

Java Servlets: Servlet Life Cycle, Basic Servlet Structure, request methods, passing initialization parameters from web.xml, Handling the client request form data, Generating HTTP Response, Request dispatching and State Management techniques.

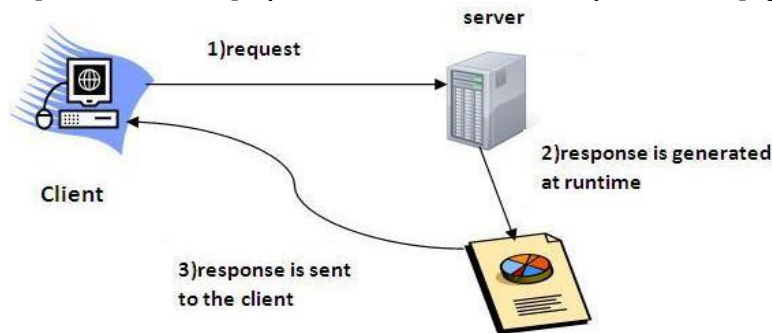
Java Server Pages: Expressions, Scripting elements, Page Directives, Actions, JSP Objects, Handling Exceptions, MVC Flow of Control, Accessing Ms Access, My SQL and Oracle databases using Servlets and JSP.

Chapter-I

Servlets provide a component-based, platform-independent method for building Web-based applications, without the performance limitations of CGI(common gateway interface) programs. Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases.

Servlet can be described in many ways, depending on the context.

- Servlet is a technology i.e. used to create web application.
- Servlet is an API that provides many interfaces and classes including documentations.
- Servlet is an interface that must be implemented for creating any servlet.
- Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
- Servlet is a web component that is deployed on the server to create dynamic web page.

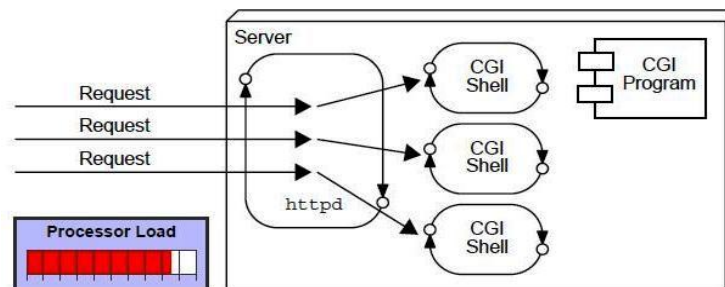


What is web application?

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML. The web components typically execute in Web Server and respond to HTTP request.

CGI(Common Gateway Interface)

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

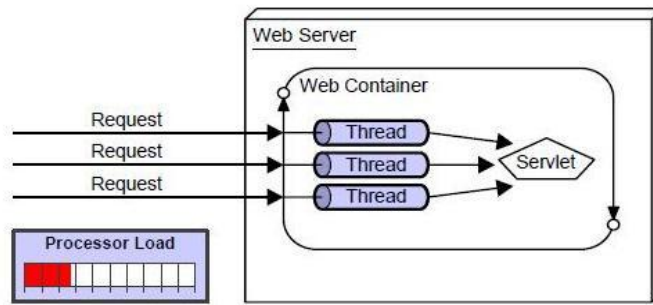


Disadvantages of CGI

There are many problems in CGI technology:

1. If number of clients increases, it takes more time for sending response.
2. For each request, it starts a process and Web server is limited to start processes.
3. It uses platform dependent language e.g. C, C++, perl.

Advantage of Servlet



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

1. better performance: because it creates a thread for each request not process.
2. Portability: because it uses java language.
3. Robust: Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.
4. Secure: because it uses java language..

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

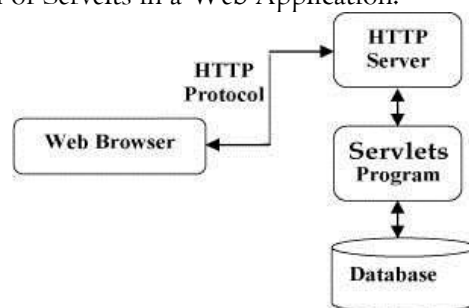
Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.

- Performance is significantly better.
- Servlets execute within the address space of a Web server. It is not necessary to create a separate process to handle each client request.
- Servlets are platform-independent because they are written in Java.
- Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. So servlets are trusted.
- The full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Servlets Architecture:

Following diagram shows the position of Servlets in a Web Application.



Servlets Tasks:

Servlets perform the following major tasks:

- Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
- Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
- Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
- Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.

- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlets Packages:

Java Servlets are Java classes run by a web server that has an interpreter that supports the Java Servlet specification. Servlets can be created using the `javax.servlet` and `javax.servlet.http` packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.

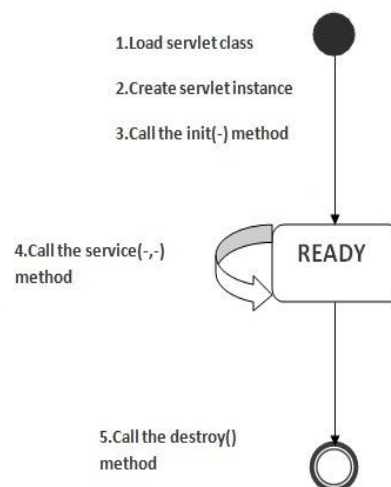
These classes implement the Java Servlet and JSP specifications. At the time of writing this tutorial, the versions are Java Servlet 2.5 and JSP 2.1.

Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's Classpath, you can compile servlets with the JDK's Java compiler or any other current compiler.

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.

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

- The servlet is initialized by calling the `init()` method.
- The servlet calls `service()` method to process a client's request.
- The servlet is terminated by calling the `destroy()` method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

The `init()` method :

The `init` method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the `init` method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to `doGet` or `doPost` as appropriate. The `init()` method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {  
    // Initialization code...  
}
```

The service() method :

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method:

```
public void service(ServletRequest request,  
    ServletResponse response)  
    throws ServletException, IOException{  
}
```

The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request,  
    HttpServletResponse response)  
    throws ServletException, IOException {  
    // Servlet code  
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request,  
    HttpServletResponse response)  
    throws ServletException, IOException {  
    // Servlet code  
}
```

The destroy() method :

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

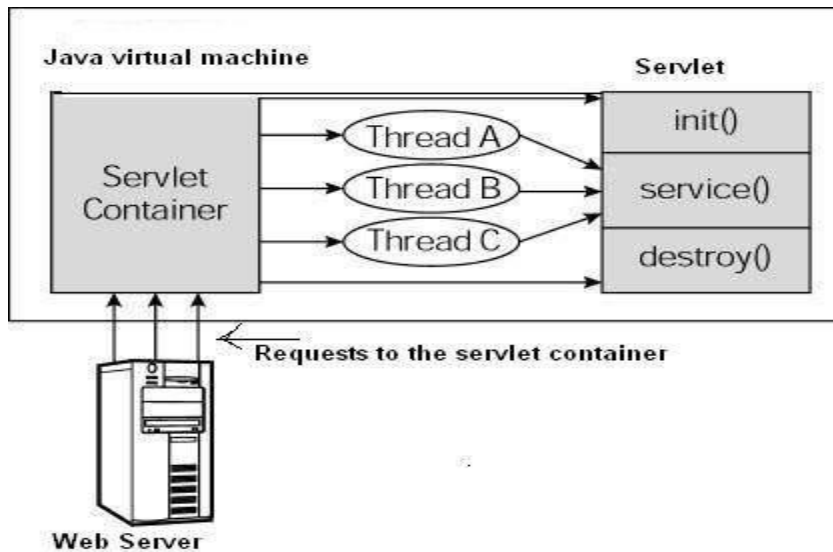
After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy() {  
    // Finalization code...  
}
```

Architecture Diagram:

The following figure depicts a typical servlet life-cycle scenario.

- First the HTTP requests coming to the server are delegated to the servlet container.
- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



Basic Servlet Structure

Basic servlet that handles GET requests. GET requests, for those unfamiliar with HTTP, are the usual type of browser requests for Web pages. A browser generates this request when the user enters a URL on the address line, follows a link from a Web page, or submits an HTML form that either does not specify a METHOD or specifies METHOD="GET". Servlets can also easily handle POST requests, which are generated when someone submits an HTML form that specifies METHOD="POST".

ServletTemplate.java

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

public class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        // Use "request" to read incoming HTTP headers
        // (e.g., cookies) and query data from HTML forms.

        // Use "response" to specify the HTTP response status
        // code and headers (e.g. the content type, cookies).
        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser.
    }
}
```

To be a servlet, a class should extend `HttpServlet` and override `doGet` or `doPost`, depending on whether the data is being sent by GET or by POST. If you want a servlet to take the same action for both GET and POST requests, simply have `doGet` call `doPost`, or vice versa.

Both `doGet` and `doPost` take two arguments: an `HttpServletRequest` and an `HttpServletResponse`. The `HttpServletRequest` has methods by which you can find out about incoming information such as form (query) data, HTTP request headers, and the client's hostname. The `HttpServletResponse` lets you specify outgoing information such as HTTP status codes (200, 404, etc.) and response headers (Content-Type, Set-Cookie, etc.). Most importantly, it lets you obtain a `PrintWriter` with which you send the document content back to the client. For simple servlets, most of the effort is spent in `println` statements that generate the desired page. Form data, HTTP request headers, HTTP responses, and cookies are all discussed in the following sections. Since `doGet` and `doPost` throw two exceptions, you are required to include them in the declaration. Finally, you must import classes in `java.io` (for `PrintWriter`, etc.), `javax.servlet` (for `HttpServlet`, etc.), and `javax.servlet.http` (for `HttpServletRequest` and `HttpServletResponse`).

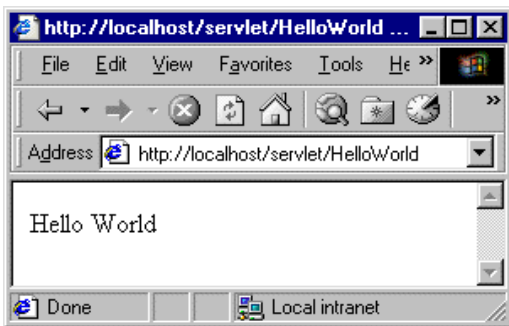
A Servlet That Generates Plain Text

simple servlet that outputs plain text, with the output shown in Figure. Before we move on, it is worth spending some time reviewing the process of installing, compiling, and running this simple servlet.

HelloWorld.java

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

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```



ServletRequest Interface:

When a browser requests for a web page, it sends lot of information to the web server which can not be read directly because this information travel as a part of header of HTTP request.

An object of ServletRequest is used to provide the client request information to a servlet such as content type, content length, parameter names and values, header informations, attributes etc.

Methods of ServletRequest interface

There are many methods defined in the ServletRequest interface. Some of them are as follows:

| Method | Description |
|--|--|
| public String getParameter(String name) | is used to obtain the value of a parameter by name. |
| public String[] getParameterValues(String name) | returns an array of String containing all values of given parameter name. It is mainly used to obtain values of a Multi select list box. |
| java.util.Enumeration getParameterNames() | returns an enumeration of all of the request parameter names. |
| public int getContentLength() | Returns the size of the request entity data, or -1 if not known. |
| public String getCharacterEncoding() | Returns the character set encoding for the input of this request. |
| public String getContentType() | Returns the Internet Media Type of the request entity data, or null if not known. |
| public ServletInputStream getInputStream() throws IOException | Returns an input stream for reading binary data in the request body. |
| public abstract String getServerName() | Returns the host name of the server that received the request. |
| public int getServerPort() | Returns the port number on which this request was received. |

HTTP Protocol:

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web (i.e. internet) since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extensions of its request methods, error codes, and headers.

Basically, HTTP is a TCP/IP based communication protocol, that is used to deliver data (HTML files, image files, query results, etc.) on the World Wide Web. The default port is TCP 80, but other ports can be used as well. It provides a standardized way for computers to communicate with each other. HTTP specification specifies how clients' request data will be constructed and sent to the server, and how the servers respond to these requests.

Basic Features

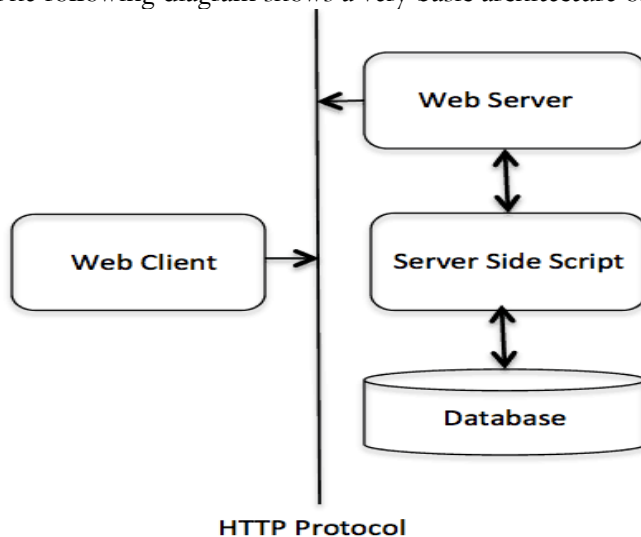
There are three basic features that make HTTP a simple but powerful protocol:

- HTTP is connectionless: The HTTP client, i.e., a browser initiates an HTTP request and after a request is made, the client disconnects from the server and waits for a response. The server processes the request and re-establishes the connection with the client to send a response back.
- HTTP is media independent: It means, any type of data can be sent by HTTP as long as both the client and the server know how to handle the data content. It is required for the client as well as the server to specify the content type using appropriate MIME-type.
- HTTP is stateless: As mentioned above, HTTP is connectionless and it is a direct result of HTTP being a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each other. Due to this nature of the protocol, neither the client nor the browser can retain information between different requests across the web pages.

HTTP/1.0 uses a new connection for each request/response exchange, where as HTTP/1.1 connection may be used for one or more request/response exchanges.

Basic Architecture

The following diagram shows a very basic architecture of a web application and depicts where HTTP sits:



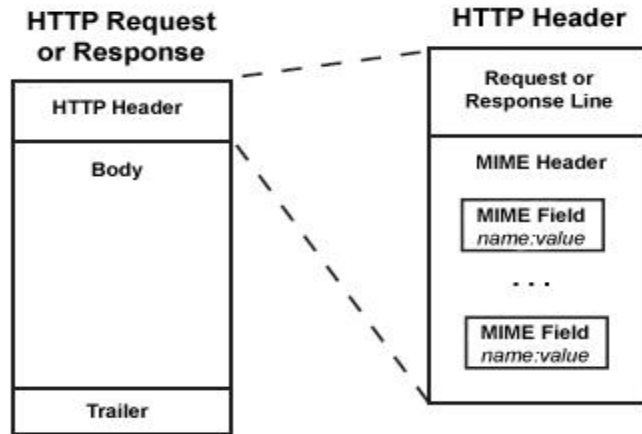
The HTTP protocol is a request/response protocol based on the client/server based architecture where web browsers, robots and search engines, etc. act like HTTP clients, and the Web server acts as a server.

Client

The HTTP client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a TCP/IP connection.

Server

The HTTP server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content.



HTTP Request

```

      Method           Request-URI           Protocol version
Request line  { PUT /hr/ergonomics/posture.doc HTTP/1.1
Headers       { Host: www.example.com:8080
               { Content-Length: 1234
Empty line
Body
(optional)   { Body must include the number of
               { characters specified in the content
               { length header...

```

HTTP Response

```

      Protocol version  Status code  Status description
Status line  { HTTP/1.1 200 OK
Headers      { Date: Sun, 29 Jul 2001 15:24:17 GMT
               { Content-Length: 1234
Empty line
Body
(optional)   { Body must include the number of
               { characters specified in the content
               { length header...

```

Methods to read HTTP Header:

There are following methods which can be used to read HTTP header in your servlet program. These methods are available with *HttpServletRequest* object.

| S.N. | Method & Description |
|------|---|
| 1 | Cookie[] getCookies() Returns an array containing all of the Cookie objects the client sent with this request. |
| 2 | Enumeration getAttributeNames() Returns an Enumeration containing the names of the attributes available to this request. |
| 3 | Enumeration getHeaderNames() Returns an enumeration of all the header names this request contains. |
| 4 | Enumeration getParameterNames() Returns an Enumeration of String objects containing the names of the parameters contained in this request. |
| 5 | HttpSession getSession() Returns the current session associated with this request, or if the request does not have a session, creates one. |
| 6 | 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. |
| 7 | Locale getLocale() Returns the preferred Locale that the client will accept content in, based on the Accept-Language header. |
| 8 | Object getAttribute(String name) Returns the value of the named attribute as an Object, or null if no attribute of the given name exists. |
| 9 | ServletInputStream getInputStream() Retrieves the body of the request as binary data using a ServletInputStream. |
| 10 | String getAuthType() Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the JSP was not protected. |
| 11 | String getCharacterEncoding() Returns the name of the character encoding used in the body of this request. |
| 12 | String getContentType() Returns the MIME type of the body of the request, or null if the type is not known. |
| 13 | String getContextPath() Returns the portion of the request URI that indicates the context of the request. |
| 14 | String getHeader(String name) Returns the value of the specified request header as a String. |
| 15 | String getMethod() Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT. |
| 16 | String getParameter(String name) Returns the value of a request parameter as a String, or null if the parameter does not exist. |
| 17 | String getPathInfo() Returns any extra path information associated with the URL the client sent when it made this request. |
| 18 | String getProtocol() Returns the name and version of the protocol the request. |
| 19 | String getQueryString() Returns the query string that is contained in the request URL after the path. |
| 20 | String getRemoteAddr() Returns the Internet Protocol (IP) address of the client that sent the request. |
| 21 | String getRemoteHost() Returns the fully qualified name of the client that sent the request. |
| 22 | String getRemoteUser() Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. |
| 23 | String getRequestURI() Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request. |
| 24 | String getRequestedSessionId() Returns the session ID specified by the client. |
| 25 | String getServletPath() Returns the part of this request's URL that calls the JSP. |
| 26 | String[] getParameterValues(String name) Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist. |
| 27 | boolean isSecure() Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS. |
| 28 | int getContentLength() Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known. |

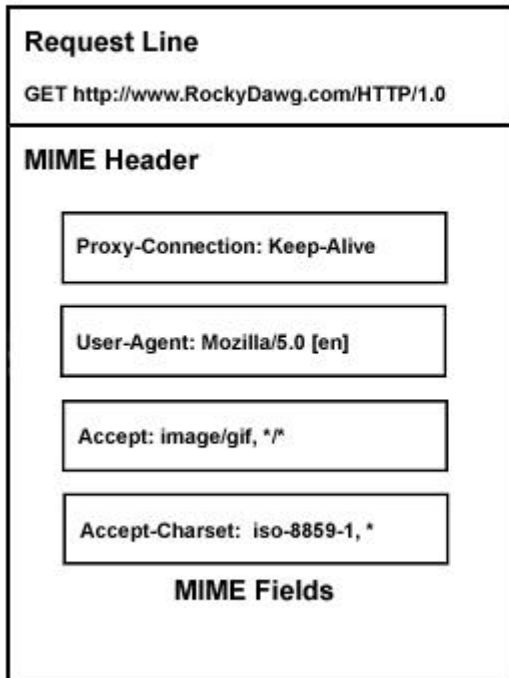
| | |
|----|--|
| 29 | <code>int getIntHeader(String name)</code> Returns the value of the specified request header as an int. |
| 30 | <code>int getServerPort()</code> Returns the port number on which this request was received. |

Following is the important header information which comes from browser side and you would use very frequently in web programming:

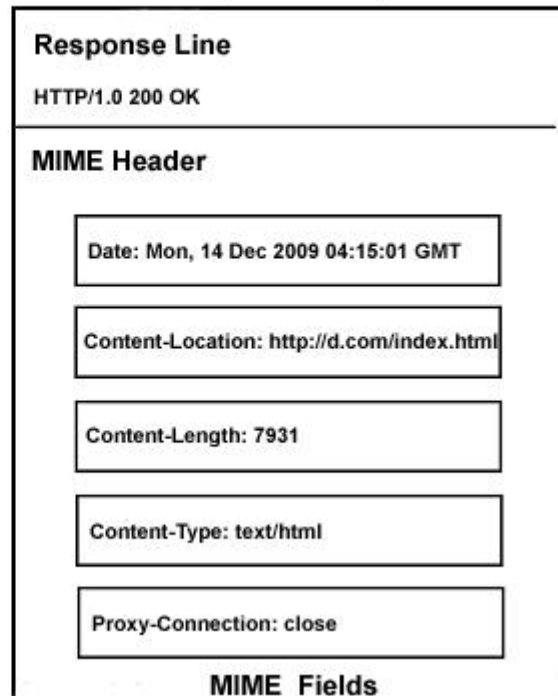
| Header | Description |
|---------------------|---|
| Accept | This header specifies the MIME types that the browser or other clients can handle. Values of image/png or image/jpeg are the two most common possibilities. |
| Accept-Charset | This header specifies the character sets the browser can use to display the information. For example ISO-8859-1. |
| Accept-Encoding | This header specifies the types of encodings that the browser knows how to handle. Values of gzip or compress are the two most common possibilities. |
| Accept-Language | This header specifies the client's preferred languages in case the servlet can produce results in more than one language. For example en, en-us, ru, etc. |
| Authorization | This header is used by clients to identify themselves when accessing password-protected Web pages. |
| Connection | This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or other browser to retrieve multiple files with a single request. A value of Keep-Alive means that persistent connections should be used |
| Content-Length | This header is applicable only to POST requests and gives the size of the POST data in bytes. |
| Cookie | This header returns cookies to servers that previously sent them to the browser. |
| Host | This header specifies the host and port as given in the original URL. |
| If-Modified-Since | This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a code, 304 which means Not Modified header if no newer result is available. |
| If-Unmodified-Since | This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date. |
| Referer | This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2. |
| User-Agent | This header identifies the browser or other client making the request and can be used to return different content to different types of browsers. |

Example HTTP Request and Response:

HTTP Header: Request Example



HTTP Header: Response Example



Example of ServletRequest to display the name of the user

In this example, we are displaying the name of the user in the servlet. For this purpose, we have used the `getParameter` method that returns the value for the given request parameter name.

index.html

```
<form action="welcome" method="get">
Enter your name<input type="text" name="name"><br>
<input type="submit" value="login">
</form>
```

DemoServ.java

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

public class DemoServ extends HttpServlet{
    public void doGet(HttpServletRequest req,HttpServletResponse res)
    throws ServletException,IOException
    {
        res.setContentType("text/html");
        PrintWriter pw=res.getWriter();

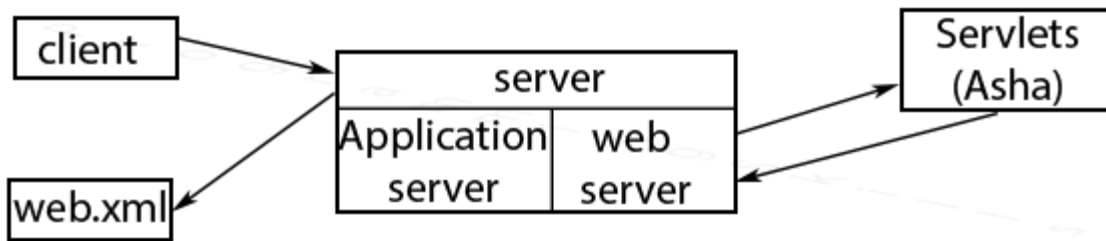
        String name=req.getParameter("name");//will return value
        pw.println("Welcome "+name);
        pw.close();
    }
}
```

Web.xml in Servlets

web.xml:

1. Whenever client makes a request to a servlet that request is received by server and server goes to a predefined file called web.xml for the details about a servlet.
2. web.xml file always gives the details of the servlets which are available in the server.

3. If the server is not able to find the requested servlet by the client then server generates an error (resource not found) [A resource is a program which resides in server].
4. If the requested servlet is available in web.xml then server will go to the servlet, executes the servlet and gives response back to client.



Every web.xml will contain the following entries:

```
<web-app>
```

```
<servlet>
```

```
<servlet-name>Asha</servlet-name>
```

```
<servlet-class>First</servlet-class>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
<servlet-name>Asha</servlet-name>
```

```
<url-pattern>Krishna</url-pattern>
```

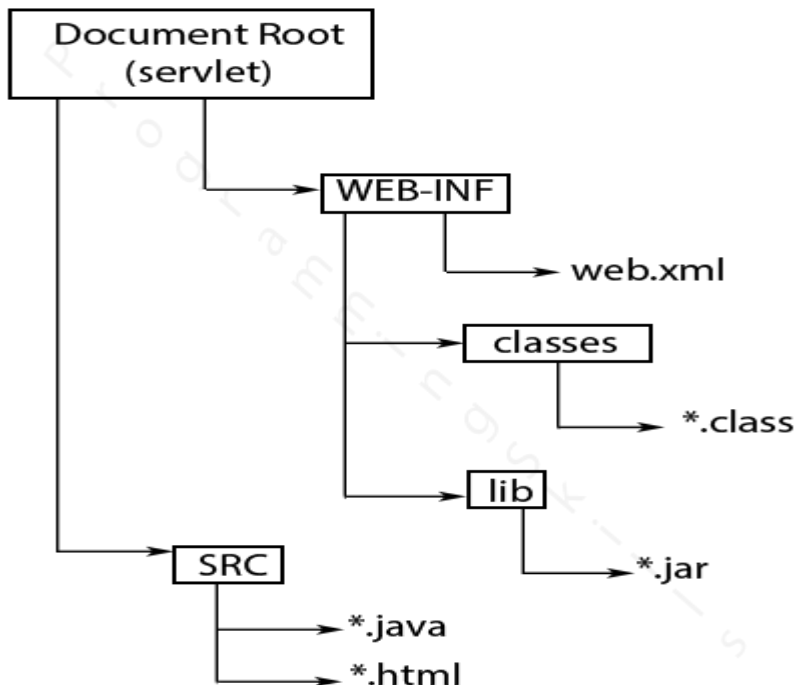
```
</servlet-mapping>
```

```
</web-app>
```

- While server is executing web.xml control goes to <url-pattern> of <servlet-mapping> tag. If the requested url and <url-name> web.xml is same then control goes to <servlet-class> tag of <servlet> and takes the <servlet-name> and executes.
- If the <url-pattern> is not matching, server generates an error.

How to execute the servlets:

In order to execute a servlet we must follow the following directory structure:



Steps for DEVELOPING a servlet:

1. Import javax.servlet.*, javax.servlet.http.* and other packages if required.
2. Choose user defined class.
3. Whichever class we have chosen in step-2 must extend either GenericServlet or HttpServlet.
4. Override the life cycle methods if required.

FLOW OF EXECUTION in a servlet:

1. Client makes a request. The general form of a request is http://(IP address or DNS [Domain Naming Service] name of the machine where server is installed) : (port number of the server) / (Document root) : (Resource name).
For example:
http://localhost:7001/DateSer/suman
2. Server receives the request.
3. Server will scan web.xml (contains declarative details) if the requested resource is not available in web.xml server generates an error called resource not available otherwise server goes to a servlet.
4. Server will call the servlet for executing.
5. Servlet will execute in the context of server.
6. While server is executing a servlet, server loads an object of servlet class only once (by calling default constructor).
7. After loading the servlet, the servlet will call init () method only once to perform one time operations.
8. After completion of init () method, service () method will be called each and every time. As long as we make number of requests only service () method will be called to provide business logic.
9. Servlet will call destroy () method either in the case of servlet is removed or in the case of server is closed.

HOW TO EXECUTE a servlet:

1. Prepare a directory structure.
2. Write a servlet program save it into either document root or document root\SRC.
3. Compile a servlet by setting a classpath. For Tomcat: Set classpath=
4. Copy *.class file into document root/WEB-INF/classes folder and write web.xml file.
5. Start the server and copy document root into:
6. Open the browser and pass a request or url

Passing initialization parameters from web.xml

ServletConfig Interface

- An object of ServletConfig is created by the web container for each servlet. This object can be used to get configuration information from web.xml file.
- If the configuration information is modified from the web.xml file, we don't need to change the servlet. So it is easier to manage the web application if any specific content is modified from time to time.

Advantage of ServletConfig

The core advantage of ServletConfig is that you don't need to edit the servlet file if information is modified from the web.xml file.

Methods of ServletConfig interface

- public String getInitParameter(String name):Returns the parameter value for the specified parameter name.
- public Enumeration getInitParameterNames():Returns an enumeration of all the initialization parameter names.
- public String getServletName():Returns the name of the servlet.
- public ServletContext getServletContext():Returns an object of ServletContext.

How to get the object of ServletConfig

getServletConfig() method of Servlet interface returns the object of ServletConfig.

Syntax of getServletConfig() method

```
public ServletConfig getServletConfig();
```

Example of getServletConfig() method

```
ServletConfig config=getServletConfig();  
//Now we can call the methods of ServletConfig interface
```

Syntax to provide the initialization parameter for a servlet

The init-param sub-element of servlet is used to specify the initialization parameter for a servlet.

```
<web-app>  
<servlet>  
.....  
  
<init-param>  
<param-name>parametername</param-name>  
<param-value>parametervalue</param-value>  
</init-param>
```

```
.....  
</servlet>  
</web-app>
```

Example of ServletConfig to get initialization parameter

In this example, we are getting the one initialization parameter from the web.xml file and printing this information in the servlet.

DemoServlet.java

```
import java.io.*;  
import javax.servlet.*;  
import javax.servlet.http.*;  
  
public class DemoServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
  
        ServletConfig config=getServletConfig();  
        String driver=config.getInitParameter("driver");  
        out.print("Driver is: "+driver);  
  
        out.close();  
    }  
}
```

web.xml

```
<web-app>  
  
    <servlet>  
        <servlet-name>DemoServlet</servlet-name>  
        <servlet-class>DemoServlet</servlet-class>  
  
        <init-param>  
            <param-name>driver</param-name>  
            <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>  
        </init-param>  
  
    </servlet>  
  
    <servlet-mapping>  
        <servlet-name>DemoServlet</servlet-name>
```

```
<url-pattern>/servletI</url-pattern>
</servlet-mapping>
```

```
</web-app>
```

Handling Client Request form data:

- Whenever we want to send an input to a servlet that input must be passed through html form.
- An html form is nothing but various controls are inherited to develop an application.
- Every form will accept client data and it must send to a servlet which resides in server side.
- Since html is a static language which cannot validate the client data. Hence, in real time applications client data will be accepted with the help of html tags by developing form and every form must call a servlet.

A typical scenario is the user fills in fields of a form and submits it. The server will process the request based on the submitted data, and send response back to the client.

To create a form in HTML we need to use the following tags:

- `<form>`: to create a form to add fields in its body.
- `<input>`, `<select>`, `<textarea>`...: to create form fields like text boxes, dropdown list, text area, check boxes, radio buttons,... and submit button.

To make the form works with Java servlet, we need to specify the following attributes for the `<form>` tag:

- **method="post"**: to send the form data as an HTTP POST request to the server. Generally, form submission should be done in HTTP POST method.
- **action="URL of the servlet"**: specifies relative URL of the servlet which is responsible for handling data posted from this form.

Reading Form Data using Servlet:

Servlets handles form data parsing automatically using the following methods depending on the situation:

- **getParameter()**: You call `request.getParameter()` method to get the value of a form parameter.
- **getParameterValues()**: Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
- **getParameterNames()**: Call this method if you want a complete list of all parameters in the current request.

HTML code of a login form:

```
<form name="loginForm" method="post" action="LoginServlet">
  Username: <input type="text" name="username"/> <br/>
  Password: <input type="password" name="password"/> <br/>
  <input type="submit" value="Login" />
</form>
```

The servlet class to process the login form:

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

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginServlet extends HttpServlet {

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

        // read form fields
        String username = request.getParameter("username");
        String password = request.getParameter("password");
```



```
System.out.println("username: " + username);
System.out.println("password: " + password);

// do some processing here...

// get response writer
PrintWriter writer = response.getWriter();

// build HTML code
String htmlResponse = "<html>";
htmlResponse += "<h2>Your username is: " + username + "<br/>";
htmlResponse += "Your password is: " + password + "</h2>";
htmlResponse += "</html>";

// return response
writer.println(htmlResponse);

}
}
```

1) Handling text field and password field

HTML code:

Username: <input type="text" name="username"/>
Password: <input type="password" name="password"/>

Java code in servlet:

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

System.out.println("username is: " + username);
System.out.println("password is: " + password);
```

2) Handling checkbox field

HTML code:

Speaking language:
<input type="checkbox" name="language" value="english" />English
<input type="checkbox" name="language" value="french" />French

Java code in servlet:

```
String languages[] = request.getParameterValues("language");

if (languages != null) {
    System.out.println("Languages are: ");
    for (String lang : languages) {
        System.out.println("\t" + lang);
    }
}
```

3) Handling radio button field

HTML code:

Gender:
<input type="radio" name="gender" value="male" />Male

```
<input type="radio" name="gender" value="female" />Female
```

Java code in servlet:

```
String gender = request.getParameter("gender");
System.out.println("Gender is: " + gender);
```

4) Handling text area field

HTML code:

```
Feedback:<br/>
<textarea rows="5" cols="30" name="feedback"></textarea>
```

Java code in servlet:

```
String feedback = request.getParameter("feedback");
System.out.println("Feed back is: " + feedback);
```

5) Handling dropdown list (combobox) field

HTML code:

Job Category:

```
<select name="jobCat">
  <option value="tech">Technology</option>
  <option value="admin">Administration</option>
  <option value="biology">Biology</option>
  <option value="science">Science</option>
</select>
```

Java code in servlet:

```
String jobCategory = request.getParameter("jobCat");
System.out.println("Job category is: " + jobCategory);
```

Introduction to Request Dispatcher

RequestDispatcher is an interface, implementation of which defines an object which can dispatch request to any resources (such as HTML, Image, JSP, Servlet) on the server.

Methods of RequestDispatcher

RequestDispatcher interface provides two important methods

| Methods | Description |
|--|---|
| void forward(ServletRequest request, ServletResponse response) | forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server |
| void include(ServletRequest request, ServletResponse response) | includes the content of a resource (servlet, JSP page, HTML file) in the response |

How to get an Object of RequestDispatcher

getRequestDispatcher() method of ServletRequest returns the object of RequestDispatcher.

```
RequestDispatcher rs = request.getRequestDispatcher("hello.html");
rs.forward(request,response);
```

ServletRequest object **resource name**

```
RequestDispatcher rs = request.getRequestDispatcher("hello.html");

rs.forward(request, response);
```

forward the request and response to "hello.html" page

OR

```
RequestDispatcher rs = request.getRequestDispatcher("hello.html");
rs.include(request, response);
```

ServletRequest object **Resource name**

```
RequestDispatcher rs = request.getRequestDispatcher("first.html");

rs.include(request, response);
```

include the response of "first.html" page in current servlet response

Example demonstrating usage of RequestDispatcher

In this example, we will show you how RequestDispatcher is used to forward or include response of a resource in a Servlet. Here we are using index.html to get username and password from the user, Validate Servlet will validate the password entered by the user, if the user has entered "studytonight" as password, then he will be forwarded to Welcome Servlet else the user will stay on the index.html page and an error message will be displayed.

Files to be created :

- index.html will have form fields to get user information.
- Validate.java will validate the data entered by the user.
- Welcome.java will be the welcome page.
- web.xml , the deployment descriptor.

index.html

```
<form method="post" action="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;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        String name = request.getParameter("user");
        String password = request.getParameter("pass");

        if(password.equals("studytonight"))
        {
            RequestDispatcher rd = request.getRequestDispatcher("Welcome");
            rd.forward(request, response);
        }
        else
        {
            out.println("<font color='red'><b>You have entered incorrect password</b></font>");
            RequestDispatcher rd = request.getRequestDispatcher("index.html");
            rd.include(request, response);
        }
    } 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;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {

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

```

web.xml

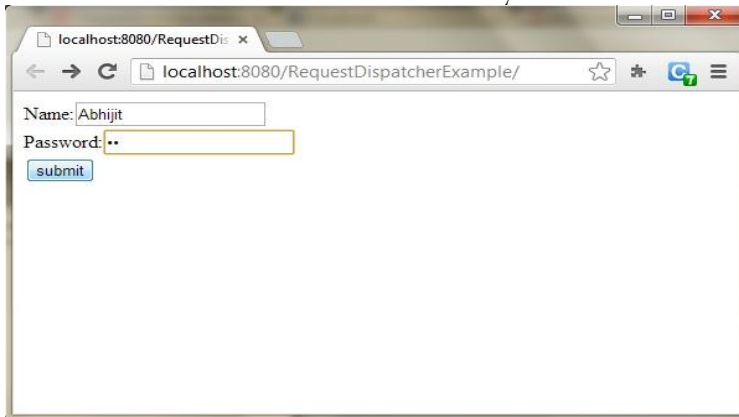
```

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

```

```
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Welcome</servlet-name>
  <url-pattern>/Welcome</url-pattern>
</servlet-mapping>
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

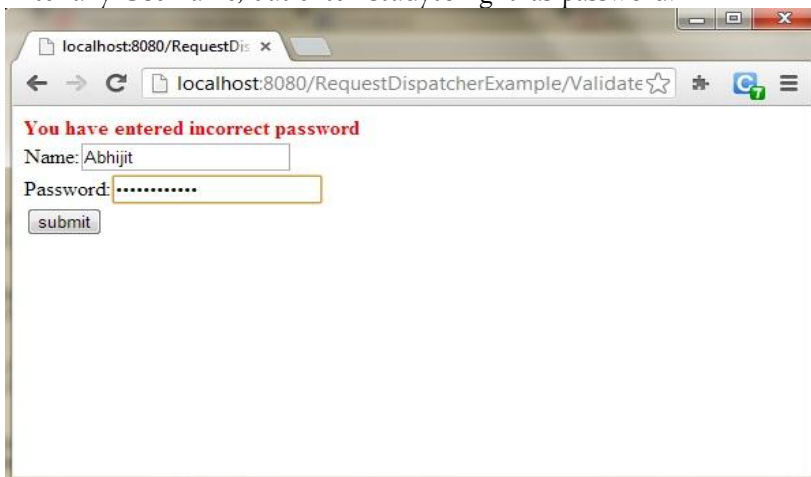
This will be the first screen. You can enter your Username and Password here.



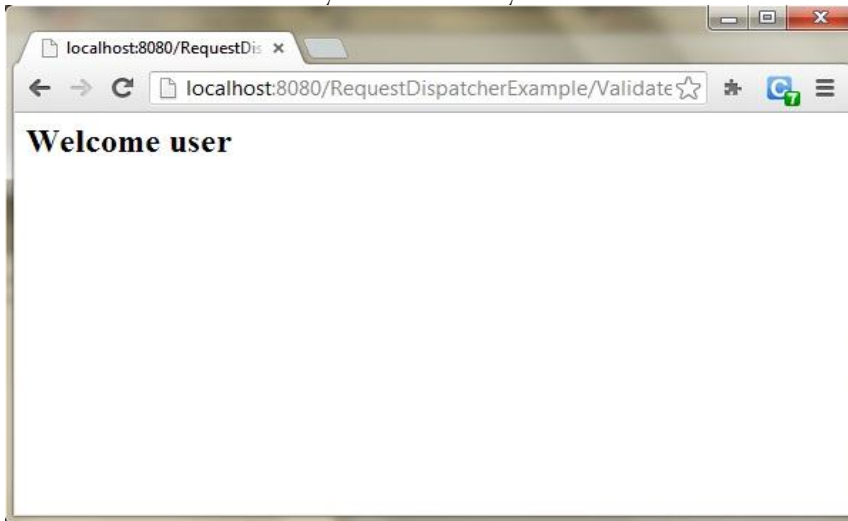
When you click on Submit, Password will be validated, if it is not 'studytonight', error message will be displayed.



Enter any Username, but enter 'studytonight' as password.



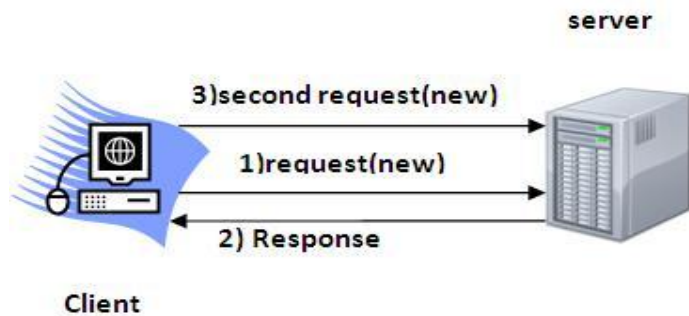
Password will be successfully validated and you will be directed to the Welcome Servlet.



State management:

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 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. It is shown in the figure given below:



Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

1. Cookies
2. Hidden Form Field
3. URL Rewriting
4. HttpSession

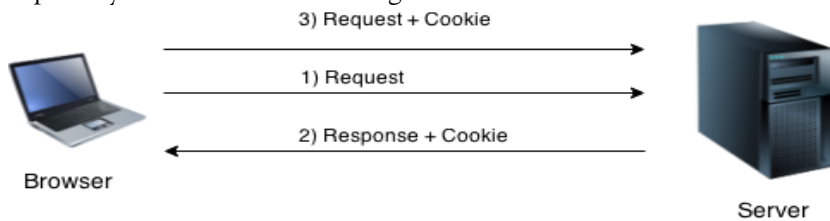
Cookies in Servlet

A **cookie** is a small piece of information 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 Cookie

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.

How to create Cookie?

Let's see the simple code to create cookie.

1. `Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object`
2. `response.addCookie(ck);//adding cookie in the response`

How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

1. `Cookie ck=new Cookie("user","");//deleting value of cookie`
2. `ck.setMaxAge(0);//changing the maximum age to 0 seconds`
3. `response.addCookie(ck);//adding cookie in the response`

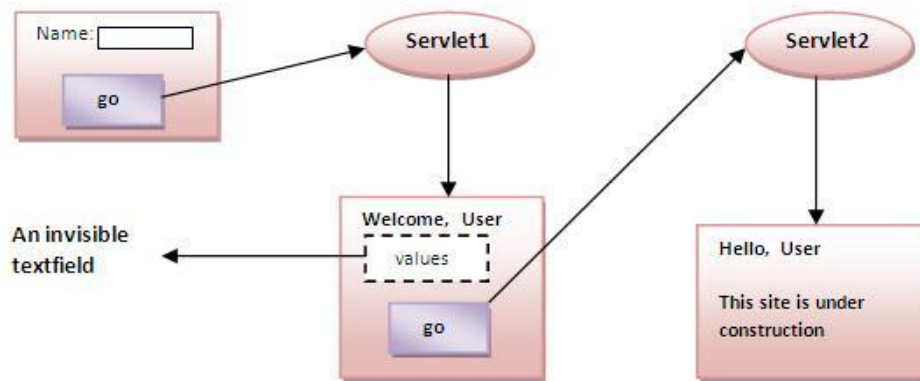
How to get Cookies?

Let's see the simple code to get all the cookies.

1. `Cookie ck[]=request.getCookies();`
2. `for(int i=0;i<ck.length;i++){`
3. `out.print("
" + ck[i].getName() + " " + ck[i].getValue());//printing name and value of cookie`
4. `}`

Hidden Form Fields:

1. A web server can send a hidden HTML form field along with a unique session ID as follows:
2. `<input type="hidden" name="sessionid" value="12345">`
3. 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.
4. 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.



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.

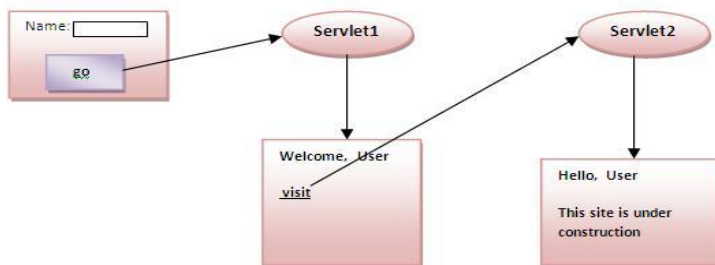
URL Rewriting:

1. 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.
2. 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.
3. 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.

We can send parameter name/value pairs using the following format:

`url?name1=value1&name2=value2&??`

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use `getParameter()` method to obtain a parameter value.



Advantage of URL Rewriting

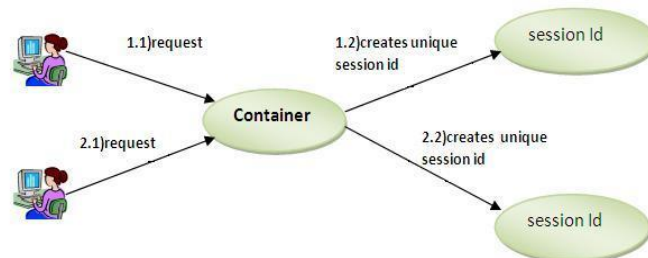
1. It will always work whether cookie is disabled or not (browser independent).
2. Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

1. It will work only with links.
2. It can send Only textual information.

HttpSession Object:

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



How to get the HttpSession object ?

The HttpServletRequest interface provides two methods 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.

Deleting Session Data:

When you are done with a user's session data, you have several options:

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

```

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

Chapter-2

JSP is the technology and whose specification is implemented by server vendors. JSP is an alternative technology for servlets.

What Is a JSP Page?

A JSP page is a text document that contains two types of text: static data, which can be expressed in any text-based format (such as HTML, SVG, WML, and XML), and JSP elements, which construct dynamic content.

The recommended file extension for the source file of a JSP page is .jsp. The page can be composed of a top file that includes other files that contain either a complete JSP page or a fragment of a JSP page. The recommended extension for the source file of a fragment of a JSP page is .jspf.

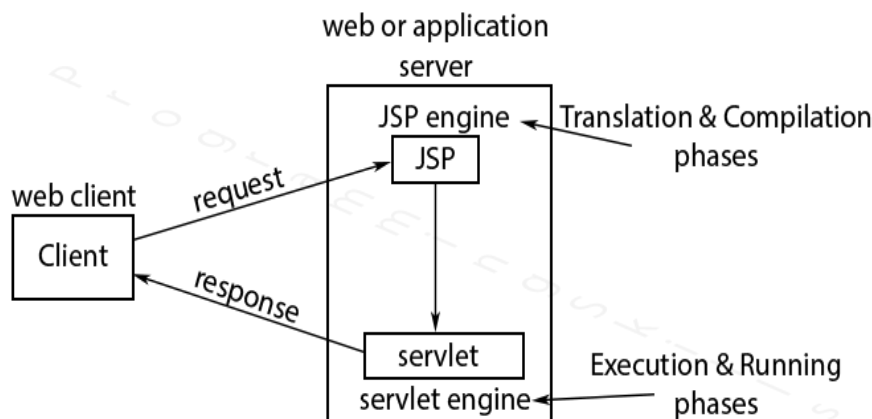
The JSP elements in a JSP page can be expressed in two syntaxes, standard and XML, though any given file can use only one syntax. A JSP page in XML syntax is an XML document and can be manipulated by tools and APIs for XML documents.

Architecturally, JSP may be viewed as a high-level abstraction of Java servlets. JSPs are translated into servlets at runtime; each JSP servlet is cached and re-used until the original JSP is modified.

JSP can be used independently or as the view component of a server-side model-view-controller design, normally with JavaBeans as the model and Java servlets (or a framework such as Apache Struts) as the controller. This is a type of Model 2 architecture.

JSP allows Java code and certain pre-defined actions to be interleaved with static web markup content, such as HTML, with the resulting page being compiled and executed on the server to deliver a document. The compiled pages, as well as any dependent Java libraries, contain Java bytecode rather than machine code. Like any other Java program, they must be executed within a Java virtual machine (JVM) that interacts with the server's host operating system to provide an abstract, platform-neutral environment.

JSPs are usually used to deliver HTML and XML documents, but through the use of OutputStream, they can deliver other types of data as well.



Life cycle methods of JSP:

Since, JSP is the server side dynamic technology and it extends the functionality of web or application server, it contains the following life cycle methods:

`public void jspInit ();` `public void jspService (ServletRequest, ServletResponse);` `public void jspDestroy ();`

The above three life cycle methods are exactly similar to life cycle methods of servlet.

TAGS in JSP

Writing a program in JSP is nothing but making use of various tags which are available in JSP. In JSP we have three categories of tags; they are scripting elements, directives and standard actions.

SCRIPTING ELEMENTS:

Scripting elements are basically used to develop preliminary programming in JSP such as, declaration of variables, expressions and writing the java code. Scripting elements are divided into three types; they are declaration tag, expression tag and scriptlet.

1. Declaration tag: Whenever we use any variables as a part of JSP we have to use those variables in the form of declaration tag i.e., declaration tag is used for declaring the variables in JSP page. Syntax:
2. `<%! Variable declaration or method definition %>`

When we declare any variable as a part of declaration tag those variables will be available as data members in the servlet and they can be accessed through out the entire servlet. When we use any methods definition as a part of declaration tag they will be available as member methods in servlet and it will be called automatically by the servlet container as a part of service method.

For example-1:

```
<%!      int a = 10, b = 30, c;%>
```

For example-2:

```
<%!
    int count() {
        return (a + b);
    }
%>
```

3. Expression tag: Expression tags are used for writing the java valid expressions as a part of JSP page.

Syntax:

```
<%= java valid expression %>
```

Whatever the expression we write as a part of expression tags that will be given as a response to client by the servlet container. All the expression we write in expression tag they will be placed automatically in out.println () method and this method is available as a part of service method.

Note: Expressions in the expression tag should not be terminated by semi-colon (;) .

For example-1:

```
<%! int a = 10, b = 20 %>
```

```
<%= a + b%>
```

The equivalent servlet code for the above expression tag is out.println (a+b); out is implicit object of JSPWriter class.

For example-2:

```
<%= new java.util.Date()%>
```

```
out.println (new java.util.Date ());
```

4. Scriptlet tag: Scriptlets are basically used to write a pure java code. Whatever the java code we write as a part of scriptlet, that code will be available as a part of service () method of servlet.

Syntax:

```
<% pure java code%>
```

Write a scriptlet for generating current system date?

Answer:

web.xml:

```
<web-apps>
```

```
</web-apps>
```

Date Time.java:

```
<html>
```

```
<title>Current Date & Time</title>
```

```
<head><h4>Current date & time</h4></head>
```

```
<body>
```

```
<%
```

```
%>
```

```
</body>
```

```
</html>
```

[or]

```

<html>
    Date d=new Date (); String s=d.toString (); out.println (s);
    <title>Current Date & Time</title>
    <head><h4>Current date & time</h4></head>
    <body>
        <%=new Date()%>
    </body>
</html>

```

Write a JSP page to print I to IO numbers?

Answer:

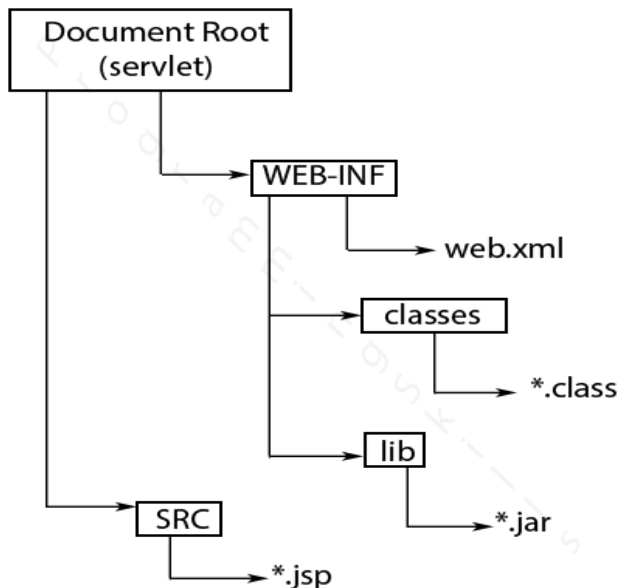
One2TenNumbers.jsp:

```

<html>
    <title>Print Numbers I-IO</title>
    <head>Numbers I-IO</head><br>
    <body>
        <%
            for (int i = I; i <= IO; i++) {
                out.println (i);
            }
        %>
    </body>
</html>

```

In order to execute any JSP program one must follow the following directory structure:



Write JSP program to print "Hello JSP world "?

Answer:

HelloJSP.jsp:

```

<html>
    <title>Fisrt Trail</title>
    <head>Fresher to JSP</head><br>
    <body>

```

```
<h3>Hello JSP world </h3>
</body>
</html>
```

Note: Whenever we deploy a JSP application in webapps folder of Tomcat we get an appropriate equivalent servlet for the corresponding JSP file. For example, when we deploy first.jsp through a document root first in webapps folder of tomcat we get first_jsp.java (which is nothing but a servlet) and first_jsp.class by Tomcat server. The location of servlet and .class file is as follows:

Write a JSP page which will display current date and time?

Answer:

Date Time.jsp:

```
<html>
<title>Current Date & Time</title>
<head>Date & Time without using out.println</head><br>
<body>
<%
    java.util.Date d = new java.util.Date();
%>
<h4>Current date & time</h4>
<h3><%= d%></h3>
</body>
</html>
```

Write a JSP page which will display number of times a request is made [write a JSP for hit counter]?

Answer:

ReloadPageCount.jsp:

```
<html>
<title>Number of Reloads</title>
<head>Number of visitings to a browser</head><br>
<body>
<%! int ctr = 0;%>
<%!
    int count() {
        return (++ctr);
    }
%>
<h3><%= count()%></h3>
</body>
</html>
```

Note: Within servlet we use to write html code to generate presentation logic whereas in JSP environment within html program we are making use of JSP tags.

Write a JSP page which will retrieve the data from database?

Answer:

```
<%@ page import="java.sql.*, java.io.*" %>
<html>
<title>Data From Database</title>
<head>Retrieve data from Database</head>
```

```
<body>
<%!
    Connection con = null;
    Statement st = null;
    public void jspInit()
    {
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:Hanuman", "scott",
"tiger");
            st = con.createStatement();
        }
        catch (Exception e)
        {
            out.println(e);
        }
    }
%>
<%!
    try
    {
        ResultSet rs = st.executeQuery("select * from employee");
        while (rs.next())
        {

            out.println("<h3>" + rs.getString(1) + " " + rs.getString(2) + "</h3>")
        }
    }

    catch (Exception e)
    {
        out.println(e);
    }
%>
</body>
</html>
```

Directives:

Directives are basically used to configure the code that is generated by container in a servlet. As a part of JSP we have three types of directives; they are page directives, include directives and taglib directives.

Page directives:

Page directives are basically used for supplying compile time information to the container for generating a servlet. The page directive will take the data in the form of (key, value) pair.

```
<%@
```

```
page attribute_name1=attribute_value1,
```

```
attribute_name2=attribute_value2,
```

```
.....,
```


```
Attribute_nameN=attribute_valueN
```

```
%>
```

- Whenever we use page directive as a part of JSP program that statement must be the first statement.
- The scope of page directive is applicable to current JSP page only.

The following table gives the page directive attribute name and page directive attribute value:

| Attribute name | Attribute value |
|--|--|
| import | This attribute is used for importing either pre-defined or user-defined packages. The default value is java.lang.* For example: <%@ page import="java.sql.*, java.io.*" %> |
| contentType | . This attribute is used for setting the MIME type (plain/text, img/jpeg, img/gif, audio/wave, etc.). The default value is text/html. For example: <%@ page contentType="img/jpeg" %> |
| language | This attribute represents by default java i.e., in order to represent any business logic a JSP program is making use of java language. The attribute language can support any of the other programming languages for developing a business logic at server side. For example: <%@ page language="java" %> |
| isThreadSafe | This attribute represents by default true which represents one server side resource can be accessed by many number of clients (each client is treated as one thread). At a time if we make the server side resource to be accessed by a single client then the value of isThreadSafe is false. For example: <%@ page isThreadSafe="false" %> |
| <ul style="list-style-type: none"> • isErrorPage • errorPage | When we write 'n' number of JSP pages, there is a possibility of occurring exceptions in each and every JSP page. It is not recommended for the JSP programmer to write try and catch blocks in each and every JSP page. It is always recommended to handle all the exceptions in a single JSP page. isErrorPage is an attribute whose default value is true which indicates exceptions to be processed in the same JSP page which is not recommended. If isErrorPage is false then exceptions are not processed as a part of current JSP page and the exceptions are processed in some other JSP page which will be specified through an attribute called errorPage. For example: <%@ page isErrorPage="false" errorPage="err.jsp" %> err.jsp: <%= exception%> [or] <%= exception.getMessage()%> |
| <ul style="list-style-type: none"> • autoFlush • buffer | Whenever the server side program wants to send large amount of data to a client, it is recommended to make autoFlush value as false and we must specify the size of the buffer in terms of kb. The default value of autoFlush is true which represents the server side program gives the response back to the client each and every time. Since, the buffer size is zero. For example: <%@ page autoFlush="false" buffer="12kb" %> |

| | |
|---------|--|
| | <code><%@ page autoflush="true" %></code> [by default] |
| session | <p>When we want to make 'n' number of independent requests as consecutive requests one must use the concept of session. In order to maintain the session we must give the value of session attribute has true in each and every JSP page (recommended). The default value of session is true which represents the session is applicable to current JSP page.</p> <p>For example:</p> <p><code><%@ page session="true" %></code></p> <p>- session will be created or old session will be continued.</p> |
| info | <p>Using this attribute it is recommended for the JSP programmer to specify functionality about a JSP page, on what date it is created and author.</p> <p><code>javax.servlet.Servlet</code></p>  <p><code>public String getServletInfo ();</code></p> |

Write a JSP page which illustrates the concept of `isErrorPage` and `errorPage`?

Answer:

web.xml:

```
<web-app>
</web-app>
```

Exception.jsp:

```
<%@ page isErrorPage="false" errorPage="ErrorPage.jsp" %>
```

```
<html>
```

```
<body>
```

```
<%= 30 / 0 %>
```

```
</body>
```

```
</html>
```

ErrorPage.jsp:

```
<%@ page isErrorPage="true" %>
```

```
<html>
```

```
<body>
```

```
Exception is <%= exception %> generated...<br> Exception message is <%= exception.getMessage() %><br>
```

```
</body>
```

```
</html>
```

Include directives:

Include is the directive to include the server side resource. The server side resource can be either an html file or JSP or a servlet. If we include html file, it will be executed by browser when the response is rendering to the client. When we include a JSP or a servlet, it will be executed by container.

Syntax:

```
<% include file = "file name to be included" %>
```

For example:

```
<% include file = "copyright.html" %>
```

Standard Actions:

These are basically used to pass runtime information to the container. As a part of JSP we have the following standard actions; they are `<JSP:forward/>`, `<JSP:include/>`, `<JSP:param/>`, `<JSP:useBean/>`, `<JSP:setProperty/>` and `<JSP:getProperty/>`.

1. `<JSP:forward/>`:

When we want to forward a request and response to the destination JSP page from the source JSP we must use `<JSP:forward>`.

Syntax:

Without body:

```
<JSP:forward page="relative or absolute path of JSP page"/>
```

With body:

```
<JSP:forward page="relative or absolute path of JSP page">
  <JSP:param name="param name1" value="param value1"/>
  <JSP:param name="param name2" value="param value2"/>
</JSP:forward>
```

For example:

```
<JSP:forward page="y.jsp">
  <JSP:param name="v1" value="10"/>
  <JSP:param name="v2" value="20"/>
</JSP:forward>
```

When we use this tag we get the response of destination JSP page only but not source JSP page.

2. *<JSP:include/>*:

This tag is used for processing a client request by a source JSP page by including other JSP pages and static resources like html's. One source JSP can include 'n' number of server side resources and finally we get the response of source JSP only.

Syntax:

Without body:

```
<JSP:include page="relative or absolute path of JSP page"/>
```

With body:

```
<JSP:include page="relative or absolute path of JSP page">
  <JSP:param name="param name1" value="param value1"/>
  <JSP:param name="param name2" value="param value2"/>
</JSP:include>
```

For example-1:

```
<JSP:include page="y.jsp">
  <JSP:param name="v1" value="10"/>
  <JSP:param name="v2" value="20"/>
</JSP:include>
```

For example-2:

```
<JSP:include page="z.jsp">
  <JSP:param name="v3" value="30"/>
  <JSP:param name="v4" value="40"/>
</JSP:include>
```

3. *<JSP:param/>*:

This tag is used for passing the local data of one JSP page to another JSP page in the form of (key, value) pair.

```
<p class="text-str"><strong>Syntax:</strong></p>
```

Here, name represents name of the parameter or attribute and it must be unique value represents value of the parameter and should be always string. <JSP:param/> tag should be used in connection with either <JSP:forward/> or <JSP:include/>.

For example-1:

```
<JSP:forward page="y.jsp">
  <JSP:param name="v1" value="10"/>
  <JSP:param name="v2" value="20"/>
</JSP:forward>
```

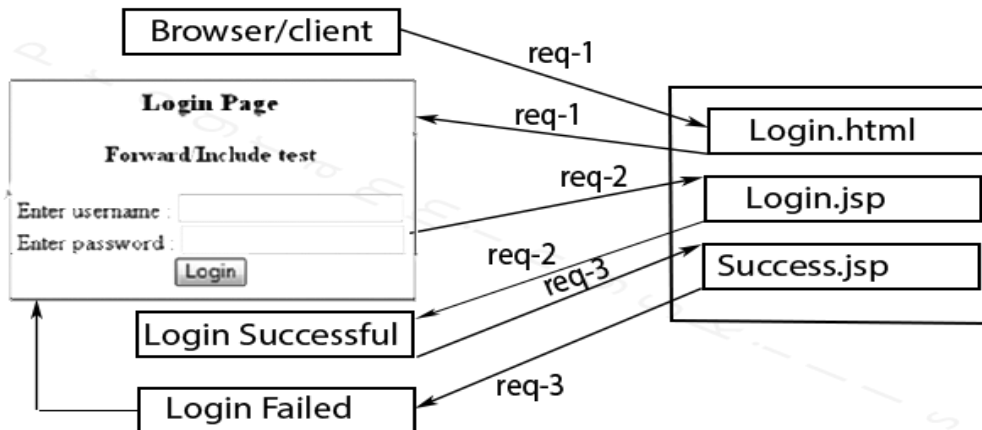
For example-2:

```
<JSP:include page="y.jsp">
  <JSP:param name="v1" value="10"/>
  <JSP:param name="v2" value="20"/>
</JSP:include>
```

For example-3:

```
<JSP:include page="z.jsp">
  <JSP:param name="v3" value="30"/>
  <JSP:param name="v4" value="40"/>
</JSP:include>
```

Write a JSP page which illustrates the concept of <JSP:forward/> and <JSP:include/>?



Answer:

web.xml:

```
<web-app>
</web-app>
```

Login.html:

```
<html>
  <head><center><h3>Login Page</h3></center></head>
  <body>
    <center>
      <h4>Forward/Include test</h4>
      <form name="login" action="Login.jsp" method="post">
        <p>Enter username : <input type="text" name="login_uname" value=""><br>
          Enter password : <input type="password" name="login_pwd" value=""><br>
          <input type="submit" value="Login">
      </form>
```

```

</center>
</body>
</html>
Login.jsp:
<%      String s1 = request.getParameter("login_uname");
        String s2 = request.getParameter("login_pwd");
        if (s1.equals("kalpana") && s2.equals("test")) { %>
            <JSP:forward page="Success.jsp"/>
        <%      } else { %>
            <h5>Login failed</h5>
            <JSP:include page="Login.html"/>
        <%      } %>

```

Success.jsp:

```

<h5>Login Successful</h5> Welcome to :&nbsp;
<h4><%= request.getParameter("login_uname") %></h4>

```

Implicit Objects

Implicit objects are those which will be available to each and every JSP page by default. In JSP we have the following implicit objects.

| Implicit object | Creator or Instantiated by |
|-----------------|------------------------------------|
| out | JSPWriter extends PrintWriter |
| request | HttpServletRequest |
| response | HttpServletResponse |
| application | ServletContext |
| config | ServletConfig |
| pageContext | PageContext |
| page | Object (core java example is this) |
| session | HttpSession |
| exception | Throwable |

The request Object:

The request object is an instance of a `javax.servlet.http.HttpServletRequest` object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.

We would see complete set of methods associated with request object in coming chapter: [JSP - Client Request](#).

The response Object:

The response object is an instance of a `javax.servlet.http.HttpServletResponse` object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.

We would see complete set of methods associated with response object in coming chapter: [JSP - Server Response](#).

The out Object:

The out implicit object is an instance of a `javax.servlet.jsp.JspWriter` object and is used to send content in a response.

The initial `JspWriter` object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the `buffered='false'` attribute of the page directive.

The `JspWriter` object contains most of the same methods as the `java.io.PrintWriter` class. However, `JspWriter` has some additional methods designed to deal with buffering. Unlike the `PrintWriter` object, `JspWriter` throws `IOExceptions`.

Following are the important methods which we would use to write boolean, char, int, double, object, String etc.

| Method | Description |
|--------|-------------|
|--------|-------------|

| | |
|---------------------------------------|--|
| <code>out.print(dataType dt)</code> | Print a data type value |
| <code>out.println(dataType dt)</code> | Print a data type value then terminate the line with new line character. |
| <code>out.flush()</code> | Flush the stream. |

The session Object:

The session object is an instance of `javax.servlet.http.HttpSession` and behaves exactly the same way that session objects behave under Java Servlets.

The session object is used to track client session between client requests. We would see complete usage of session object in coming chapter: [JSP - Session Tracking](#).

The application Object:

The application object is direct wrapper around the `ServletContext` object for the generated Servlet and in reality an instance of a `javax.servlet.ServletContext` object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the `jspDestroy()` method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

You can check a simple use of Application Object in chapter: [JSP - Hits Counter](#)

The config Object:

The config object is an instantiation of `javax.servlet.ServletConfig` and is a direct wrapper around the `ServletConfig` object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following config method is the only one you might ever use, and its usage is trivial: `config.getServletName()`;

This returns the servlet name, which is the string contained in the `<servlet-name>` element defined in the `WEB-INF\web.xml` file

The pageContext Object:

The `pageContext` object is an instance of a `javax.servlet.jsp.PageContext` object. The `pageContext` object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The application, config, session, and out objects are derived by accessing attributes of this object.

The `pageContext` object also contains information about the directives issued to the JSP page, including the buffering information, the `errorPageURL`, and page scope.

The page Object:

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

The page object is really a direct synonym for the this object.

The exception Object:

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

When you are writing JSP code, a programmer may leave coding errors which can occur at any part of the code. You can have following type of errors in your JSP code:

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

There are three ways to handle exceptions in a JSP page:

- Catching the exceptions directory in the JSP page.
- Specifying a separate JSP error page using page directive. This is called page-level exception handling.
- Configuring mapping of exception types-error pages in web.xml file (application-level exception handling).

Using Exception Object:

The exception object is an instance of a subclass of Throwable (e.g., java.lang. NullPointerException) and is only available in error pages. Following is the list of important methods available in the Throwable class.

| SN | Methods with Description |
|----|--|
| 1 | public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | public Throwable getCause() Returns the cause of the exception as represented by a Throwable object. |
| 3 | public String toString() Returns the name of the class concatenated with the result of getMessage() |
| 4 | public void printStackTrace() Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 5 | public StackTraceElement[] getStackTrace() Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |
| 6 | public Throwable fillInStackTrace() Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace. |

Using Try...Catch Block:

If you want to handle errors with in the same page and want to take some action instead of firing an error page, you can make use of try....catch block.

Following is a simple example which shows how to use try...catch block. Let us put following code in main.jsp:

```
<html>
<head>
  <title>Try...Catch Example</title>
</head>
<body>
<%
  try{
    int i = 1;
    i = i / 0;
    out.println("The answer is " + i);
  }
  catch (Exception e){
    out.println("An exception occurred: " + e.getMessage());
  }
%>
</body>
</html>
```

Now try to access main.jsp, it should generate something as follows:

An exception occurred: / by zero

Page-level exception handling

In this way, we use JSP page directives to specify a separate error page which will be redirected in case exceptions occurred in the normal JSP page.

With this approach, there are two attributes of the page directive we have to specify for handling exceptions:

- **errorPage**="*path/to/error/handling/page*": Used in the JSP page in which exceptions might occur. This specifies an alternate JSP page which is written solely for handling exceptions such as showing exception class name, error message and exception stack trace. When the code in this page throws an exception, the server will redirect the client to the specified error handling page.
- **isErrorPage**="true": Used to indicate a JSP page is an error handling page so that the server will pass the exception object thrown by the original JSP page. The exception object is a subclass of java.lang.Throwable class so we can access the following details:

Let's see an example. Consider the following JSP page:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"    errorPage="error.jsp"
pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Calculate division of two numbers</title>
</head>
<body>
<%
    int x = Integer.parseInt(request.getParameter("x"));
    int y = Integer.parseInt(request.getParameter("y"));

    int div = x / y;

%>
<h2>Division of <%=x%> and <%=y%> is <%=div%></h2>
</body>
</html>
```

This page calculates division of two numbers passed from parameters query string. There are two possible exceptions here:

The passed parameters do not have numeric values (a NumberFormatException will be thrown).

The dividing number is zero (an ArithmeticException will be thrown).

And here is code of the error handling page (error.jsp):

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    isErrorPage="true"
    pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Error</title>
</head>
<body>
<h2>
    Error: <%=exception.getClass().getName() %><br/>
    <%=exception.getMessage() %><br/>
</h2>
</body>
</html>
```

Application-level exception handling

In this way, we can configure error handling page per exception type for all JSPs and servlets of the web application, by adding some entries in the web.xml file. Here is the syntax for configuring an error page for a specific exception type:

```
<error-page>
    <exception-type>fully_qualified_name_of_exception_class</exception-type>
```

```
<location>path_to_error_jsp_page</location>
</error-page>
```

For example:

```
<error-page>
  <exception-type>java.sql.SQLException</exception-type>
  <location>/dbError.jsp</location>
</error-page>
```

That will tell the server to redirect the clients to the dbError.jsp page whenever any JSP/Servlet throws an exception of type java.sql.SQLException. The dbError.jsp page does not have to declare the attribute isErrorPage.

Using this approach we can also specify a custom error handling page for specific HTTP error code like 404, 500... The syntax is:

```
<error-page>
  <error-code>HTTP_error_code</error-code>
  <location>path_to_error_jsp_page</location>
</error-page>
```

For example:

```
<error-page>
  <error-code>404</error-code>
  <location>/404error.jsp</location>
</error-page>
```

That will tell the server redirects all requests causing HTTP 404 error to the custom page 404error.jsp, instead of showing the default 404 error. Following is example code of the 404error.jsp page:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>404 Error</title>
</head>
<body>
<center>
  <h2>
    Apologize, we could not find the page you were looking for:
  </h2>
  ${pageContext.errorData.requestURI}
</center>
</body>
</html>
```

The errorData object provides detailed information about the error such as the request URI, status code, servlet name.

Model I and Model 2 (MVC) Architecture

Before developing the web applications, we need to have idea about design models. There are two types of programming models (design models)

1. Model I Architecture
2. Model 2 (MVC) Architecture

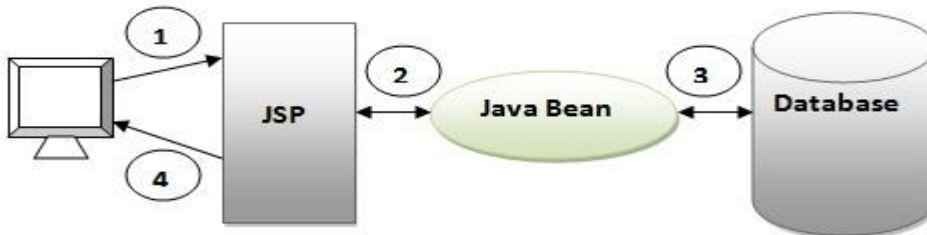
Model I Architecture

Servlet and JSP are the main technologies to develop the web applications.

Servlet was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

Problem in Servlet technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

JSP overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



As you can see in the above figure, there is picture which show the flow of the model I architecture.

1. Browser sends request for the JSP page
2. JSP accesses Java Bean and invokes business logic
3. Java Bean connects to the database and get/save data
4. Response is sent to the browser which is generated by JSP

Advantage of Model I Architecture

- Easy and Quick to develop web application

Disadvantage of Model I Architecture

- **Navigation control is decentralized** since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- **Time consuming** You need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
- **Hard to extend** It is better for small applications but not for large applications.

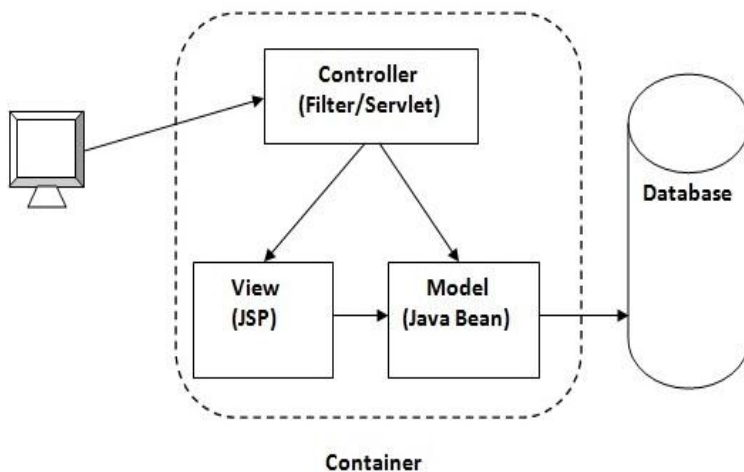
Model 2 (MVC) Architecture

Model 2 is based on the MVC (Model View Controller) design pattern. The MVC design pattern consists of three modules model, view and controller.

Model The model represents the state (data) and business logic of the application.

View The view module is responsible to display data i.e. it represents the presentation.

Controller The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.



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.
- 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.

Advantage of Model 2 (MVC) Architecture

- **Navigation control is centralized** Now only controller contains the logic to determine the next page.
- **Easy to maintain**
- **Easy to extend**
- **Easy to test**
- **Better separation of concerns**

Disadvantage of Model 2 (MVC) Architecture

- We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.