Servlet is a java program that runs inside JVM on the web server. It is used for developing dynamic web applications.  
Before we proceed further lets understand what is **dynamic web application?** A web application can be described as collection of web pages (e.g. a website) and when we call it dynamic, it simply means that the web pages are not same for all the users, web pages would be generated on server side based on the request made by client(user’s browser).

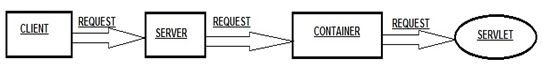
The main **difference between static and dynamic web page** is that static page as name suggests remains same for all users however a dynamic web page changes based on the request from client (user’s browser). For example, consider a web application that shows you two input fields & an add button and when you enter two numbers and click add, it shows you another web page that has the result of addition of two numbers, this web application is dynamic in nature as the second web page that shows you the result changes based on the user input, it is not static for all users.

**Web Container**

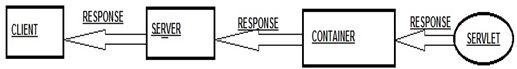
Web Container is an java application that controls servlet. Servlet do not have a main() method, So they require a container to load them. Container is a place where servlet gets deployed. When a client sends a request to web server that contain a servlet, server sends that request to container rather than to servlet directly. Container then finds out the requested servlet and pass the Http Request and response to servlet and loads the servlet methods i.e. doGet() or do Post(). Example of a web container is Tomcat.

Diagrams to show the request made by the client to the server and response received by the client.

**1. Request made by client to server**



**2. Response received by client**



## What is a Servlet?

A Java Servlet is a Java object that responds to HTTP requests. It runs inside a Servlet container. Here is an illustration of that:

|  |
| --- |
| Servlets inside a Java Servlet Container |
| **Servlets inside a Java Servlet Container** |

A Servlet is part of a Java web application. A Servlet container may run multiple web applications at the same time, each having multiple servlets running inside. Here is an llustration of that:

|  |
| --- |
| Web applications with multiple servlets inside a Java Servlet container |
| **Web applications with multiple servlets inside a Java Servlet container** |

A Java web application can contain other components than servlets. It can also contain Java Server Pages (JSP), Java Server Faces (JSF) and Web Services. This tutorial is about Java Servlets only, though.

## HTTP Request and Response

The browser sends an HTTP request to the Java web server. The web server checks if the request is for a servlet. If it is, the servlet container is passed the request. The servlet container will then find out which servlet the request is for, and activate that servlet. The servlet is activated by calling the Servlet.service() method.

Once the servlet has been activated via the service() method, the servlet processes the request, and generates a response. The response is then sent back to the browser.

## Servlet Containers

Java servlet containers are usually running inside a Java web server. A few common well known, free Java web servers are:

* [Jetty](http://jetty.codehaus.org/jetty/)
* [Tomcat](http://tomcat.apache.org/)

# Servlet Life Cycle

* [Load Servlet Class](http://tutorials.jenkov.com/java-servlets/servlet-life-cycle.html#load-servlet-class)
* [Create Instance of Servlet](http://tutorials.jenkov.com/java-servlets/servlet-life-cycle.html#create-instance-of-servlet)
* [Call the Servlets init() Method](http://tutorials.jenkov.com/java-servlets/servlet-life-cycle.html#call-the-servlets-init-method)
* [Call the Servlets service() Method](http://tutorials.jenkov.com/java-servlets/servlet-life-cycle.html#call-the-servlets-service-method)
* [Call the Servlets destroy() Method](http://tutorials.jenkov.com/java-servlets/servlet-life-cycle.html#call-the-servlets-destroy-method)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-05-25 |

A servlet follows a certain life cycle. The servlet life cycle is managed by the servlet container. The life cycle contains the following steps:

1. Load Servlet Class.
2. Create Instance of Servlet.
3. Call the servlets init() method.
4. Call the servlets service() method.
5. Call the servlets destroy() method.

Step 1, 2 and 3 are executed only once, when the servlet is initially loaded. By default the servlet is not loaded until the first request is received for it. You can force the container to load the servlet when the container starts up though. See [web.xml Servlet Configuration](http://tutorials.jenkov.com/java-servlets/web-xml.html#load-on-startup) for more details about that.

Step 4 is executed multiple times - once for every HTTP request to the servlet.  
Step 5 is executed when the servlet container unloads the servlet.  
Each step is described in more detail below:

|  |
| --- |
| The Java Servlet life cycle |
| **The Java Servlet life cycle** |

## Load Servlet Class

Before a servlet can be invoked the servlet container must first load its class definition. This is done just like any other class is loaded.

## Create Instance of Servlet

When the servlet class is loaded, the servlet container creates an instance of the servlet.

Typically, only a single isntance of the servlet is created, and concurrent requests to the servlet are executed on the same servlet instance. This is really up to the servlet container to decide, though. But typically, there is just one instance.

## Call the Servlets init() Method

When a servlet instance is created, its init() method is invoked. The init() method allows a servlet to initialize itself before the first request is processed.

You can specify init parameters to the servlet in the web.xml file. See [web.xml Servlet Configuration](http://tutorials.jenkov.com/java-servlets/web-xml.html#initParams) for more details.

## Call the Servlets service() Method

For every request received to the servlet, the servlets service() method is called. For HttpServlet subclasses, one of the doGet(), doPost() etc. methods are typically called.

As long as the servlet is active in the servlet container, the service() method can be called. Thus, this step in the life cycle can be executed multiple times.

## Call the Servlets destroy() Method

When a servlet is unloaded by the servlet container, its destroy() method is called. This step is only executed once, since a servlet is only unloaded once.

A servlet is unloaded by the container if the container shuts down, or if the container reloads the whole web application at runtime.

## Features of Servlet

Now that we have understood what is a servlet and for what purpose it is being used. Let’s proceed further and discuss its main features.

**1. Portable:**  
As I mentioned above that Servlet uses Java as a programming language, Since java is platform independent, the same holds true for servlets. For example, you can create a servlet on Windows operating system that users GlassFish as web server and later run it on any other operating system like Unix, Linux with Apache tomcat web server, this feature makes servlet portable and this is the main advantage servlet has over CGI.

**2. Efficient and scalable:**  
Once a servlet is deployed and loaded on a web server, it can instantly start fulfilling request of clients. The web server invokes servlet using a lightweight thread so multiple client requests can be fulling by servlet at the same time using the multithreading feature of Java. Compared to CGI where the server has to initiate a new process for every client request, the servlet is truly efficient and scalable.

**3. Robust:**  
By inheriting the top features of Java (such as Garbage collection, Exception handling, Java Security Manager etc.) the servlet is less prone to memory management issues and memory leaks. This makes development of web application in servlets secure and less error prone.

**Let’s see the hierarchy of packages:**

java.lang.Object

|\_extended byjavax.servlet.GenericServlet

|\_extended byjavax.servlet.http.HttpServlet

Every Servlet must implement the java.servlet.Servlet interface, you can do it by extending one of the following two classes: javax.servlet.GenericServlet or javax.servlet.http.HttpServlet. The first one is for protocol independent Servlet and the second one for http Servlet.

## HTTP Servlet

If you creating Http Servlet you must extend javax.servlet.http.HttpServlet class, which is an abstract class. Unlike Generic Servlet, the HTTP Servlet doesn’t override the service() method. Instead it overrides one or more of the following methods. It must override at least one method from the list below:

* **doGet()** – This method is called by servlet service method to handle the HTTP GET request from client. The Get method is used for getting information from the server
* **doPost()** – Used for posting information to the Server
* **doPut()** – This method is similar to doPost method but unlike doPost method where we send information to the server, this method sends file to the server, this is similar to the FTP operation from client to server
* **doDelete()** – allows a client to delete a document, webpage or information from the server
* **init() and destroy()** – Used for managing resources that are held for the life of the servlet
* **getServletInfo()** – Returns information about the servlet, such as author, version, and copyright.

**In Http Servlet there is no need to override the service() method** as this method dispatches the Http Requests to the correct method handler, for example if it receives HTTP GET Request it dispatches the request to the doGet() method.

**Interfaces in javax.servlet package**

* Servlet
* ServletRequest
* ServletResponse
* ServletConfig
* ServletContext
* SingleThreadModel
* RequestDispatcher
* ServletRequestListener
* ServletRequestAttributeListener
* ServletContextListener
* ServletContextAttributeListener
* Filter
* FilterConfig
* FilterChain

**Classes in javax.servlet package**

* GenericServlet
* ServletInputStream
* ServletOutputStream
* ServletException
* ServletRequestWrapper
* ServletRequestEvent
* ServletResponseWrapper
* ServletContextEvent
* ServletRequestAttributeEvent
* ServletContextAttributeEvent
* UnavailableException

**Interfaces in javax.servlet.http package**

* HttpSession
* HttpServletRequest
* HttpServletResponse
* HttpSessionAttributeListener
* HttpSessionListener
* HttpSessionBindingListener
* HttpSessionActivationListener
* HttpSessionContext

**Classes in javax.servlet.http package**

* HttpServlet
* Cookie
* HttpSessionEvent
* HttpSessionBindingEvent
* HttpServletRequestWrapper
* HttpServletResponseWrapper
* HttpUtils

**HTTP Servlet**

the HTTP Servlet doesn’t override the service() method. Instead it overrides the doGet() method or doPost() method or both. The doGet() method is used for getting the information from server while the doPost() method is used for sending information to the server.

In Http Servlet there is no need to override the service() method because this method dispatches the Http Requests to the correct method handler, for example if it receives HTTP GET Request it dispatches the request to the doGet() method.

## Hierarchy of Http Servlet

java.lang.Object

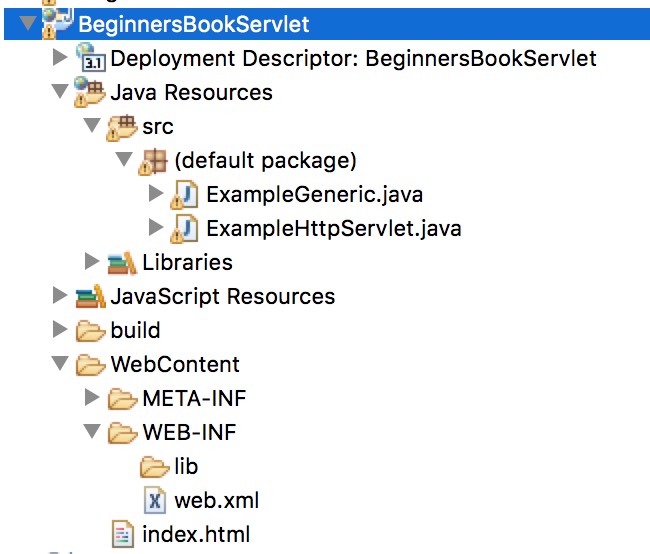
|\_extended byjavax.servlet.GenericServlet

|\_extended byjavax.servlet.http.HttpServlet

I have already discussed in the [Generic Servlet article](https://beginnersbook.com/2014/04/genericservlet-class/) that you should always use HttpServlet instead of the GenericServlet. HttpServlet is easier to work with, and has more methods to work with than GenericServlet.

## Http Servlet example

I am using Eclipse IDE for this example. Create New “Dynamic Web Project” from the Eclipse file menu.

Project structure (or you can hierarchy) would look like this, once you are done creating all the following files in IDE.  


**index.html**  
We are creating an html file that would call the servlet once we click on the link on web page. Create this file in WebContent folder. The path of the file should look like this: WebContent/index.html

index<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Http Servlet Demo</title>

</head>

<body>

<a href="welcome">Click to call Servlet</a>

</body>

</html>

**ExampleHttpServlet.java**  
Now, we are creating a Http Servlet by extending HttpServlet class. Right click on the src folder and create a new class file, name the file as ExampleHttpServlet. The file path should look like this: Java Resources/src/default package/ExampleHttpServlet.java

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Creating Http Servlet by Extending HttpServlet class

public class ExampleHttpServlet extends HttpServlet

{

private String mymsg;

public void init() throws ServletException

{

mymsg = "Http Servlet Demo";

}

public void doGet(HttpServletRequest request,

HttpServletResponse response) throws ServletException,

IOException

{

// Setting up the content type of web page

response.setContentType("text/html");

// Writing the message on the web page

PrintWriter out = response.getWriter();

out.println("<h1>" + mymsg + "</h1>");

out.println("<p>" + "Hello Friends!" + "</p>");

}

public void destroy()

{

// Leaving empty. Use this if you want to perform

//something at the end of Servlet life cycle.

}

}

**web.xml**  
This file can be found at this path WebContent/WEB-INF/web.xml. In this file we will map the Servlet with the specific URL. Since we are calling welcome page upon clicking the link on index.html page so we are mapping the welcome page to the Servlet class we created above.

<web-app>

<display-name>BeginnersBookServlet</display-name>

<welcome-file-list>

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

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

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

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>MyHttpServlet</servlet-name>

<servlet-class>ExampleHttpServlet</servlet-class>

</servlet>

<servlet-mapping>

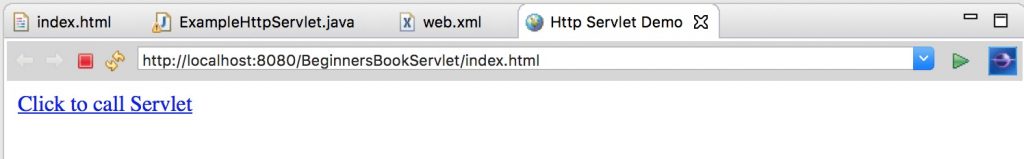
<servlet-name>MyHttpServlet</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

</web-app>

**Run the project:**  
Right click on the index.html, run on server.

**Output:**  


Upon clicking the link you would get this screen:  


## Get vs. Post

There are many differences between the Get and Post request. Let's see these differences:

|  |  |
| --- | --- |
| **GET** | **POST** |
| 1) In case of Get request, only **limited amount of data**can be sent because data is sent in header. | In case of post request, **large amount of data**can be sent because data is sent in body. |
| 2) Get request is **not secured**because data is exposed in URL bar. | Post request is **secured**because data is not exposed in URL bar. |
| 3) Get request **can be bookmarked.** | Post request **cannot be bookmarked.** |
| 4) Get request is **idempotent**. It means second request will be ignored until response of first request is delivered | Post request is **non-idempotent.** |
| 5) Get request is **more efficient**and used more than Post. | Post request is **less efficient**and used less than get. |

## GET and POST

Two common methods for the request-response between a server and client are:

* **GET**- It requests the data from a specified resource
* **POST**- It submits the processed data to a specified resource

## Methods of HttpServlet class

1. **protected void doGet(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the HTTP GET request from client. When overriding this method, read the request data, write the response headers, get the response’s writer or output stream object, and finally, write the response data.

2. **protected long getLastModified(HttpServletRequest req)**: Returns a long integer specifying the time the HttpServletRequest object was last modified, in milliseconds since midnight, January 1, 1970 GMT, or -1 if the time is not known

3. **protected void doHead(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the HTTP HEAD request from client. The client sends a HEAD request when it wants to see only the headers of a response, such as Content-Type or Content-Length

4. **protected void doPost(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the POST request from client. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information to the server. Unlike, doGet where we get information from the sever this method is used when we are transferring information from client to the server.

5. **protected void doPut(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the PUT request from client. This method is similar to doPost method but unlike doPost method where we send information to the server, this method sends file to the server, this is similar to the FTP operation from client to server.

6. **protected void doDelete(HttpServletRequest req, HttpServletResponse resp)**: Called by servlet service() method to handle the DELETE request from client that allows a client to delete a document, webpage or information from the server.

7. **protected void doOptions(HttpServletRequest req, HttpServletResponse resp)**: Called by the service method to allow a servlet to handle a OPTIONS request. The OPTIONS request determines which HTTP methods the server supports and returns an appropriate header.

8. **protected void doTrace(HttpServletRequest req, HttpServletResponse resp)**: This method is called by service() method for handling TRACE request. Used for debugging purposes.

9. **protected void service(HttpServletRequest req, HttpServletResponse resp)**: There is no need to override this method, this method receives the HTTP request from client and forwards them to the corresponding doXXX methods such as doGet(), doPost(), doHEAD() etc.

10. **public void service(ServletRequest req, ServletResponse res)**: Forwards client request to the protected service method. There’s no need to override this method as well.

# Servlet Life Cycle

Servlet life cycle can be described as a series of steps through which a servlet goes during its life span, starting from loading till it gets destroyed.

Before I start explaining the life cycle of Servlet, lets discuss few terminologies that you will encounter while reading this guide. It is important to learn what each term means, this will help you understand things faster.

**Web Server**: It is also known as HTTP Server, it can handle HTTP Requests send by client and responds the request with an HTTP Response.

**Web Container**: Also known as Servlet Container and Servlet Engine. It is a part of Web Server that interacts with Servlets. This is the main component of Web Server that manages the life cycle of Servlets.

Note: The servlet tutorials you find in this website uses apache tomcat web server. Although I mentioned it as web server, it is in fact a web server and web container both. (As mentioned above web container is a part of web server).

## Life Cycle of Servlet

Servlet life cycle contains five steps: 1) Loading of Servlet 2) Creating instance of Servlet 3) Invoke init() once 4) Invoke service() repeatedly for each client request 5) Invoke destroy()

For those who are wondering what is instance and invoke means: Instance and objects are same thing. Invoking a method means calling a method, it is just a fancy word that we use in programming world in place of calling :)

Let’s back to the main topic. Here are the five steps of servlet life cycle.

**Step 1: Loading of Servlet**  
When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the servlets.

**Step 2: Creating instance of Servlet**  
Once all the Servlet classes loaded, the servlet container creates instances of each servlet class. Servlet container creates only once instance per servlet class and all the requests to the servlet are executed on the same servlet instance.

**Step 3: Invoke init() method**  
Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet. This method initializes the servlet. There are certain init parameters that you can specify in the deployment descriptor (web.xml) file. For example, if a servlet has value >=0 then its init() method is immediately invoked during web container startup.

You can specify the element in web.xml file like this:

<servlet>

<servlet-name>MyServlet</servlet-name>

<servlet-class>com.beginnersbook.MyServletDemo</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

Now the init() method for corresponding servlet class **com.beginnersbook.MyServletDemo** would be invoked during web container startup.

**Note: The init() method is called only once during the life cycle of servlet.**

**Step 4: Invoke service() method**  
Each time the web server receives a request for servlet, it spawns a new thread that calls service() method. If the servlet is GenericServlet then the request is served by the service() method itself, if the servlet is HttpServlet then service() method receives the request and dispatches it to the correct handler method based on the type of request.

For example if its a Get Request the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like Post, Head, Put etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.

**Note**: Unlike init() and destroy() that are called only once, the service() method can be called any number of times during servlet life cycle. As long as servlet is not destroyed, for each client request the service() method is invoked.

**Out of all the 5 steps in life cycle, this is the only step that executes multiple times.**

**Step 5: Invoke destroy() method**  
When servlet container shuts down(this usually happens when we stop the web server), it unloads all the servlets and calls destroy() method for each initialized servlets.

**Web Server**: it can handle HTTP Requests send by clients and responds the request with an HTTP Response.

**Web Application(webapp)**: I would refer this as webapp in this guide. Basically the project is your web application, it is the collection of servlets.

**Web Container**: Also known as Servlet Container and Servlet Engine. It is a part of Web Server that interacts with Servlets. This is the main component of Web Server that manages the life cycle of Servlets.

## How Servlet Works?

1) When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the servlets. During this step Servlet container creates ServletContext object. **ServletContext is an interface that defines the set of methods that a servlet can use to communicate with the servlet container**.

**Note**: **There is only one ServletContext per webapp** which is common to all the servlets. ServletContext has several useful methods such as addListener(), addFilter() etc. For now I am not explaining them as I will cover them in a separate text about ServletContext.

2) Once the servlet is loaded, the servlet container creates the instance of servlet class. For each instantiated servlet, its init() method is invoked.

3) Client (user browser) sends an Http request to web server on a certain port. Each time the web server receives a request, the servlet container creates HttpServletRequest and HttpServletResponse objects. The HttpServletRequest object provides the access to the request information and the HttpServletResponse object allows us to format and change the http response before sending it to the client.

The servlet container spawns a new thread that calls service() method for each client request. **The service() method dispatches the request to the correct handler method based on the type of request**.  
For example if server receives a Get Request the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like Post, Head, Put etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.

4) When servlet container shuts down, it unloads all the servlets and calls destroy() method for each initialized servlets.

# welcome-file-list tag in web.xml file of Project

Have you ever seen <welcome-file-list> tag in your web.xml file and wondering what it is? In this text, I will explain what is this tag and why we use it.

The tag <welcome-file-list> is used for specifying the files that needs to be invoked by server by default, if you do not specify a file name while loading the project on browser.

For e.g. You have created a project named “MyServletProject” and you have few html pages and servlet classes defined in the project. However in browser you have given the url like this:

http://localhost:8080/MyServletProject

Usually we give the complete path like this:http://localhost:8080/MyServletProject/index.html. However if you have given the path like above then the webserver will look for the <welcome-file-list> tag in your project’s web.xml file. Lets say you have the following content in your web.xml file:

<web-app>

....

<welcome-file-list>

<welcome-file>myhome.htm</welcome-file>

<welcome-file>myindex.htm</welcome-file>

<welcome-file>mydefaultpage.htm</welcome-file>

</welcome-file-list>

....

</web-app>

Based on the welcome file list, server would look for the myhome.htm page if this doesn’t exist then the second welcome file myindex.html and so on till it finds a valid welcome file.

**Note**: If the <welcome-file-list> tag is not defined in web.xml or the welcome files defined in the <welcome-file> tags does not exist then the server would look for the following files in the given sequence:  
1) index.html  
2) index.htm  
3) index.jsp

# HttpRequest

* [Parameters](http://tutorials.jenkov.com/java-servlets/httprequest.html#parameters)
* [Headers](http://tutorials.jenkov.com/java-servlets/httprequest.html#headers)
* [InputStream](http://tutorials.jenkov.com/java-servlets/httprequest.html#inputstream)
* [Session](http://tutorials.jenkov.com/java-servlets/httprequest.html#session)
* [ServletContext](http://tutorials.jenkov.com/java-servlets/httprequest.html#servletcontext)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-05-25 |

The HttpServlet class request processing methods take two parameters.

1. javax.servlet.http.HttpRequest
2. javax.servlet.http.HttpResponse

For instance, here is the signature of the HttpServlet.doGet() method:

protected void doGet(

**HttpServletRequest request**,

HttpServletResponse response)

throws ServletException, IOException {

}

In this text I will look at the HttpRequest object.

The purpose of the HttpRequest object is to represent the HTTP request a browser sends to your web application. Thus, anything the browser may send, is accessible via the HttpRequest.

The HttpRequest object has a lot of methods, so I will just cover the most commonly used here. The rest you can read about in the JavaDoc, if you are interested.

## Parameters

The request parameters are parameters that are sent from the browser along with the request. Request parameters are typically sent as part of the URL (in the "query string"), or as part of the body of an HTTP request. For instance:

http://jenkov.com/somePage.html?param1=hello¶m2=world

Notice the "query string" part of the URL: ?param1=hello¶m2=world This part contains two parameters with parameter values:

param1=hello

param2=world

You can access these parameters from the HttpRequest object like this:

protected void doGet(

HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

**String param1 = request.getParameter("param1");**

**String param2 = request.getParameter("param2");**

}

You would use the same code, if the request parameters were sent in the body part of the HTTP request. If no parameter exists with the given name, null is returned.

In general, if the browser sends an HTTP GET request, the parameters are included in the query string in the URL. If the browser sends an HTTP POST request, the parameters are included in the body part of the HTTP request.

## Headers

The request headers are name, value pairs sent by the browser along with the HTTP request. The request headers contain information about e.g. what browser software is being used, what file types the browser is capable of receiving etc. In short, at lot of meta data around the HTTP request.

You can access the request headers from the HttpRequest object like this:

String contentLength = request.getHeader("Content-Length");

This example reads the Content-Length header sent by the browser.

The Content-Length header contains the number of bytes sent in the HTTP request body, in case the browser sends an HTTP POST request. If the browser sends an HTTP GET request, the Content-Length header is not used, and the above code will return null.

In general, If no header exists with the name passed to getHeader(), null is returned.

## InputStream

If the browser sends an HTTP POST request, request parameters and other potential data is sent to the server in the HTTP request body. It doesn't have to be request parameters that is sent in the HTTP request body. It could be pretty much any data, like a file or a SOAP request (web service request).

To give you access to the request body of an HTTP POST request, you can obtain an InputStream pointing to the HTTP request body. Here is how it is done:

InputStream requestBodyInput = request.getInputStream();

NOTE: You will have to call this method **before** calling any getParameter() method, because calling the getParameter() method on an HTTP POST request will cause the servlet engine to parse the HTTP request body for parameters. Once parsed, you cannot access the body as a raw stream of bytes anymore.

What you do with the data read from the InputStream is up to you. The servlet engine does not help you parse or interprete that data. You just get it raw.

## Session

It is possible to obtain the session object from the HttpRequest object too.

The session object can hold information about a given user, between requests. So, if you set an object into the session object during one request, it will be available for you to read during any subsequent requests within the same session time scope.

Here is how you access the session object from the HttpRequest object:

HttpSession session = request.getSession();

I will not get into more detail about the session object here. It is covered in more detail in its own text.

## ServletContext

You can access the ServletContext object from the HttpRequest object too. The ServletContext contains meta information about the web application. For instance, you can access context parameters set in the web.xml file, you can forward the request to other servlets, and you can store application wide parameters in the ServletContext too.

Here is how you access the ServletContext object from the HttpRequest object:

ServletContext context = request.getSession().getServletContext();

As you can see, you have to first get the session object, to get access to the ServletContext object.

I will not get into more detail about the ServletContext object here. It will be covered in more detail in its own text.

# HttpResponse

* [Writing HTML](http://tutorials.jenkov.com/java-servlets/httpresponse.html#writingHtml)
* [Headers](http://tutorials.jenkov.com/java-servlets/httpresponse.html#headers)
* [Content-Type](http://tutorials.jenkov.com/java-servlets/httpresponse.html#contentType)
* [Writing Text](http://tutorials.jenkov.com/java-servlets/httpresponse.html#writingText)
* [Content-Length](http://tutorials.jenkov.com/java-servlets/httpresponse.html#contentLength)
* [Writing Binary Data](http://tutorials.jenkov.com/java-servlets/httpresponse.html#writingBinary)
* [Redirecting to a Different URL](http://tutorials.jenkov.com/java-servlets/httpresponse.html#redirect)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-05-25 |

The HttpServlet class request processing methods take two parameters.

1. javax.servlet.http.HttpRequest
2. javax.servlet.http.HttpResponse

For instance, here is the signature of the HttpServlet.doGet() method:

protected void doGet(

HttpServletRequest request,

**HttpServletResponse response**)

throws ServletException, IOException {

}

In this text I will look at the HttpResponse object.

The purpose of the HttpResponse object is to represent the HTTP response your web application sends back to the browser, in response to the HTTP request the browser send to your web application.

The HttpResponse object has a lot of methods, so I will just cover the most commonly used here. The rest you can read about in the JavaDoc, if you are interested.

## Writing HTML

To send HTML back to the browser, you have to obtain the a PrintWriter from the HttpResponse object. Here is how:

PrintWriter writer = response.getWriter();

writer.write("<html><body>GET/POST response</body></html>");

## Headers

Just like the request object, the HttpRequest can contain HTTP headers. Headers must be set before any data is written to the response. You set a header on the response object like this:

response.setHeader("Header-Name", "Header Value");

As you can see, a response header is a name, value pair.

## Content-Type

The Content-Type header is a response header that tells the browser the type of the content you are sending back to it. For instance, the content type for HTML is text/html. Similarly, if what you send back to the browser is plain text, you use the content type text/plain.

Here is how you set the Content-Type header on the HttpResponse object:

response.setHeader("Content-Type", "text/html");

## Writing Text

You can write text back to the browser instead of HTML, like this:

response.setHeader("Content-Type", "text/plain");

PrintWriter writer = response.getWriter();

writer.write("This is just plain text");

First the Content-Type header is set to text/plain. Then a plain text string is written to the writer obtained from the response object.

## Content-Length

The Content-Length header tells the browser how many bytes your servlet is sending back. If you are sending binary data back you need to set the content length header. Here is how:

response.setHeader("Content-Length", "31642");

## Writing Binary Data

You can also write binary data back to the browser instead of text. For instance, you can send an image back, a PDF file or a Flash file or something like that.

Again, you will first have to set the Content-Type header to the type matching the data you are sending back. For instance, the content type for a PNG image is image/png.

You can search for "mime types" in your favourite search engine to find a list of mime types (content types), so you can find the mime type for the content you are sending back.

In order to write binary data back to the browser you cannot use the Writer obtained from response.getWriter(). Afterall, Writer's are intended for text.

Instead you have to use the OutputStream obtained from the response.getOutputStream() method. Here is how:

OutputStream outputStream = response.getOutputStream();

outputStream.write(...);

## Redirecting to a Different URL

You can redirect the browser to a different URL from your servlet. You cannot send any data back to the browser when redirecting. Here is how you redirect:

response.sendRedirect("http://jenkov.com");

# HttpSession

|  |  |
| --- | --- |
|  |  |

The HttpSession object represents a user session. A user session contains information about the user across multiple HTTP requests.

When a user enters your site for the first time, the user is given a unique ID to identify his session by. This ID is typically stored in a cookie or in a request parameter.

Here is how you access the session object:

protected void doPost(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

}

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

You can store values in the session object, and retrieve them later. First, let's see how you can store values in the session object:

session.setAttribute("userName", "theUserName");

This code sets an attribute named "userName", with the value "theUserName".

To read the value again, you do this:

String userName = (String) session.getAttribute("userName");

Values stored in the session object are stored in the memory of the servlet container.

## Sessions and Clusters

If you have an architecture with 2 web servers in a cluster, keep in mind that values stored in the session object of one server, may not be available in the session object on the other server. So, if a user's requests are divided evenly between the two servers, sometimes session values may be missing.

The solution to this problem would be one of:

1. Do not use session attributes.
2. Use a session database, into which session attributes are written, and from which it is read.
3. Use sticky session, where a user is always sent to the same server, throughout the whole session.

# RequestDispatcher

|  |  |
| --- | --- |
|  |  |

The RequestDispatcher class enables your servlet to "call" another servlet from inside another servlet. The other servlet is called as if an HTTP request was sent to it by a browser.

You can obtain a RequestDispatcher from the HttpServletRequest object, like this:

protected void doPost(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

RequestDispatcher requestDispatcher =

request.getRequestDispatcher("/anotherURL.simple");

}

The above code obtains a RequestDispatcher targeted at whatever Servlet (or JSP) that is mapped to the URL /anotherUrl.simple.

You can call the RequestDispatcher using either its include() or forward() method:

requestDispatcher.forward(request, response);

requestDispatcher.include(request, response);

By calling either the include() or forward() method the servlet container activates whatever Servlet is mapped to the URL the RequestDispatcher.

The activated servlet has access to the same request as the servlet calling it, and will write to the same response as your current servlet. That way you can merge the output of servlets into a single repsonse.

There is a little difference between calling the forward() and include() method.

The forward() method intended for use in **forwarding** the request, meaning after the response of the calling servlet has been committed. You cannot merge response output using this method.

The include() method merges the response written by the calling servlet, and the activated servlet. This way you can achieve "server side includes" using the include().

# Cookies and Servlets

* [Java Cookie Example](http://tutorials.jenkov.com/java-servlets/cookies.html#java-cookie-example)
* [Reading Cookies Sent From the Browser](http://tutorials.jenkov.com/java-servlets/cookies.html#reading-cookies)
* [Cookie Expiration](http://tutorials.jenkov.com/java-servlets/cookies.html#cookie-expiration)
* [Removing Cookies](http://tutorials.jenkov.com/java-servlets/cookies.html#removing-cookies)
* [Additional Cookie Settings](http://tutorials.jenkov.com/java-servlets/cookies.html#additional-cookie-settings)
* [Cookie Use Cases](http://tutorials.jenkov.com/java-servlets/cookies.html#cookie-use-cases)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-05-25 |

HTTP Cookies are little pieces of data that a web application can store on the client machine of users visiting the web application. Typically up to 4 kilo bytes of data. This text will explain how to set, read and remove cookies from inside Java servlets (or JSPs).

## Java Cookie Example

You can write cookies using the HttpServletResponse object like this:

Cookie cookie = new Cookie("myCookie", "myCookieValue");

response.addCookie(cookie);

As you can see, the cookie is identified by a name, "myCookie", and has a value, "myCookieValue". Thus, you can add many different cookies with different identifies (names). It's a bit like a Hashtable.

Whenever the the browser accesses the web application it submits the cookies stored on the client machine to the web application. Only cookies stored by the accessed web application are submitted. Cookies from other web applications are not submitted.

## Reading Cookies Sent From the Browser

You can read the cookies via the HttpServletRequest like this:

Cookie[] cookies = request.getCookies();

Note: the getCookies() method may return null!

Now you can iterate through the array of cookies and find the cookies you need. Unfortunately there is no way to obtain a cookie with a specific name. The only way to find that cookie again is to iterate the Cookie[] array and check each cookie name. Here is an example:

Cookie[] cookies = request.getCookies();

String userId = null;

for(Cookie cookie : cookies){

if("uid".equals(cookie.getName())){

userId = cookie.getValue();

}

}

This example finds the cookie with the name "uid" and stores its value in the

If you need to access more than one cookie, you could iterate the Cookie[] array once, and put the Cookie instances into a Map, using the cookie name as key, and the Cookie instance as value. Here is how that could look:

Map cookieMap = new HashMap();

Cookie[] cookies = request.getCookies();

for(Cookie cookie : cookies){

cookieMap.put(cookie.getName(), cookie);

}

After this code is executed, you can now access the cookies in the cookieMap using the cookie names as keys (cookieMap.get("cookieName")).

## Cookie Expiration

One important Cookie setting is the cookie expiration time. This time tells the browser receiving the cookie how long time it should keep the cookie before deleting it.

You set the cookie expiration time via the setMaxAge() method. This method takes the number of seconds the cookie is to live as parameter. Here is an example:

Cookie cookie = new Cookie("uid", "123");

**cookie.setMaxAge(24 \* 60 \* 60); // 24 hours.**

response.addCookie(cookie);

This example first creates a Cookie instance with the name "uid" and the value "123". Second, it sets the expiration to 24 hours using the setMaxAge() method. 24 hours is 60 seconds x 60 minutes x 24 hours (24 x 60 x 60). Finally the example sets the cookie on the HttpServletResponse object, so the cookie is included in the response sent to the browser.

## Removing Cookies

Sometimes you may want to remove a cookie from the browser. You do so by setting the cookie expiration time. You can set the expiration time to 0 or -1. If you set the expiration time to 0 the cookie will be removed immediately from the browser. If you set the expiration time to -1 the cookie will be deleted when the browser shuts down.

Here is an example:

Cookie cookie = new Cookie("uid", "");

**cookie.setMaxAge(0);**

response.addCookie(cookie);

If the browser already has a cookie stored with the name "uid", it will be deleted after receiving the cookie with the same name ("uid") with an expiration time of 0. If the browser did not already have the cookie stored, this new cookie is just thrown out immediately since its expiration time is 0.

## Additional Cookie Settings

A cookie has various other settings you can modify and access in addition to its expiration. Check out the Cookie class JavaDoc for more details.

## Cookie Use Cases

Cookies are most often used to store user specific information, like e.g. a unique user ID (for anonymous users which do not login), a session ID, or user specific setttings you do not want to store in your web applications database (if it has one).

# Servlet Filters

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-05-25 |

A Servlet filter is an object that can intercept HTTP requests targeted at your web application.

A servlet filter can intercept requests both for servlets, JSP's, HTML files or other static content, as illustrated in the diagram below:

|  |
| --- |
| A Servlet Filter in a Java Web Application |
| **A Servlet Filter in a Java Web Application** |

In order to create a servlet filter you must implement the javax.servlet.Filter interface. Here is an example servlet filter implementation:

import javax.servlet.\*;

import java.io.IOException;

/\*\*

\*/

public class SimpleServletFilter implements Filter {

public void init(FilterConfig filterConfig) throws ServletException {

}

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain filterChain)

throws IOException, ServletException {

}

public void destroy() {

}

}

When the servlet filter is loaded the first time, its init() method is called, just like with servlets.

When a HTTP request arrives at your web application which the filter intercepts, the filter can inspect the request URI, the request parameters and the request headers, and based on that decide if it wants to block or forward the request to the target servlet, JSP etc.

It is the doFilter() method that does the interception. Here is a sample implementation:

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain filterChain)

throws IOException, ServletException {

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

if(!"blockTheRequest".equals(myParam)){

filterChain.doFilter(request, response);

}

}

Notice how the doFilter() method checks a request parameter, myParam, to see if it equals the string "blockTheRequest". If not, the request is forwarded to the target of the request, by calling the filterChain.doFilter() method. If this method is not called, the request is not forwarded, but just blocked.

The servlet filter above just ignores the request if the request parameter myParam equals "blockTheRequest". You can also write a different response back to the browser. Just use the ServletResponse object to do so, just like you would inside a servlet.

You may have to cast the ServletResponse to a HttpResponse to obtain a PrintWriter from it. Otherwise you only have the OutputStream available via getOutputStream().

Here is an example:

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain filterChain)

throws IOException, ServletException {

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

if(!"blockTheRequest".equals(myParam)){

filterChain.doFilter(request, response);

return;

}

HttpResponse httpResponse = (HttpResponse) httpResponse;

httpResponse.getWriter().write("a different response... e.g in HTML");

}

## Configuring the Servlet Filter in web.xml

You need to configure the servlet filter in the web.xml file of your web application, before it works. Here is how you do that:

<filter>

<filter-name>myFilter</filter-name>

<filter-class>servlets.SimpleServletFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>myFilter</filter-name>

<url-pattern>\*.simple</url-pattern>

</filter-mapping>

With this configuration all requests with URL's ending in .simple will be intercepted by the servlet filter. All others will be left untouched.

# Introduction to JSP

**Introduction**

* It stands for **Java Server Pages**.
* It is a server side technology.
* It is used for creating web application.
* It is used to create dynamic web content.
* In this JSP tags are used to insert JAVA code into HTML pages.
* It is an advanced version of Servlet Technology.
* It is a Web based technology helps us to create dynamic and platform independent web pages.
* In this, Java code can be inserted in HTML/ XML pages or both.
* JSP is first converted into servlet by JSP container before processing the client’s request.

**JSP pages are more advantageous than Servlet:**

* They are easy to maintain.
* No recompilation or redeployment is required.
* JSP has access to entire API of JAVA .
* JSP are extended version of Servlet.

**JSP syntax**

Syntax available in JSP are following

* 1. **Declaration Tag**:-It is used to declare variables.

**Syntax:-**

<%! Dec var %>

**Example:-**

<%! int var=10; %>

* 1. **Java Scriplets**:- It allows us to add any number of JAVA code, variables and expressions.

**Syntax:-**

<% java code %>

* 1. **JSP Expression**:- It evaluates and convert the expression to a string.

**Syntax:-**

<%= expression %>

**Example:-**

<% num1 = num1+num2 %>

* 1. **JAVA Comments**:- It contains the text that is added for information which has to be ignored.

**Syntax:-**

<% -- JSP Comments %>

**Example of Hello World**  
We will make one .html file and .jsp file

**demo.jsp**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Hello World - JSP tutorial</title>

</head>

<body>

<%= "Hello World!" %>

</body>

</html>

**Advantages of using JSP**

* + It does not require advanced knowledge of JAVA
  + It is capable of handling exceptions
  + Easy to use and learn
  + It can tags which are easy to use and understand
  + Implicit objects are there which reduces the length of code
  + It is suitable for both JAVA and non JAVA programmer

**Disadvantages of using JSP**

* + Difficult to debug for errors.
  + First time access leads to wastage of time
  + It’s output is HTML which lacks features.

### The Lifecycle of a JSP Page

The JSP pages follow these phases:

* Translation of JSP Page
* Compilation of JSP Page
* Classloading (the classloader loads class file)
* Instantiation (Object of the Generated Servlet is created).
* Initialization ( the container invokes jspInit() method).
* Request processing ( the container invokes \_jspService() method).
* Destroy ( the container invokes jspDestroy() method).

## JSP Scripting elements

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

* scriptlet tag
* expression tag
* declaration tag

### JSP scriptlet tag

A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:

1. <%  java source code %>

### Example of JSP scriptlet tag

In this example, we are displaying a welcome message.

1. **<html>**
2. **<body>**
3. **<**% out.print("welcome to jsp"); %**>**
4. **</body>**
5. **</html>**

### Example of JSP scriptlet tag that prints the user name

In this example, we have created two files index.html and welcome.jsp. The index.html file gets the username from the user and the welcome.jsp file prints the username with the welcome message.

*File: index.html*

1. **<html>**
2. **<body>**
3. **<form** action="welcome.jsp"**>**
4. **<input** type="text" name="uname"**>**
5. **<input** type="submit" value="go"**><br/>**
6. **</form>**
7. **</body>**
8. **</html>**

*File: welcome.jsp*

1. <html>
2. <body>
3. <%
4. String name=request.getParameter("uname");
5. out.print("welcome "+name);
6. %>
7. </form>
8. </body>
9. </html>

# JSP expression tag

The code placed within **JSP expression tag** is written to the output stream of the response. So you need not write out.print() to write data. It is mainly used to print the values of variable or method.

### Syntax of JSP expression tag

1. **<**%=  statement %**>**

### Example of JSP expression tag

In this example of jsp expression tag, we are simply displaying a welcome message.

1. **<html>**
2. **<body>**
3. **<**%= "welcome to jsp" %**>**
4. **</body>**
5. **</html>**

#### Note: Do not end your statement with semicolon in case of expression tag.

### Example of JSP expression tag that prints current time

To display the current time, we have used the getTime() method of Calendar class. The getTime() is an instance method of Calendar class, so we have called it after getting the instance of Calendar class by the getInstance() method.

*index.jsp*

1. **<html>**
2. **<body>**
3. Current Time: **<**%= java.util.Calendar.getInstance().getTime() %**>**
4. **</body>**
5. **</html>**

### Example of JSP expression tag that prints the user name

In this example, we are printing the username using the expression tag. The index.html file gets the username and sends the request to the welcome.jsp file, which displays the username.

*File: index.jsp*

1. **<html>**
2. **<body>**
3. **<form** action="welcome.jsp"**>**
4. **<input** type="text" name="uname"**><br/>**
5. **<input** type="submit" value="go"**>**
6. **</form>**
7. **</body>**
8. **</html>**

*File: welcome.jsp*

1. **<html>**
2. **<body>**
3. **<**%= "Welcome "+request.getParameter("uname") %**>**
4. **</body>**
5. **</html>**

# JSP Declaration Tag

1. [JSP declaration tag](https://www.javatpoint.com/jsp-declaration-tag)
2. [Difference between JSP scriptlet tag and JSP declaration tag](https://www.javatpoint.com/jsp-declaration-tag#diff)
3. [Example of JSP declaration tag that declares field](https://www.javatpoint.com/jsp-declaration-tag#declarationex1)
4. [Example of JSP declaration tag that declares method](https://www.javatpoint.com/jsp-declaration-tag#declarationex2)

The **JSP declaration tag** is used to declare fields and methods.

The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.

So it doesn't get memory at each request.

#### Syntax of JSP declaration tag

The syntax of the declaration tag is as follows:

1. **<**%!  field or method declaration %**>**

### Difference between JSP Scriptlet tag and Declaration tag

|  |  |
| --- | --- |
| **Jsp Scriptlet Tag** | **Jsp Declaration Tag** |
| The jsp scriptlet tag can only declare variables not methods. | The jsp declaration tag can declare variables as well as methods. |
| The declaration of scriptlet tag is placed inside the \_jspService() method. | The declaration of jsp declaration tag is placed outside the \_jspService() method. |

### Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

### index.jsp

1. **<html>**
2. **<body>**
3. **<**%! int data=50; %**>**
4. **<**%= "Value of the variable is:"+data %**>**
5. **</body>**
6. **</html>**

### Example of JSP declaration tag that declares method

In this example of JSP declaration tag, we are defining the method which returns the cube of given number and calling this method from the jsp expression tag. But we can also use jsp scriptlet tag to call the declared method.

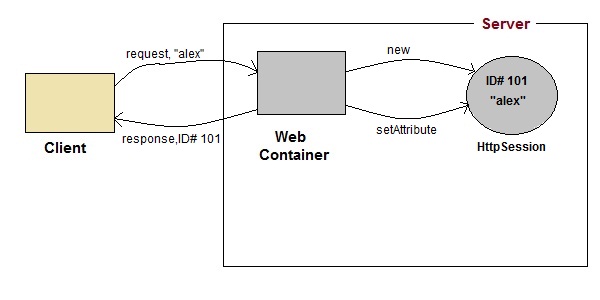
### index.jsp

1. **<html>**
2. **<body>**
3. **<**%!
4. int cube(int n){
5. return n\*n\*n\*;
6. }
7. %**>**
8. **<**%= "Cube of 3 is:"+cube(3) %**>**
9. **</body>**
10. **</html>**

# Servlet: What is HttpSession?

**HttpSession** object is used to store entire session with a specific client. We can store, retrieve and remove attribute from **HttpSession** object. Any servlet can have access to **HttpSession** object throughout the getSession() method of the **HttpServletRequest** object.

## Servlet: How HttpSession works



1. On client's first request, the **Web Container** generates a unique session ID and gives it back to the client with response. This is a temporary session created by web container.
2. The client sends back the session ID with each request. Making it easier for the web container to identify where the request is coming from.
3. The **Web Container** uses this ID, finds the matching session with the ID and associates the session with the request.

## Servlet: HttpSession Interface



### Some Important Methods of Servlet HttpSession

|  |  |
| --- | --- |
| **Methods** | **Description** |
| long getCreationTime() | returns the time when the session was created, measured in milliseconds since midnight January 1, 1970 GMT. |
| String getId() | returns a string containing the unique identifier assigned to the session. |
| long getLastAccessedTime() | returns the last time the client sent a request associated with the session |
| int getMaxInactiveInterval() | returns the maximum time interval, in seconds. |
| void invalidate() | destroy the session |
| boolean isNew() | returns true if the session is new else false |
| void setMaxInactiveInterval(int interval) | Specifies the time, in seconds,after servlet container will invalidate the session. |

### Complete Example demonstrating usage of HttpSession

All the files mentioned below are required for the example.

**index.html**

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

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

Password: <input type="text" name="pass" ><br/>

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

</form>

**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-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>

**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");

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

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

if(pass.equals("1234"))

{

//creating a session

HttpSession session = request.getSession();

session.setAttribute("user", name);

response.sendRedirect("Welcome");

}

}

}

**Welcome.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class Welcome extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

HttpSession session = request.getSession();

String user = (String)session.getAttribute("user");

out.println("Hello "+user);

}

}

# Using Cookies for Session Management in Servlet

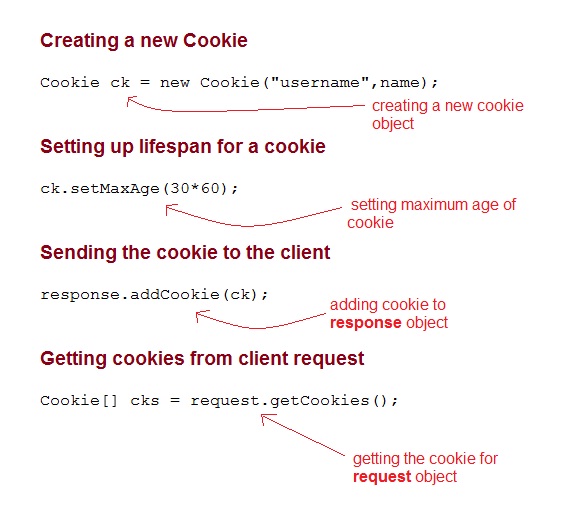
**Cookies** are small pieces of information that are sent in response from the web server to the client. **Cookies** are the simplest technique used for storing client state.

**Cookies** are stored on client's computer. They have a lifespan and are destroyed by the client browser at the end of that lifespan.

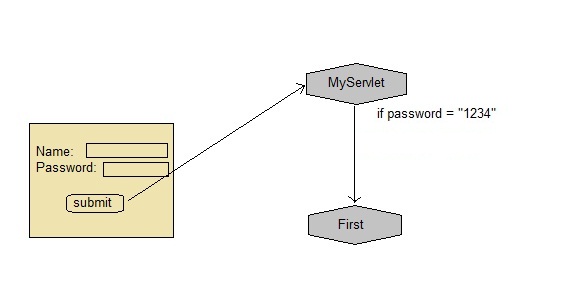
Using Cookies for storing client state has one shortcoming though, if the client has turned of COokie saving settings in his browser then, client state can never be saved because the browser will not allow the application to store cookies.

## Servlet: Cookies API

Cookies are created using **Cookie** class present in Servlet API. Cookies are added to **response** object using the addCookie() method. This method sends cookie information over the HTTP response stream. getCookies() method is used to access the cookies that are added to response object.



### Example demonstrating usage of Cookies



Below mentioned files are required for the example:

**index.html**

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

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

Password:<input type="text" name="pass" ><br/>

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

</form>

**web.xml**

<web-app...>

<servlet>

<servlet-name>validate</servlet-name>

<servlet-class>MyServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>validate</servlet-name>

<url-pattern>/validate</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>First</servlet-name>

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

</servlet>

<servlet-mapping>

<servlet-name>First</servlet-name>

<url-pattern>/First</url-pattern>

</servlet-mapping>

<welcome-file-list>

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

</welcome-file-list>

</web-app>

**MyServlet.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class MyServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html;charset=UTF-8");

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

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

if(pass.equals("1234"))

{

Cookie ck = new Cookie("username", name);

response.addCookie(ck);

response.sendRedirect("First");

}

}

}

**First.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class First extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

Cookie[] cks = request.getCookies();

out.println("Welcome "+ cks[0].getValue());

}

}

# Servlet Filter

1. [Filter](https://www.javatpoint.com/servlet-filter)
2. [Usage of Filter](https://www.javatpoint.com/servlet-filter#filterusage)
3. [Advantage of Filter](https://www.javatpoint.com/servlet-filter#filteradvantage)
4. [Filter API](https://www.javatpoint.com/servlet-filter#filterapi)
   1. [Filter interface](https://www.javatpoint.com/servlet-filter#filterinterface)
   2. [FilterChain interface](https://www.javatpoint.com/servlet-filter#filterchain)
   3. [FilterConfig interface](https://www.javatpoint.com/servlet-filter#filterconfig)
5. [Simple Example of Filter](https://www.javatpoint.com/servlet-filter#filterex)

A **filter** is an object that is invoked at the preprocessing and postprocessing of a request.

It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc.

The **servlet filter is pluggable**, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the servlet.

So maintenance cost will be less.

#### Note: Unlike Servlet, One filter doesn't have dependency on another filter.

### Usage of Filter

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

### Advantage of Filter

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

### Filter API

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

1. Filter
2. FilterChain
3. FilterConfig

### 1) Filter interface

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

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

### 2) FilterChain interface

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

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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next>>**](https://www.javatpoint.com/authentication-filter)[**<<prev**](https://www.javatpoint.com/http-session-in-session-tracking) Servlet Filter  1. [Filter](https://www.javatpoint.com/servlet-filter) 2. [Usage of Filter](https://www.javatpoint.com/servlet-filter#filterusage) 3. [Advantage of Filter](https://www.javatpoint.com/servlet-filter#filteradvantage) 4. [Filter API](https://www.javatpoint.com/servlet-filter#filterapi)    1. [Filter interface](https://www.javatpoint.com/servlet-filter#filterinterface)    2. [FilterChain interface](https://www.javatpoint.com/servlet-filter#filterchain)    3. [FilterConfig interface](https://www.javatpoint.com/servlet-filter#filterconfig) 5. [Simple Example of Filter](https://www.javatpoint.com/servlet-filter#filterex)   A **filter** is an object that is invoked at the preprocessing and postprocessing of a request.  It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc.  The **servlet filter is pluggable**, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the servlet.  So maintenance cost will be less. Note: Unlike Servlet, One filter doesn't have dependency on another filter.Usage of Filter  * recording all incoming requests * logs the IP addresses of the computers from which the requests originate * conversion * data compression * encryption and decryption * input validation etc.  Advantage of Filter  1. Filter is pluggable. 2. One filter don't have dependency onto another resource. 3. Less Maintenance  Filter API Like servlet filter have its own API. The javax.servlet package contains the three interfaces of Filter API.   1. Filter 2. FilterChain 3. FilterConfig  1) Filter interface For creating any filter, you must implement the Filter interface. Filter interface provides the life cycle methods for a filter.   |  |  | | --- | --- | | **Method** | **Description** | | public void init(FilterConfig config) | init() method is invoked only once. It is used to initialize the filter. | | public void doFilter(HttpServletRequest request,HttpServletResponse response, FilterChain chain) | doFilter() method is invoked every time when user request to any resource, to which the filter is mapped.It is used to perform filtering tasks. | | public void destroy() | This is invoked only once when filter is taken out of the service. |  2) FilterChain interface The object of FilterChain is responsible to invoke the next filter or resource in the chain.This object is passed in the doFilter method of Filter interface.The FilterChain interface contains only one method:   1. **public void doFilter(HttpServletRequest request, HttpServletResponse response):** it passes the control to the next filter or resource.  How to define Filter We can define filter same as servlet. Let's see the elements of filter and filter-mapping.   1. <web-app> 3. <filter> 4. <filter-name>...</filter-name> 5. <filter-**class**>...</filter-**class**> 6. </filter> 8. <filter-mapping> 9. <filter-name>...</filter-name> 10. <url-pattern>...</url-pattern> 11. </filter-mapping> 13. </web-app>   For mapping filter we can use, either url-pattern or servlet-name. The url-pattern elements has an advantage over servlet-name element i.e. it can be applied on servlet, JSP or HTML. Simple Example of Filter In this example, we are simply displaying information that filter is invoked automatically after the post processing of the request. index.html  1. <a href="servlet1">click here</a>  MyFilter.java  1. **import** java.io.IOException; 2. **import** java.io.PrintWriter; 4. **import** javax.servlet.\*; 6. **public** **class** MyFilter **implements** Filter{ 8. **public** **void** init(FilterConfig arg0) **throws** ServletException {} 10. **public** **void** doFilter(ServletRequest req, ServletResponse resp, 11. FilterChain chain) **throws** IOException, ServletException { 13. PrintWriter out=resp.getWriter(); 14. out.print("filter is invoked before"); 16. chain.doFilter(req, resp);//sends request to next resource 18. out.print("filter is invoked after"); 19. } 20. **public** **void** destroy() {} 21. }  HelloServlet.java  1. **import** java.io.IOException; 2. **import** java.io.PrintWriter; 4. **import** javax.servlet.ServletException; 5. **import** javax.servlet.http.\*; 7. **public** **class** HelloServlet **extends** HttpServlet { 8. **public** **void** doGet(HttpServletRequest request, HttpServletResponse response) 9. **throws** ServletException, IOException { 11. response.setContentType("text/html"); 12. PrintWriter out = response.getWriter(); 14. out.print("<br>welcome to servlet<br>"); 16. } 18. }   **web.xml**   |  | | --- | | For defining the filter, filter element of web-app must be defined just like servlet. |  1. <web-app> 3. <servlet> 4. <servlet-name>s1</servlet-name> 5. <servlet-**class**>HelloServlet</servlet-**class**> 6. </servlet> 8. <servlet-mapping> 9. <servlet-name>s1</servlet-name> 10. <url-pattern>/servlet1</url-pattern> 11. </servlet-mapping> 13. <filter> 14. <filter-name>f1</filter-name> 15. <filter-**class**>MyFilter</filter-**class**> 16. </filter> 18. <filter-mapping> 19. <filter-name>f1</filter-name> 20. <url-pattern>/servlet1</url-pattern> 21. </filter-mapping>  24. </web-app> |

**ResourceBundle**

The Java ResourceBundle class, java.util.ResourceBundle, is used to store texts and components that are locale sensitive. For instance, the text labels used inside your application might need to change depending on the language of the user currently using your application. The text labels are thus said to be user locale sensitive. A user's locale is represented by the [**Java Locale**](http://tutorials.jenkov.com/java-internationalization/locale.html) class, by the way. This text takes a closer look at the ResourceBundle class and its subclasses.

## The ResourceBundle Class Hierarchy

The ResourceBundle class has two subclasses called PropertyResourceBundle and ListResourceBundle. Here is a diagram illustrating the class hierarchy:

|  |
| --- |
| ResourceBundle has the subclasses PropertiesResourceBundle and ListResourceBundle. |
| **ResourceBundle has the subclasses PropertiesResourceBundle and ListResourceBundle.** |

The PropertyResourceBundle class stores localized texts in standard Java property files. The format of these files is explained in my [**Java Properties**](http://tutorials.jenkov.com/java-collections/properties.html) tutorial.

You do not directly interact with these two subclasses. All interaction goes through the ResourceBundle class.

## Creating a ResourceBundle

You create a ResourceBundle instance like this:

Locale locale = new Locale("en", "US");

ResourceBundle labels = ResourceBundle.getBundle("i18n.MyBundle", locale);

System.out.println(labels.getString("label1"));

First you need a Locale instance. Then you pass that Locale instance to the ResourceBundle.getBundle() method along with the name of the resource bundle to load. Finally you can access the localized values in the ResourceBundle via its different getString() and getObject() etc. methods.

You are never actually creating a ResourceBundle instance, but an instance of one of its two subclasses. Both are created using the above factory method. First the ResourceBundle class will look for a ListResourceBundle, and then for a PropertyResourceBundle. It does so by matching the name of the requested resource bundle (first parameter in the getBundle() method) against the class names of a ListResourceBundle first, and if none found, against a property file resource bundle.

Both ListResourceBundle and PropertyResourceBundle are covered in more detail in the following sections.

## Property Files as ResourceBundle

You can use standard property files for storing localized texts. You can load these properties via the ResourceBundle class. Here is an example:

Locale locale = new Locale("en", "US");

ResourceBundle labels = ResourceBundle.getBundle("i18n.MyBundle", locale);

System.out.println(labels.getString("label1"));

For this example to work you should put a standard Java property file named MyBundle.properties in a Java package named i18n. Make sure this property file is available on your class path when you run the above code, meaning the property file should be located among the classes of your application, and in the i18n package.

The name of a resource bundle is like a class name. Thus, i18n.MyBundle means a property file named MyBundle.properties in the package (directory) i18n.

Here is an example of what the content of the property file could look like:

label1 = Label 1 is done!

label2 = Label 2 is through!

As is the standard with Java property files, it is a list of key and value pairs. The key is on the left side of the = , and the value is on the right side. The value is what you should localize, not the key.

### Different Languages in Different Property Files

In order to provide strings in different languages, create a property file for each language, and suffix them with underscore (\_) and then the language code. For instance:

MyBundle.properties

MyBundle\_da.properties

MyBundle\_de.properties

MyBundle\_fr.properties

All of these files should be located in the same package (directory).

The file without language suffix (e.g. MyBundle.properties) is the default property file. In case no property file is available for the language (Locale) passed to the ResourceBundle.getBundle() method, and the system has no default Locale set (e.g. a German computer will have a German Locale as default), this file is read and returned as a ResourceBundle.

The other property files with the language code suffixes contain the same keys but with values in different languages. Thus, the danish property file could look like this:

label1 = Label 1 er klar!

label2 = Label 2 er igennem!

## Classes as ResourceBundle

You can also use a set of classes to contain your resources. Using classes you can use more than just string values.

Like with the property files, you create a set of classes with a bundle base name and language suffixes. For instance:

i18n.MyClassBundle

i18n.MyClassBundle\_da

i18n.MyClassBundle\_en

i18n.MyClassBundle\_de

Here is an example implementation of the default bundle class file:

package i18n;

import java.util.ListResourceBundle;

public class MyClassBundle extends ListResourceBundle {

@Override

protected Object[][] getContents() {

return contents;

}

private Object[][] contents = {

{ "price" , new Double(10.00) },

{ "currency", "EUR" },

};

}

And here is an implementation of the resource bundle for the danish language:

public class MyClassBundle\_da extends ListResourceBundle {

@Override

protected Object[][] getContents() {

return contents;

}

private Object[][] contents = {

{ "price" , new Double(75.00) },

{ "currency", "DKK" },

};

}

Notice the contents array. It consists of a 2-dimensional array of keys and values. Thus, price and currency are keys, and the values to the right of them are the localized values. These two examples contains a price in two different currencies.

You obtain an instance of a ListResourceBundle the same way you get an instance of a PropertyResourceBundle. Here are two examples that obtain an instance of both the default ResourceBundle and the ResourceBundle for the danish language:

Locale locale = new Locale("de", "DE"); //no bundle for German -> default

ResourceBundle bundle = ResourceBundle.getBundle("i18n.MyClassBundle", locale);

System.out.println("price : " + bundle.getObject("price"));

System.out.println("currency: " + bundle.getObject("currency"));

locale = new Locale("da", "DK");

bundle = ResourceBundle.getBundle("i18n.MyClassBundle", locale);

System.out.println("price : " + bundle.getObject("price"));

System.out.println("currency: " + bundle.getObject("currency"));

The output printed from this code would be:

price : 10.0

currency: EUR

price : 75.0

currency: DKK

JSTL stands for **JSP Standard Tag Library**. JSTL is the standard tag library that provides tags to control the JSP page behavior. JSTL tags can be used for iteration and control statements, internationalization, SQL etc. We will look into JSTL Tags in detail in this JSTL tutorial.

## JSTL

[](https://cdn.journaldev.com/wp-content/uploads/2013/08/jstl-tags-example-tutorial.jpg)

JSTL is part of the Java EE API and included in most servlet containers. But to use JSTL in our JSP pages, we need to download the JSTL jars for your servlet container. Most of the times, you can find them in the example projects of server download and you can use them. You need to include these libraries in your web application project **WEB-INF/lib** directory.

### JSTL jars

JSTL jars are container specific, for example in Tomcat, we need to include jstl.jar and standard.jar jar files in project build path. If they are not present in the container lib directory, you should include them into your application. If you have maven project, below dependencies should be added in pom.xml file or else you will get following error in JSP pages – eclipse Can not find the tag library descriptor for "https://java.sun.com/jsp/jstl/ core"

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>taglibs</groupId>

<artifactId>standard</artifactId>

<version>1.1.2</version>

</dependency>

### JSTL Tags

Based on the JSTL functions, they are categorized into five types.

1. **JSTL Core Tags**: JSTL Core tags provide support for iteration, conditional logic, catch exception, url, forward or redirect response etc. To use JSTL core tags, we should include it in the JSP page like below.
2. <%@ taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c" %>

In this article, we will look into important JSTL core tags.

1. **JSTL Formatting and Localisation Tags**: JSTL Formatting tags are provided for formatting of Numbers, Dates and i18n support through locales and resource bundles. We can include these jstl tags in JSP with below syntax:
2. <%@ taglib uri="https://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
3. **JSTL SQL Tags**: JSTL SQL Tags provide support for interaction with relational databases such as Oracle, MySql etc. Using JSTL SQL tags we can run database queries, we include these JSTL tags in JSP with below syntax:
4. <%@ taglib uri="https://java.sun.com/jsp/jstl/sql" prefix="sql" %>
5. **JSTL XML Tags**: JSTL XML tags are used to work with XML documents such as parsing XML, transforming XML data and XPath expressions evaluation. Syntax to include JSTL XML tags in JSP page is:
6. <%@ taglib uri="https://java.sun.com/jsp/jstl/xml" prefix="x" %>
7. **JSTL Functions Tags**: JSTL tags provide a number of functions that we can use to perform common operation, most of them are for String manipulation such as String Concatenation, Split String etc. Syntax to include JSTL functions in JSP page is:
8. <%@ taglib uri="https://java.sun.com/jsp/jstl/functions" prefix="fn" %>

Note that all the JSTL standard tags URI starts with https://java.sun.com/jsp/jstl/ and we can use any prefix we want but it’s best practice to use the prefix defined above because everybody uses them, so it will not create any confusion.

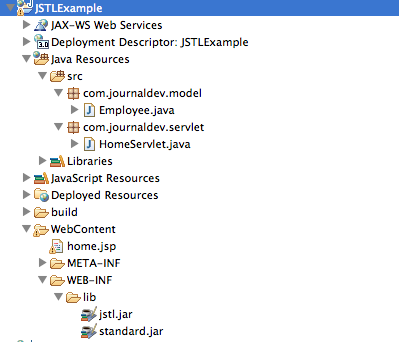
### JSTL Core Tags

JSTL Core Tags are listed in the below table.

|  |  |
| --- | --- |
| JSTL Core Tag | Description |
| <c:out> | To write something in JSP page, we can use EL also with this tag |
| <c:import> | Same as <jsp:include> or include directive |
| <c:redirect> | redirect request to another resource |
| <c:set> | To set the variable value in given scope. |
| <c:remove> | To remove the variable from given scope |
| <c:catch> | To catch the exception and wrap it into an object. |
| <c:if> | Simple conditional logic, used with EL and we can use it to process the exception from <c:catch> |
| <c:choose> | Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <c:when> and <c:otherwise> |
| <c:when> | Subtag of <c:choose> that includes its body if its condition evalutes to ‘true’. |
| <c:otherwise> | Subtag of <c:choose> that includes its body if its condition evalutes to ‘false’. |
| <c:forEach> | for iteration over a collection |
| <c:forTokens> | for iteration over tokens separated by a delimiter. |
| <c:param> | used with <c:import> to pass parameters |
| <c:url> | to create a URL with optional query string parameters |

### JSTL Tutorial

Let’s see some of the JSTL core tags usages in a simple web application. Our project will include a Java Bean and we will create a list of objects and set some attributes that will be used in the JSP. JSP page will show how to iterate over a collection, using conditional logic with EL and some other common usage.

[](https://cdn.journaldev.com/wp-content/uploads/2013/08/JSTL-Tags-Example.png)

### JSTL Tutorial – Java Bean Class

package com.journaldev.model;

public class Employee {

private int id;

private String name;

private String role;

public Employee() {

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getRole() {

return role;

}

public void setRole(String role) {

this.role = role;

}

}

### JSTL Tutorial – Servlet Class

package com.journaldev.servlet;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import com.journaldev.model.Employee;

@WebServlet("/HomeServlet")

public class HomeServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

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

List<Employee> empList = new ArrayList<Employee>();

Employee emp1 = new Employee();

emp1.setId(1); emp1.setName("Pankaj");emp1.setRole("Developer");

Employee emp2 = new Employee();

emp2.setId(2); emp2.setName("Meghna");emp2.setRole("Manager");

empList.add(emp1);empList.add(emp2);

request.setAttribute("empList", empList);

request.setAttribute("htmlTagData", "<br/> creates a new line.");

request.setAttribute("url", "https://www.journaldev.com");

RequestDispatcher rd = getServletContext().getRequestDispatcher("/home.jsp");

rd.forward(request, response);

}

}

### JSTL Tutorial – JSP Page

home.jsp code:

<%@ page language="java" contentType="text/html; charset=US-ASCII"

pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Home Page</title>

<%@ taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c" %>

<style>

table,th,td

{

border:1px solid black;

}

</style>

</head>

<body>

<%-- Using JSTL forEach and out to loop a list and display items in table --%>

<table>

<tbody>

<tr><th>ID</th><th>Name</th><th>Role</th></tr>

<c:forEach items="${requestScope.empList}" var="emp">

<tr><td><c:out value="${emp.id}"></c:out></td>

<td><c:out value="${emp.name}"></c:out></td>

<td><c:out value="${emp.role}"></c:out></td></tr>

</c:forEach>

</tbody>

</table>

<br><br>

<%-- simple c:if and c:out example with HTML escaping --%>

<c:if test="${requestScope.htmlTagData ne null }">

<c:out value="${requestScope.htmlTagData}" escapeXml="true"></c:out>

</c:if>

<br><br>

<%-- c:set example to set variable value --%>

<c:set var="id" value="5" scope="request"></c:set>

<c:out value="${requestScope.id }" ></c:out>

<br><br>

<%-- c:catch example --%>

<c:catch var ="exception">

<% int x = 5/0;%>

</c:catch>

<c:if test = "${exception ne null}">

<p>Exception is : ${exception} <br>

Exception Message: ${exception.message}</p>

</c:if>

<br><br>

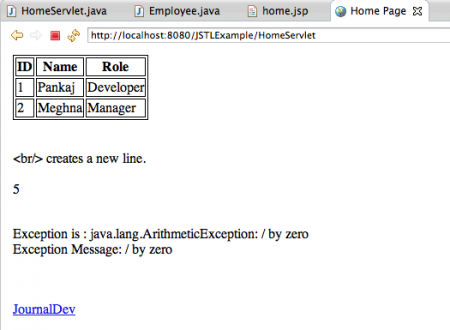
<%-- c:url example --%>

<a href="<c:url value="${requestScope.url }"></c:url>">JournalDev</a>

</body>

</html>

Now when we run application with URL https://localhost:8080/JSTLExample/HomeServlet, we get response as in below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/08/JSTL-Example.png)