		<servlet-name>InitParam</servlet-name>
		<servlet-class> InitParamDemo</servlet-class>
		<init-param>
			<param-name>name</param-name>
			<param-value>Ravi</param-value>
		</init-param>
		<init-param>
			<param-name>class</param-name>
			<param-value>ParamReadingDemo</param-value>
		</init-param>
```

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

InitParamDemo.java:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class InitParamDemo extends HttpServlet
{
        public void service(HttpServletRequest req, HttpServletResponse res) throws
IOException, ServletException
        {
                ServletConfig sc=getServletConfig();
                res.setContentType("text/html");
                PrintWriter out=res.getWriter();
                out.println("Name: "+sc.getInitParameter("name")+"\n");
                out.println("Class: "+sc.getInitParameter("class"));
        }
}
```

Ouput:



Reading Servlet Parameters:

The **ServletRequest** interface includes methods that allow us to read the names and values of the parameters that are included in a client request. These requests are usually come from a HTML page. The following example illustrates the usage of servlet parameters. The example contains two files. One is a web page, which sends the data to the server. And other one is a servlet which is defined for handling of the parameters sent by the client.

LoginPage.html:

```
<html>
<head>
<title>Login Form</title>
<style type="text/css">
```

```
body
{
text-align:center;
}
</style>
<body>
<form name="form1" action="LoginServlet" method="get">
<h1>Login Form</h1>
<table>
<tr>
<td>User Name:</td>
<td><input type="text" name="uname">
</td>
</tr>
<tr>
<td>Password:</td>
<td><input type="password" name="pwd">
</td>
</tr>
<tr>
<td><input type="submit" value="Login"></td>
<td><input type="reset" value="Clear"></td>
</tr>
</table>
</form>
</body>
</html>
```

LoginServlet.java:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class LoginServlet extends HttpServlet
{
	public void doGet(HttpServletRequest req,HttpServletResponse res) throws IOException,
ServletException
	{
		String name=req.getParameter("uname");
		String pwd=req.getParameter("pwd");
		res.setContentType("text/html");
		PrintWriter out=res.getWriter();
		if(name.equals("Chythu")&&pwd.equals("chythu536"))
		{
			out.println("Login Successfully Completed");
			out.println("<b><br>");
			out.println("Welcome Mr. "+name);
			out.println("</b>");
```


SREC

```
        }
        else
            out.println("Login Failed");
        out.close();
    }
}
```

web.xml:

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

Handling HTTP Requests and Responses:

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete( ), doGet( ), doHead( ), doOptions( ), doPost( ), doPut( ),** and **doTrace( ).**

Handling HTTP GET Request:

The **GET** requests from the form submitted by the user are handled by the servlet with the help of **doGet()** method which is provided by the **HttpServlet** class. Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **GetFruit.html**, and a servlet is defined in **FruitServlet.java.** The HTML page defines a form that contains a select element and a submit button.

GetFruit.html:

```
<html>
    <head>
        <title> Do Get Demo</title>
    </head>
    <body>
        <form name="form1" action="Myserv1" method="get">
            <b> Fruit</b>
            <select name="fruit" >
                <option value="banana"> Banana</option>
                <option value="mango"> Mango</option>
                <option value="orange"> Orange</option>
                <option value="apple"> Apple</option>
```

```
                </select>
                <input type="submit" value="submit">
            </form>
        </body>
</html>
```

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

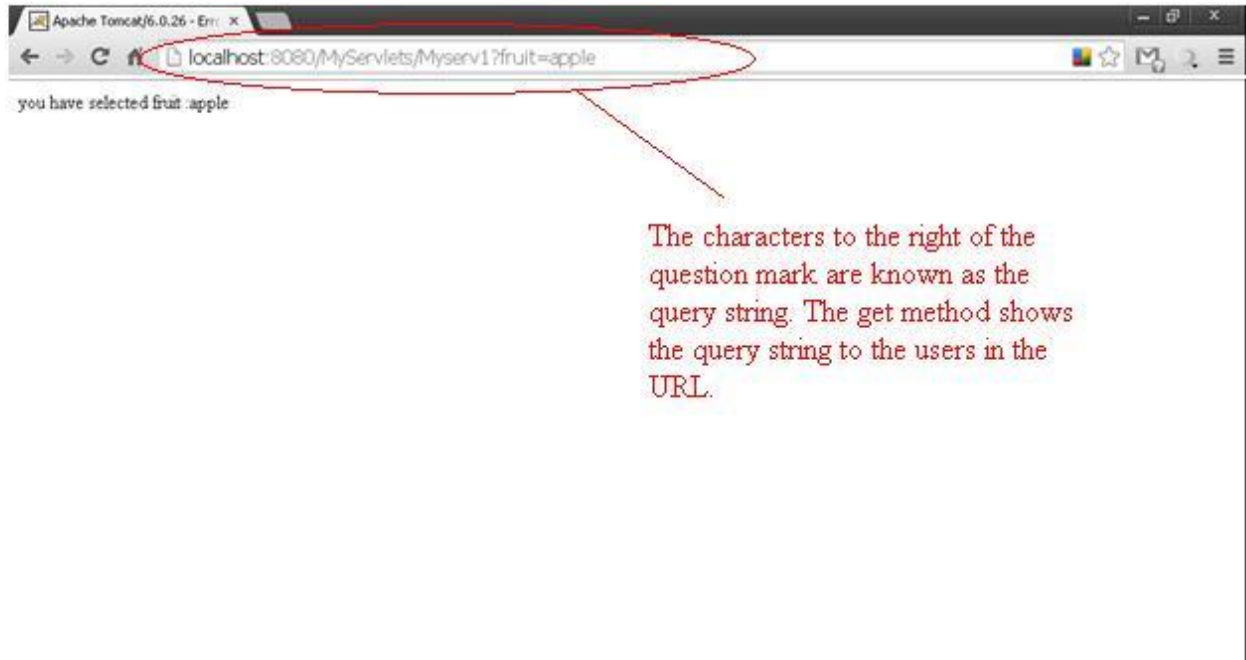
FruitServlet.java:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FruitServlet extends HttpServlet
{
	public void doGet(HttpServletRequest req,HttpServletResponse res) throws IOException, ServletException
	{
		String Fruit=req.getParameter("fruit");
		res.setContentType("text/html");
		PrintWriter out=res.getWriter();
		out.println("you have selected fruit   :" +Fruit);
		out.close();
	}
}
```

Compile the servlet. Next, copy the .class file the web application's classes folder and update the web.xml file. Then perform these steps to run this servlet.

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

SREC



Handling HTTP POST Requests:

The **POST** requests from the form submitted by the user are handled by the servlet with the help of **doPost()** method which is provided by the **HttpServlet** class. Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **PostFruit.html**, and a servlet is defined in **FruitServlet.java.** The HTML page defines a form that contains a select element and a submit button.

PostFruit.html:

```
<html>
        <head>
                <title> Do Post Demo</title>
        </head>
        <body>
                <form name="form1" action="Myserv1" method="post">
                        <b> Fruit</b>
                        <select name="fruit" >
                                <option value="banana"> Banana</option>
                                <option value="mango"> Mango</option>
                                <option value="orange"> Orange</option>
                                <option value="apple"> Apple</option>
                        </select>
                        <input type="submit" value="submit">
                </form>
        </body>
</html>
```

The source code for **FruitServlet.java** is shown below. The doPost() method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the

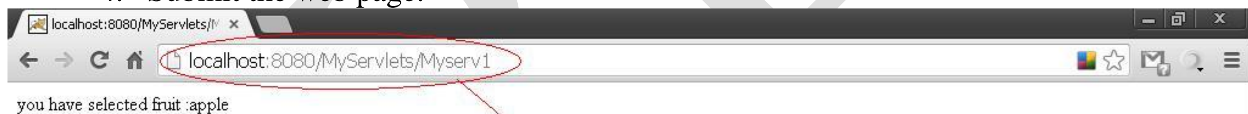
getParameter() method of HttpServletRequest to obtain the selection that was made by the user. A response is then formulated.

FruitServlet.java:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FruitServlet extends HttpServlet
{
	public void doPost(HttpServletRequest req,HttpServletResponse res) throws IOException, ServletException
	{
		String Fruit=req.getParameter("fruit");
		res.setContentType("text/html");
		PrintWriter out=res.getWriter();
		out.println("you have selected fruit   :" +Fruit);
		out.close();
	}
}
```

Compile the servlet. Next, copy the .class file the web application's classes folder and update the web.xml file. Then perform these steps to run this servlet.

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



Using Cookies:

A *cookie* is small amount information stored on a client machine and contains state information. Cookies are valuable for tracking user activities. This example contains two files as below:

CookieDemo.html→ Allows a user to specify a value for the cookie named MyCookie.
CookieDemo.java→ Processes the submission of the CookieDemo.html.

CookieDemo.html:

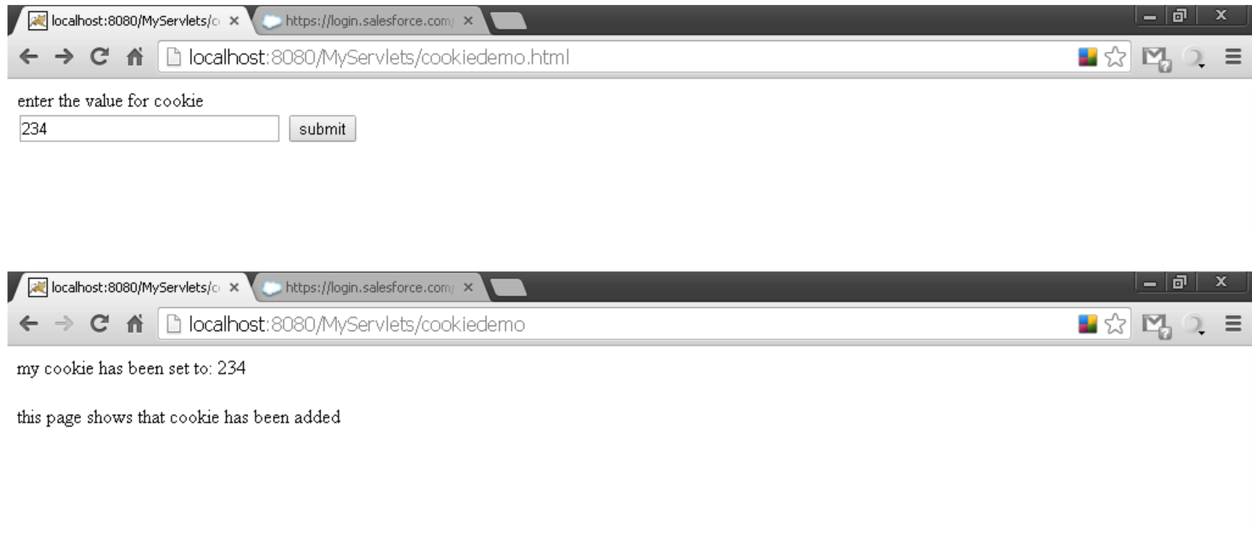
```
<html>
<body>
<form name="form1" method="post" action="CookieDemo">
<b>Enter the value for cookie</b>
<input type=text name="txt_data" size=30 value="">
<input type=submit value="Add Cookie">
</form>
</body>
</html>
```

CookieDemo.java:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieDemo extends HttpServlet
{
	public void doPost(HttpServletRequest req,HttpServletResponse res)throws
ServletException, IOException
	{
		String data=req.getParameter("txt_data");
		Cookie cookie=new Cookie("Mycookie",data);
		res.addCookie(cookie);
		res.setContentType("text/html");
		PrintWriter out=res.getWriter();
		out.println("my cookie has been set to: ");
		out.println(data);
		out.println("<br>");
		out.println("this page shows that cookie has been added");
		out.close();
	}
}
```

Compile the servlet. Next, copy the .class file the web application’s classes folder and update the web.xml file. Then perform these steps to run this servlet.

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



Session Tracking:

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism. A session can be created via the **getSession( )** method of the **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects.

The **setAttribute( ), getAttribute( ). getAttributeNames( ),** and **removeAttribute( )** methods of **HttpSession** manage these bindings. The following example illustrates how to use session state. The **getSession( )** method gets the current session. A new session is created if one does not already exist. The getAttribute( ) method is called to obtain the object that is bound to the name “cnt”.

Sessions.html:

```
<html>
<head>
<title>Session Demo</title>
</head>
<body>
<form name="form1" action="serv3" >
<b> Session Demo</b><br>
<input type="submit" value="Go to My Session Demo">
</form>
</body>
</html>
```

SessionDemo.java:

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

SREC

```
public void doGet(HttpServletRequest req, HttpServletResponse res)throws IOException,ServletException
	{
		res.setContentType("text/html");
		HttpSession session=req.getSession();
		String heading;
		Integer cnt=(Integer)session.getAttribute("cnt");
		if(cnt==null)
		{
			cnt=new Integer(0);
			heading="Welcome for the first time";
		}
		else
		{
			heading="Welcome once again";
			cnt=new Integer(cnt.intValue()+1);
		}
		session.setAttribute("cnt",cnt);
		PrintWriter out=res.getWriter();
		out.println("<html>");
		out.println("<body>");
out.println("<h1>"+heading);
out.println("The number of previous accesses: "+cnt);
out.println("</body>");
out.println("</html>");
	}
}
```

Connecting to the Database using JDBC:

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database independent connectivity between the Java programming language, and a wide range of databases. The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as:

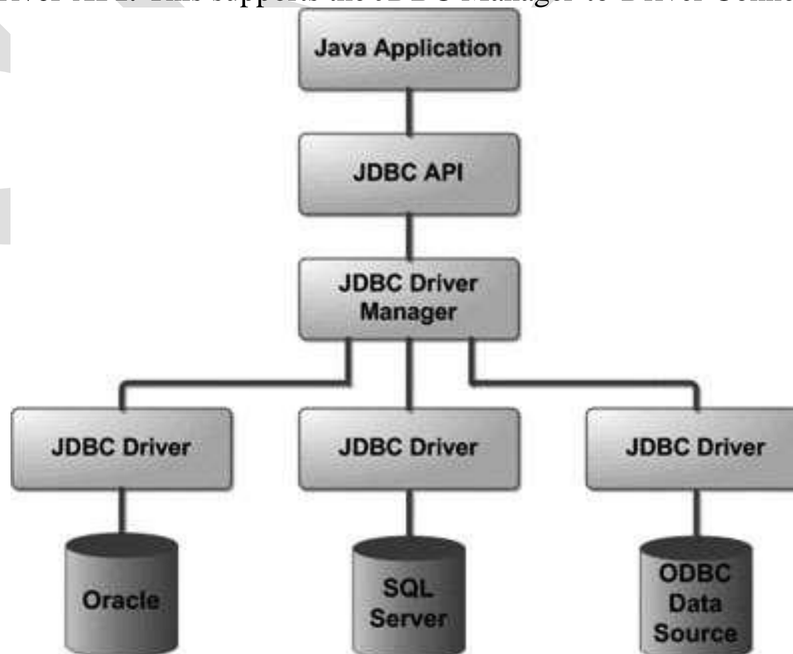
- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data. JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

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.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.



Above figure is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application:

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.

What is JDBC Driver?

The JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java. The *java.sql* package that ships with JDK contains various classes with their behavior defined and their actual implementations are done in third-party drivers. Third party vendors implement the *java.sql.Driver* interface in their database driver.

JDBC Drivers Types:

The JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which are listed below:

1) Type-1: JDBC-ODBC Bridge Driver
2) Type-2: JDBC Native API
3) Type-3: JDBC-Net Pure Java
4) Type-4: 100% Pure Java

1) Type-1: JDBC-ODBC Bridge Driver:

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

2) Type-2: JDBC Native API:

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

3) Type-3: JDBC-Net Pure Java:

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

SREC

4) Type-4: 100%Pure Java:

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

Common JDBC Components:

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manage objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

The Basic Steps to Connect to a Database Using Servlet:

There are 5 basic steps to be followed to connect to a database using servlets. They are as follows:

1) Establish a **Connection**
2) Create JDBC **Statements**
3) Execute **SQL** Statements
4) GET **ResultSet**
5) **Close** connections

Example:

Login.html:

```
<html>
   <head>
      <title>Login Form</title>
</head>
   <body>
   <center>
      <h1>Login Form</h1>
      <form action="login" method="post">
         <table>
            <tr>
               <td>User name:</td>
               <td><input type="text" name="uname"/></td>
            </tr>
            <tr>
               <td>Password:</td>
               <td><input type="password" name="pwd"/></td>
            </tr>
            <tr>
               <td><input type="submit" value="Login"></td>
               <td><input type="reset" value="Clear"/></td>
            </tr>
         </table>
      </form>
   </center>
   </body>
</html>
```

LoginSrv.java:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
public class LoginSrv extends HttpServlet
{
public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException
{
try
{
response.setContentType("text/html");
PrintWriter out=response.getWriter();
String un=request.getParameter("uname");
String pwd=request.getParameter("pwd");
out.println("<head><title>SCCE</title></head>");
```

SREC

```
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/scce", "root", "");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from Login where uname='"+un+"' and pwd='"+pwd+"'");
if(rs.next())
out.println("<h1>Welcome "+rs.getString(1)+", You are logged in successfully</h1>");
else
out.println("<h1>Login Failed</h1>");
}
catch(Exception se)
{
se.printStackTrace();
}
}
```

web.xml:

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

SREC