**What are Servlets?**

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests 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**

The 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.

# Servlets - Life Cycle

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.

Now let us discuss the life cycle methods in detail.

## The init() Method

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. 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 doGe, 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.

# Servlets - Examples

Servlets are Java classes which service HTTP requests and implement the **javax.servlet.Servlet** interface. Web application developers typically write servlets that extend javax.servlet.http.HttpServlet, an abstract class that implements the Servlet interface and is specially designed to handle HTTP requests.

## Sample Code

Following is the sample source code structure of a servlet example to show Hello World −

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Extend HttpServlet class

public class HelloWorld extends HttpServlet {

private String message;

public void init() throws ServletException {

// Do required initialization

message = "Hello World";

}

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

// Actual logic goes here.

PrintWriter out = response.getWriter();

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

}

public void destroy() {

// do nothing.

}

}

**Compiling a Servlet**

Let us create a file with name HelloWorld.java with the code shown above. Place this file at C:\ServletDevel (in Windows) or at /usr/ServletDevel (in Unix). This path location must be added to CLASSPATH before proceeding further.

Assuming your environment is setup properly, go in **ServletDevel** directory and compile HelloWorld.java as follows −

$ javac HelloWorld.java

If the servlet depends on any other libraries, you have to include those JAR files on your CLASSPATH as well. I have included only servlet-api.jar JAR file because I'm not using any other library in Hello World program.

This command line uses the built-in javac compiler that comes with the Sun Microsystems Java Software Development Kit (JDK). For this command to work properly, you have to include the location of the Java SDK that you are using in the PATH environment variable.

If everything goes fine, above compilation would produce **HelloWorld.class** file in the same directory. Next section would explain how a compiled servlet would be deployed in production.

**Servlet Deployment**

By default, a servlet application is located at the path <Tomcat-installationdirectory>/webapps/ROOT and the class file would reside in <Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes.

If you have a fully qualified class name of **com.myorg.MyServlet**, then this servlet class must be located in WEB-INF/classes/com/myorg/MyServlet.class.

For now, let us copy HelloWorld.class into <Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes and create following entries in **web.xml** file located in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/

<servlet>

<servlet-name>HelloWorld</servlet-name>

<servlet-class>HelloWorld</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloWorld</servlet-name>

<url-pattern>/HelloWorld</url-pattern>

</servlet-mapping>

Above entries to be created inside <web-app>...</web-app> tags available in web.xml file. There could be various entries in this table already available, but never mind.

You are almost done, now let us start tomcat server using <Tomcat-installationdirectory>\bin\startup.bat (on Windows) or <Tomcat-installationdirectory>/bin/startup.sh (on Linux/Solaris etc.) and finally type **http://localhost:8080/HelloWorld** in the browser's address box. If everything goes fine, you would get the following result

# Servlets - Form Data

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your backend program. The browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

## GET Method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the **?** (question mark) symbol as follows −

http://www.test.com/hello?key1 = value1&key2 = value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation: only 1024 characters can be used in a request string.

This information is passed using QUERY\_STRING header and will be accessible through QUERY\_STRING environment variable and Servlet handles this type of requests using **doGet()** method.

## POST Method

A generally more reliable method of passing information to a backend program is the POST method. This packages the information in exactly the same way as GET method, but instead of sending it as a text string after a ? (question mark) in the URL it sends it as a separate message. This message comes to the backend program in the form of the standard input which you can parse and use for your processing. Servlet handles this type of requests using **doPost()** method.

## 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.

## GET Method Example using URL

Here is a simple URL which will pass two values to HelloForm program using GET method.

**http://localhost:8080/HelloForm?first\_name = ZARA&last\_name = ALI**

Given below is the **HelloForm.java** servlet program to handle input given by web browser. We are going to use **getParameter()** method which makes it very easy to access passed information −

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Extend HttpServlet class

public class HelloForm extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "Using GET Method to Read Form Data";

String docType =

"<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

out.println(docType +

"<html>\n" +

"<head><title>" + title + "</title></head>\n" +

"<body bgcolor = \"#f0f0f0\">\n" +

"<h1 align = \"center\">" + title + "</h1>\n" +

"<ul>\n" +

" <li><b>First Name</b>: "

+ request.getParameter("first\_name") + "\n" +

" <li><b>Last Name</b>: "

+ request.getParameter("last\_name") + "\n" +

"</ul>\n" +

"</body>

</html>"

);

}

}

Assuming your environment is set up properly, compile HelloForm.java as follows −

$ javac HelloForm.java

If everything goes fine, above compilation would produce **HelloForm.class** file. Next you would have to copy this class file in <Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes and create following entries in **web.xml** file located in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/

<servlet>

<servlet-name>HelloForm</servlet-name>

<servlet-class>HelloForm</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloForm</servlet-name>

<url-pattern>/HelloForm</url-pattern>

</servlet-mapping>

Now type *http://localhost:8080/HelloForm?first\_name=ZARA&last\_name=ALI* in your browser's Location:box and make sure you already started tomcat server, before firing above command in the browser. This would generate following result −

# Using GET Method to Read Form Data

* **First Name**: ZARA
* **Last Name**: ALI

## Reading All Form Parameters

Following is the generic example which uses **getParameterNames()** method of HttpServletRequest to read all the available form parameters. This method returns an Enumeration that contains the parameter names in an unspecified order

Once we have an Enumeration, we can loop down the Enumeration in standard way by, using *hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name.

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

// Extend HttpServlet class

public class ReadParams extends HttpServlet {

// Method to handle GET method request.

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "Reading All Form Parameters";

String docType =

"<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

out.println(docType +

"<html>\n" +

"<head><title>" + title + "</title></head>\n" +

"<body bgcolor = \"#f0f0f0\">\n" +

"<h1 align = \"center\">" + title + "</h1>\n" +

"<table width = \"100%\" border = \"1\" align = \"center\">\n" +

"<tr bgcolor = \"#949494\">\n" +

"<th>Param Name</th>

<th>Param Value(s)</th>\n"+

"</tr>\n"

);

Enumeration paramNames = request.getParameterNames();

while(paramNames.hasMoreElements()) {

String paramName = (String)paramNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n<td>");

String[] paramValues = request.getParameterValues(paramName);

// Read single valued data

if (paramValues.length == 1) {

String paramValue = paramValues[0];

if (paramValue.length() == 0)

out.println("<i>No Value</i>");

else

out.println(paramValue);

} else {

// Read multiple valued data

out.println("<ul>");

for(int i = 0; i < paramValues.length; i++) {

out.println("<li>" + paramValues[i]);

}

out.println("</ul>");

}

}

out.println("</tr>\n</table>\n</body></html>");

}

// Method to handle POST method request.

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response);

}

}

Now, try the above servlet with the following form −

<html>

<body>

<form action = "ReadParams" method = "POST" target = "\_blank">

<input type = "checkbox" name = "maths" checked = "checked" /> Maths

<input type = "checkbox" name = "physics" /> Physics

<input type = "checkbox" name = "chemistry" checked = "checked" /> Chem

<input type = "submit" value = "Select Subject" />

</form>

</body>

</html>

Now calling servlet using the above form would generate the following result −

# Reading All Form Parameters

|  |  |
| --- | --- |
| **Param Name** | **Param Value(s)** |
| maths | on |
| chemistry | on |

You can try the above servlet to read any other form's data having other objects like text box, radio button or drop down box etc.

# Servlets - Client HTTP Request

When a browser requests for a web page, it sends lot of information to the web server which cannot be read directly because this information travel as a part of header of HTTP request. You can check [HTTP Protocol](https://www.tutorialspoint.com/http/index.htm) for more information on this.

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

|  |  |
| --- | --- |
| **Sr.No.** | **Header & Description** |
| 1 | **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. |
| 2 | **Accept-Charset**  This header specifies the character sets the browser can use to display the information. For example ISO-8859-1. |
| 3 | **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. |
| 4 | **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 |
| 5 | **Authorization**  This header is used by clients to identify themselves when accessing password-protected Web pages. |
| 6 | **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. |
| 7 | **Content-Length**  This header is applicable only to POST requests and gives the size of the POST data in bytes. |
| 8 | **Cookie**  This header returns cookies to servers that previously sent them to the browser. |
| 9 | **Host**  This header specifies the host and port as given in the original URL. |
| 10 | **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. |
| 11 | **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. |
| 12 | **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 Referrer header when the browser requests Web page 2. |
| 13 | **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. |

**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

|  |  |
| --- | --- |
| **Sr.No.** | **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 if there is no current session and value of 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 | **int getIntHeader(String name)**  Returns the value of the specified request header as an int. |
| 30 | **int getServerPort()**  Returns the port number on which this request was received. |

## HTTP Header Request Example

Following is the example which uses **getHeaderNames()** method of HttpServletRequest to read the HTTP header information. This method returns an Enumeration that contains the header information associated with the current HTTP request.

Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using *hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

// Extend HttpServlet class

public class DisplayHeader extends HttpServlet {

// Method to handle GET method request.

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "HTTP Header Request Example";

String docType =

"<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

out.println(docType +

"<html>\n" +

"<head><title>" + title + "</title></head>\n"+

"<body bgcolor = \"#f0f0f0\">\n" +

"<h1 align = \"center\">" + title + "</h1>\n" +

"<table width = \"100%\" border = \"1\" align = \"center\">\n" +

"<tr bgcolor = \"#949494\">\n" +

"<th>Header Name</th><th>Header Value(s)</th>\n"+

"</tr>\n"

);

Enumeration headerNames = request.getHeaderNames();

while(headerNames.hasMoreElements()) {

String paramName = (String)headerNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n");

String paramValue = request.getHeader(paramName);

out.println("<td> " + paramValue + "</td></tr>\n");

}

out.println("</table>\n</body></html>");

}

// Method to handle POST method request.

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response);

}

}

# HTTP Header Request Example

|  |  |
| --- | --- |
| **Header Name** | **Header Value(s)** |
| accept | \*/\* |
| accept-language | en-us |
| user-agent | Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; MS-RTC LM 8) |
| accept-encoding | gzip, deflate |
| host | localhost:8080 |
| connection | Keep-Alive |
| cache-control | no-cache |

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example).

Following is a list of HTTP status codes and associated messages that might be returned from the Web Server −

|  |  |  |
| --- | --- | --- |
| **Code** | **Message** | **Description** |
| 100 | Continue | Only a part of the request has been received by the server, but as long as it has not been rejected, the client should continue with the request |
| 101 | Switching Protocols | The server switches protocol. |
| 200 | OK | The request is OK |
| 201 | Created | The request is complete, and a new resource is created |
| 202 | Accepted | The request is accepted for processing, but the processing is not complete. |
| 203 | Non-authoritative Information |  |
| 204 | No Content |  |
| 205 | Reset Content |  |
| 206 | Partial Content |  |
| 300 | Multiple Choices | A link list. The user can select a link and go to that location. Maximum five addresses |
| 301 | Moved Permanently | The requested page has moved to a new url |
| 302 | Found | The requested page has moved temporarily to a new url |
| 303 | See Other | The requested page can be found under a different url |
| 304 | Not Modified |  |
| 305 | Use Proxy |  |
| 306 | *Unused* | This code was used in a previous version. It is no longer used, but the code is reserved |
| 307 | Temporary Redirect | The requested page has moved temporarily to a new url. |
| 400 | Bad Request | The server did not understand the request |
| 401 | Unauthorized | The requested page needs a username and a password |
| 402 | Payment Required | *You cannot use this code yet* |
| 403 | Forbidden | Access is forbidden to the requested page |
| 404 | Not Found | The server cannot find the requested page. |
| 405 | Method Not Allowed | The method specified in the request is not allowed. |
| 406 | Not Acceptable | The server can only generate a response that is not accepted by the client. |
| 407 | Proxy Authentication Required | You must authenticate with a proxy server before this request can be served. |
| 408 | Request Timeout | The request took longer than the server was prepared to wait. |
| 409 | Conflict | The request could not be completed because of a conflict. |
| 410 | Gone | The requested page is no longer available. |
| 411 | Length Required | The "Content-Length" is not defined. The server will not accept the request without it. |
| 412 | Precondition Failed | The precondition given in the request evaluated to false by the server. |
| 413 | Request Entity Too Large | The server will not accept the request, because the request entity is too large. |
| 414 | Request-url Too Long | The server will not accept the request, because the url is too long. Occurs when you convert a "post" request to a "get" request with a long query information. |
| 415 | Unsupported Media Type | The server will not accept the request, because the media type is not supported. |
| 417 | Expectation Failed |  |
| 500 | Internal Server Error | The request was not completed. The server met an unexpected condition. |
| 501 | Not Implemented | The request was not completed. The server did not support the functionality required. |
| 502 | Bad Gateway | The request was not completed. The server received an invalid response from the upstream server. |
| 503 | Service Unavailable | The request was not completed. The server is temporarily overloading or down. |
| 504 | Gateway Timeout | The gateway has timed out. |
| 505 | HTTP Version Not Supported | The server does not support the "http protocol" version. |

**Methods to Set HTTP Status Code**

The following methods can be used to set HTTP Status Code in your servlet program. These methods are available with *HttpServletResponse* object.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void setStatus ( int statusCode )**  This method sets an arbitrary status code. The setStatus method takes an int (the status code) as an argument. If your response includes a special status code and a document, be sure to call setStatus before actually returning any of the content with the *PrintWriter*. |
| 2 | **public void sendRedirect(String url)**  This method generates a 302 response along with a *Location* header giving the URL of the new document |
| 3 | **public void sendError(int code, String message)**  This method sends a status code (usually 404) along with a short message that is automatically formatted inside an HTML document and sent to the client. |

## HTTP Status Code Example

Following is the example which would send a 407 error code to the client browser and browser would show you "Need authentication!!!" message.

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

// Extend HttpServlet class

public class showError extends HttpServlet {

// Method to handle GET method request.

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set error code and reason.

response.sendError(407, "Need authentication!!!" );

}

// Method to handle POST method request.

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response);

}

}

Now calling the above servlet would display the following result −

# HTTP Status 407 - Need authentication!!!

**type** Status report

**message**Need authentication!!!

**description**The client must first authenticate itself with the proxy (Need authentication!!!).

### Apache Tomcat/5.5.29

# Servlets - Writing Filters

Servlet Filters are Java classes that can be used in Servlet Programming for the following purposes −

* To intercept requests from a client before they access a resource at back end.
* To manipulate responses from server before they are sent back to the client.

There are various types of filters suggested by the specifications −

* Authentication Filters.
* Data compression Filters.
* Encryption Filters.
* Filters that trigger resource access events.
* Image Conversion Filters.
* Logging and Auditing Filters.
* MIME-TYPE Chain Filters.
* Tokenizing Filters .
* XSL/T Filters That Transform XML Content.

Filters are deployed in the deployment descriptor file **web.xml** and then map to either servlet names or URL patterns in your application's deployment descriptor.

When the web container starts up your web application, it creates an instance of each filter that you have declared in the deployment descriptor. The filters execute in the order that they are declared in the deployment descriptor.

## Servlet Filter Methods

A filter is simply a Java class that implements the javax.servlet.Filter interface. The javax.servlet.Filter interface defines three methods −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void doFilter (ServletRequest, ServletResponse, FilterChain)**  This method is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain. |
| 2 | **public void init(FilterConfig filterConfig)**  This method is called by the web container to indicate to a filter that it is being placed into service. |
| 3 | **public void destroy()**  This method is called by the web container to indicate to a filter that it is being taken out of service. |

## Servlet Filter − Example

Following is the Servlet Filter Example that would print the clients IP address and current date time. This example would give you basic understanding of Servlet Filter, but you can write more sophisticated filter applications using the same concept −

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

// Implements Filter class

public class LogFilter implements Filter {

public void init(FilterConfig config) throws ServletException {

// Get init parameter

String testParam = config.getInitParameter("test-param");

//Print the init parameter

System.out.println("Test Param: " + testParam);

}

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) throws java.io.IOException, ServletException {

// Get the IP address of client machine.

String ipAddress = request.getRemoteAddr();

// Log the IP address and current timestamp.

System.out.println("IP "+ ipAddress + ", Time " + new Date().toString());

// Pass request back down the filter chain

chain.doFilter(request,response);

}

public void destroy( ) {

/\* Called before the Filter instance is removed from service by the web container\*/

}

}

Compile **LogFilter.java** in usual way and put your class file in <Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes

## Servlet Filter Mapping in Web.xml

Filters are defined and then mapped to a URL or Servlet, in much the same way as Servlet is defined and then mapped to a URL pattern. Create the following entry for filter tag in the deployment descriptor file **web.xml**

<filter>

<filter-name>LogFilter</filter-name>

<filter-class>LogFilter</filter-class>

<init-param>

<param-name>test-param</param-name>

<param-value>Initialization Paramter</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>LogFilter</filter-name>

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

</filter-mapping>

The above filter would apply to all the servlets because we specified **/\*** in our configuration. You can specicy a particular servlet path if you want to apply filter on few servlets only.

Now try to call any servlet in usual way and you would see generated log in your web server log. You can use Log4J logger to log above log in a separate file.

## Using Multiple Filters

Your web application may define several different filters with a specific purpose. Consider, you define two filters *AuthenFilter* and *LogFilter*. Rest of the process would remain as explained above except you need to create a different mapping as mentioned below −

<filter>

<filter-name>LogFilter</filter-name>

<filter-class>LogFilter</filter-class>

<init-param>

<param-name>test-param</param-name>

<param-value>Initialization Paramter</param-value>

</init-param>

</filter>

<filter>

<filter-name>AuthenFilter</filter-name>

<filter-class>AuthenFilter</filter-class>

<init-param>

<param-name>test-param</param-name>

<param-value>Initialization Paramter</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>LogFilter</filter-name>

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

</filter-mapping>

<filter-mapping>

<filter-name>AuthenFilter</filter-name>

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

</filter-mapping>

## Filters Application Order

The order of filter-mapping elements in web.xml determines the order in which the web container applies the filter to the servlet. To reverse the order of the filter, you just need to reverse the filter-mapping elements in the web.xml file.

For example, above example would apply LogFilter first and then it would apply AuthenFilter to any servlet but the following example would reverse the order −

<filter-mapping>

<filter-name>AuthenFilter</filter-name>

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

</filter-mapping>

<filter-mapping>

<filter-name>LogFilter</filter-name>

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

</filter-mapping>

# Servlets - Exception Handling

When a servlet throws an exception, the web container searches the configurations in **web.xml** that use the exception-type element for a match with the thrown exception type.

You would have to use the **error-page** element in web.xml to specify the invocation of servlets in response to certain **exceptions** or HTTP **status codes**.

## web.xml Configuration

Consider, you have an *ErrorHandler* servlet which would be called whenever there is any defined exception or error. Following would be the entry created in web.xml.

<!-- servlet definition -->

<servlet>

<servlet-name>ErrorHandler</servlet-name>

<servlet-class>ErrorHandler</servlet-class>

</servlet>

<!-- servlet mappings -->

<servlet-mapping>

<servlet-name>ErrorHandler</servlet-name>

<url-pattern>/ErrorHandler</url-pattern>

</servlet-mapping>

<!-- error-code related error pages -->

<error-page>

<error-code>404</error-code>

<location>/ErrorHandler</location>

</error-page>

<error-page>

<error-code>403</error-code>

<location>/ErrorHandler</location>

</error-page>

<!-- exception-type related error pages -->

<error-page>

<exception-type>

javax.servlet.ServletException

</exception-type >

<location>/ErrorHandler</location>

</error-page>

<error-page>

<exception-type>java.io.IOException</exception-type >

<location>/ErrorHandler</location>

</error-page>

If you want to have a generic Error Handler for all the exceptions then you should define following error-page instead of defining separate error-page elements for every exception −

<error-page>

<exception-type>java.lang.Throwable</exception-type >

<location>/ErrorHandler</location>

</error-page>

Following are the points to be noted about above web.xml for Exception Handling −

* The servlet ErrorHandler is defined in usual way as any other servlet and configured in web.xml.
* If there is any error with status code either 404 (Not Found) or 403 (Forbidden ), then ErrorHandler servlet would be called.
* If the web application throws either ServletException or IOException, then the web container invokes the /ErrorHandler servlet.
* You can define different Error Handlers to handle different type of errors or exceptions. Above example is very much generic and hope it serve the purpose to explain you the basic concept.

# Servlets - Cookies Handling

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users −

* Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
* Browser stores this information on local machine for future use.
* When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them.

## The Anatomy of a Cookie

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A servlet that sets a cookie might send headers that look something like this −

HTTP/1.1 200 OK

Date: Fri, 04 Feb 2000 21:03:38 GMT

Server: Apache/1.3.9 (UNIX) PHP/4.0b3

Set-Cookie: name = xyz; expires = Friday, 04-Feb-07 22:03:38 GMT;

path = /; domain = tutorialspoint.com

Connection: close

Content-Type: text/html

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this −

GET / HTTP/1.0

Connection: Keep-Alive

User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)

Host: zink.demon.co.uk:1126

Accept: image/gif, \*/\*

Accept-Encoding: gzip

Accept-Language: en

Accept-Charset: iso-8859-1,\*,utf-8

Cookie: name = xyz

A servlet will then have access to the cookie through the request method *request.getCookies()* which returns an array of *Cookie* objects.

## Servlet Cookies Methods

Following is the list of useful methods which you can use while manipulating cookies in servlet.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void setDomain(String pattern)**  This method sets the domain to which cookie applies, for example tutorialspoint.com. |
| 2 | **public String getDomain()**  This method gets the domain to which cookie applies, for example tutorialspoint.com. |
| 3 | **public void setMaxAge(int expiry)**  This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session. |
| 4 | **public int getMaxAge()**  This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown. |
| 5 | **public String getName()**  This method returns the name of the cookie. The name cannot be changed after creation. |
| 6 | **public void setValue(String newValue)**  This method sets the value associated with the cookie |
| 7 | **public String getValue()**  This method gets the value associated with the cookie. |
| 8 | **public void setPath(String uri)**  This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories. |
| 9 | **public String getPath()**  This method gets the path to which this cookie applies. |
| 10 | **public void setSecure(boolean flag)**  This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections. |
| 11 | **public void setComment(String purpose)**  This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user. |
| 12 | **public String getComment()**  This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment. |

## Setting Cookies with Servlet

Setting cookies with servlet involves three steps −

**(1) Creating a Cookie object** − You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

Cookie cookie = new Cookie("key","value");

Keep in mind, neither the name nor the value should contain white space or any of the following characters −

[ ] ( ) = , " / ? @ : ;

**(2) Setting the maximum age** − You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

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

**(3) Sending the Cookie into the HTTP response headers** − You use response.addCookie to add cookies in the HTTP response header as follows −

response.addCookie(cookie);

### Example

Let us modify our [Form Example](https://www.tutorialspoint.com/servlets/servlets-form-data.htm) to set the cookies for first and last name.

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Extend HttpServlet class

public class HelloForm extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Create cookies for first and last names.

Cookie firstName = new Cookie("first\_name", request.getParameter("first\_name"));

Cookie lastName = new Cookie("last\_name", request.getParameter("last\_name"));

// Set expiry date after 24 Hrs for both the cookies.

firstName.setMaxAge(60\*60\*24);

lastName.setMaxAge(60\*60\*24);

// Add both the cookies in the response header.

response.addCookie( firstName );

response.addCookie( lastName );

// Set response content type

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "Setting Cookies Example";

String docType =

"<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

out.println(docType +

"<html>\n" +

"<head>

<title>" + title + "</title>

</head>\n" +

"<body bgcolor = \"#f0f0f0\">\n" +

"<h1 align = \"center\">" + title + "</h1>\n" +

"<ul>\n" +

" <li><b>First Name</b>: "

+ request.getParameter("first\_name") + "\n" +

" <li><b>Last Name</b>: "

+ request.getParameter("last\_name") + "\n" +

"</ul>\n" +

"</body>

</html>"

);

}

}

## Reading Cookies with Servlet

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the **getCookies()** method of *HttpServletRequest*. Then cycle through the array, and use getName() and getValue() methods to access each cookie and associated value.

### Example

Let us read cookies which we have set in previous example −

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Extend HttpServlet class

public class ReadCookies extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

Cookie cookie = null;

Cookie[] cookies = null;

// Get an array of Cookies associated with this domain

cookies = request.getCookies();

// Set response content type

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "Reading Cookies Example";

String docType =

"<!doctype html public \"-//w3c//dtd html 4.0 " +

"transitional//en\">\n";

out.println(docType +

"<html>\n" +

"<head><title>" + title + "</title></head>\n" +

"<body bgcolor = \"#f0f0f0\">\n" );

if( cookies != null ) {

out.println("<h2> Found Cookies Name and Value</h2>");

for (int i = 0; i < cookies.length; i++) {

cookie = cookies[i];

out.print("Name : " + cookie.getName( ) + ", ");

out.print("Value: " + cookie.getValue( ) + " <br/>");

}

} else {

out.println("<h2>No cookies founds</h2>");

}

out.println("</body>");

out.println("</html>");

}

}

**The HttpSession Object**

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.

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.

You would get HttpSession object by calling the public method **getSession()** of HttpServletRequest, as below −

HttpSession session = request.getSession();

You need to call *request.getSession()* before you send any document content to the client. Here is a summary of the important methods available through HttpSession object −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public Object getAttribute(String name)**  This method returns the object bound with the specified name in this session, or null if no object is bound under the name. |
| 2 | **public Enumeration getAttributeNames()**  This method returns an Enumeration of String objects containing the names of all the objects bound to this session. |
| 3 | **public long getCreationTime()**  This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. |
| 4 | **public String getId()**  This method returns a string containing the unique identifier assigned to this session. |
| 5 | **public long getLastAccessedTime()**  This method returns the last accessed time of the session, in the format of milliseconds since midnight January 1, 1970 GMT |
| 6 | **public int getMaxInactiveInterval()**  This method returns the maximum time interval (seconds), that the servlet container will keep the session open between client accesses. |
| 7 | **public void invalidate()**  This method invalidates this session and unbinds any objects bound to it. |
| 8 | **public boolean isNew(**  This method returns true if the client does not yet know about the session or if the client chooses not to join the session. |
| 9 | **public void removeAttribute(String name)**  This method removes the object bound with the specified name from this session. |
| 10 | **public void setAttribute(String name, Object value)**  This method binds an object to this session, using the name specified. |
| 11 | **public void setMaxInactiveInterval(int interval)**  This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session. |

# Servlets - File Uploading

A Servlet can be used with an HTML form tag to allow users to upload files to the server. An uploaded file could be a text file or image file or any document.

## Creating a File Upload Form

The following HTM code below creates an uploader form. Following are the important points to be noted down −

* The form **method** attribute should be set to **POST** method and GET method can not be used
* The form **enctype** attribute should be set to **multipart/form-data**.
* The form **action** attribute should be set to a servlet file which would handle file uploading at backend server. Following example is using **UploadServlet** servlet to upload file.
* To upload a single file you should use a single <input .../> tag with attribute type="file". To allow multiple files uploading, include more than one input tags with different values for the name attribute. The browser associates a Browse button with each of them.

<html>

<head>

<title>File Uploading Form</title>

</head>

<body>

<h3>File Upload:</h3>

Select a file to upload: <br />

<form action = "UploadServlet" method = "post" enctype = "multipart/form-data">

<input type = "file" name = "file" size = "50" />

<br />

<input type = "submit" value = "Upload File" />

</form>

</body>

</html>

This will display following result which would allow to select a file from local PC and when user would click at "Upload File", form would be submitted along with the selected fil −

**File Upload:**

Select a file to upload:

NOTE: This is just dummy form and would not work.

## Getting Current Date & Time

This is very easy to get current date and time in Java Servlet. You can use a simple Date object with *toString()* method to print current date and time as follows −

// Import required java libraries

import java.io.\*;

import java.util.Date;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Extend HttpServlet class

public class CurrentDate extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "Display Current Date & Time";

Date date = new Date();

String docType = "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

out.println(docType +

"<html>\n" +

"<head><title>" + title + "</title></head>\n" +

"<body bgcolor = \"#f0f0f0\">\n" +

"<h1 align = \"center\">" + title + "</h1>\n" +

"<h2 align = \"center\">" + date.toString() + "</h2>\n" +

"</body>

</html>"

);

}

}