Chapter 06
Servlets

Contents:

- Background,
- The Life Cycle of a Servlet,
- The Java Servlet Development Kit, The Simple Servlet,
- The Servlet API
- The javax.Servlet Package, Reading Servlet Parameters, Reading Initialization Parameters
- The javax.servlet.http package, Handling HTTP Requests and responses
- Using Cookies, Session Tracking, Security Issues, Exploring Servlet

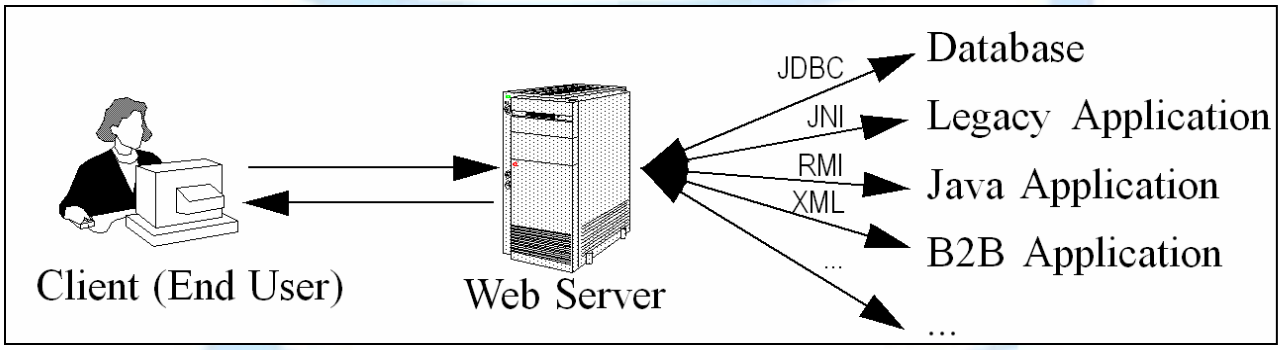
Introduction[Ref.1]

Servlets are small programs that execute on the server side of a Web connection. Just as applets dynamically extend the functionality of a Web browser, servlets dynamically extend the functionality of a Web server.

Servlets are generic extensions to Java-enabled servers. They are secure, portable, and easy to use replacement for CGI. Servlet is a dynamically loaded module that services requests from a Web server and executed within the Java Virtual Machine. Because the servlet is running on the server side, it does not depend on browser compatibility.

Servlet's Job

- Read explicit data sent by client (form data)
- Read implicit data sent by client (request headers)
- Generate the results
- Send the explicit data back to client (HTML)
- Send the implicit data to client (status codes and response headers)



The Hypertext Transfer Protocol (HTTP) [Ref.1]

HTTP is the protocol that allows web servers and browsers to exchange data over the web. It is a request and response protocol. The client requests a file and the server responds to the request. HTTP uses reliable TCP connections—by default on TCP port 80. HTTP (currently at version 1.1 at the time of this writing) was first defined in RFC 2068. It was then refined in RFC 2616, which can be found at http://www.w3c.org/Protocols/.

In HTTP, it's always the client who initiates a transaction by establishing a connection and sending an HTTP request. The server is in no position to contact a client or make a callback connection to the client. Either the client or the server can prematurely terminate a connection. For example, when using a web browser we can click the Stop button on our browser to stop the download process of a file, effectively closing the HTTP connection with the web server.

HTTP Requests [Ref.1]

An HTTP transaction begins with a request from the client browser and ends with a response from the server. An HTTP request consists of three components:

- Method——URI—Protocol/Version
- Request headers
- Entity body

An example of an HTTP request is the following:

```
GET /servlet/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/ch8/SendDetails.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
```

LastName=Franks&FirstName=Michael
The method—URI—protocol version appears as the first line of the request.

GET /servlet/default.jsp HTTP/1.1
where GET is the request method, /servlet/default.jsp represents the URI and HTTP/1.1 the Protocol/Version section.

The request method will be explained in more details in the next section, "HTTP request Methods."

The URI specifies an Internet resource completely. A URI is usually interpreted as being relative to the server's root directory. Thus, it should always begin with a forward slash /. A URL is actually a type of URI (see http://www.ietf.org/rfc/rfc2396.txt). The Protocol version represents the version of the HTTP protocol being used.

The request header contains useful information about the client environment and the entity body of the request. For example, it could contain the language the browser is set for, the length of the entity body, and so on. Each header is separated by a carriage return/linefeed (CRLF) sequence.

Between the headers and the entity body, there is a blank line (CRLF) that is important to the HTTP request format. The CRLF tells the HTTP server where the entity body begins. In some Internet programming books, this CRLF is considered the fourth component of an HTTP request.

In the previous HTTP request, the entity body is simply the following line:

LastName=Franks&FirstName=Michael

The entity body could easily become much longer in a typical HTTP request.

Method	Description
GET	GET is the simplest, and probably, most used HTTP method. GET simply retrieves the data identified by the URL. If the URL refers to a script (CGI, servlet, and so on), it returns the data produced by the script.
HEAD	The HEAD method provides the same functionality as GET, but HEAD only returns HTTP headers without the document body.
POST	Like GET, POST is also widely used. Typically, POST is used in HTML forms. POST is used to transfer a block of data to the server in the entity body of the request.
OPTIONS	The OPTIONS method is used to query a server about the capabilities it provides. Queries can be general or specific to a particular resource.
PUT	The PUT method is a complement of a GET request, and PUT stores the entity body at the location specified by the URI. It is similar to the PUT function in FTP.
DELETE	The DELETE method is used to delete a document from the server. The document to be deleted is indicated in the URI section of the request.
TRACE	The TRACE method is used to tract the path of a request through firewall and multiple proxy servers. TRACE is useful for debugging complex network problems and is similar to the traceroute tool.

Of the seven methods, only GET and POST are commonly used in an Internet application.

HTTP Responses [Ref.1]

Similar to requests, an HTTP response also consists of three parts:

- Protocol—Status code—Description
- Response headers
- Entity body

The following is an example of an HTTP response:

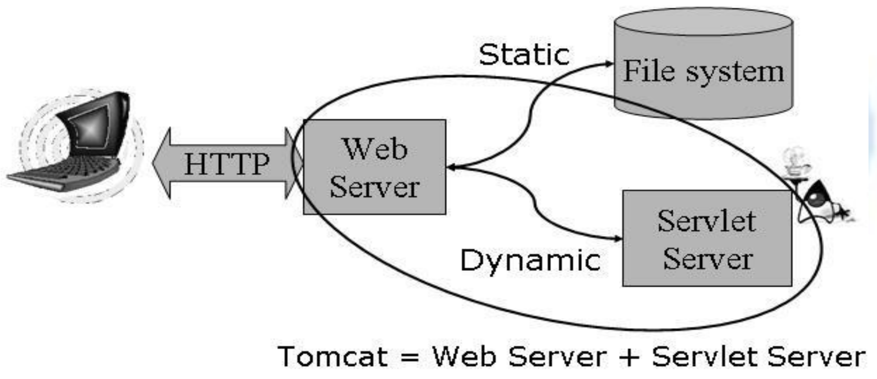
```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 3 Jan 1998 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 11 Jan 1998 13:23:42 GMT
Content-Length: 112
```

```
<HTML>
<HEAD>
<TITLE>HTTP Response Example</TITLE></HEAD><BODY>
Welcome to Brainy Software
</BODY>
</HTML>
```

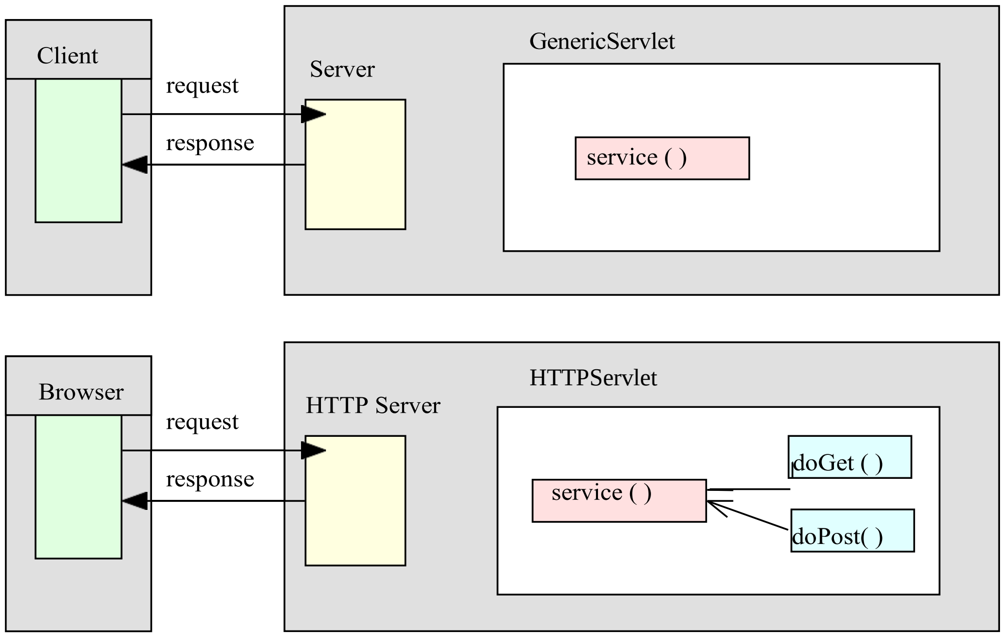
The first line of the response header is similar to the first line of the request header. The first line tells you that the protocol used is HTTP version 1.1, the request succeeded (200 = success), and that everything went okay.

The response headers contain useful information similar to the headers in the request. The entity body of the response is the HTML content of the response itself. The headers and the entity body are separated by a sequence of CRLFs.

Where are servlets?



Servlet Application Architecture



Applications of Java Servlets

- Building e-commerce store fronts
  - Servlet builds an online catalog based on the contents of a database
  - Customer places an order, which is processed by another servlet
- Servlets as wrappers for legacy systems
- Servlets interacting with EJB applications

Java Servlet alternatives [Ref.1]

ColdFusion. Allaire's ColdFusion provides HTML-like custom tags that can be used to perform a number of operations, especially querying a database. This technology had its glamorous time in the history of the World Wide Web as the main technology for web application programming. Its glorious time has since gone with the invention of other technologies.

Server-side JavaScript (SSJS). SSJS is an extension of the JavaScript language, the scripting language that still rules client-side web programming. SSJS can access Java classes deployed at the server side using the LiveWire technology from Netscape.

PHP. PHP is an exciting open-source technology that has matured in recent years. The technology provides easy web application development with its session management and includes some built-in functionality, such as file upload. The number of programmers embracing PHP as their technology of choice has risen sharply in recent years.

Servlet. The servlet technology was introduced by Sun Microsystems in 1996.

JavaServer Pages (JSP). JSP is an extension of the servlet technology.

Active Server Pages (ASP). Microsoft's ASP employs scripting technologies that work in Windows platforms, even though there have been efforts to port this technology to other operating systems. Windows ASP works with the Internet Information Server web server. This technology will soon be replaced by Active Server Pages.NET.

Active Server Pages.NET (ASP.NET). This technology is part of Microsoft's .NET initiative. Interestingly, the .NET Framework employs a runtime called the Common Language Runtime that is very similar to Java Virtual Machine and provides a vast class library available to all .NET languages and from ASP.NET pages. ASP.NET is an exciting technology. It introduced several new technologies including state management that does not depend on cookies or URL rewriting.

The Benefits of Servlets

- *Efficiency*: More efficient – uses lightweight java threads as opposed to individual processes.
- *Persistency*: Servlets remain in memory. They can maintain state between requests.
- *Portability*: Since servlets are written in Java, they are platform independent.
- *Robustness*: Error handling, Garbage collector to prevent problems with memory leaks. Large class library – network, file, database, distributed object components, security, etc.
- *Extensibility*: Creating new subclasses that suite your needs Inheritance, polymorphism, etc.
- *Security*: Security provided by the server as well as the Java Security Manager. It eliminates problems associated with executing cgi scripts using operating system "shells".
- *Powerful*: Servlets can directly talk to web server and facilitates database connection pooling, session tracking etc.
- *Convenient*: Parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, etc.
- *Rapid development cycle:* As a Java technology, servlets have access to the rich Java library, which helps speed up the development process.
- *Widespread acceptance:* Java is a widely accepted technology. This means that numerous vendors work on Java-based technologies. One of the advantages of this widespread acceptance is that we can easily find and purchase components that suit our needs, which saves precious development time.

How a Servlet Works

A servlet is loaded by the servlet container the first time the servlet is requested. The servlet then is forwarded the user request, processes it, and returns the response to the servlet container, which in turn sends the response back to the user. After that, the servlet stays in memory waiting for other requests—it will not be unloaded from the memory unless the servlet container sees a shortage of memory. Each time the servlet is requested, however, the servlet container compares the timestamp of the loaded servlet with the servlet class file. If the class file timestamp is more recent, the servlet is reloaded into memory. This way, we don't need to restart the servlet container every time we update our servlet.

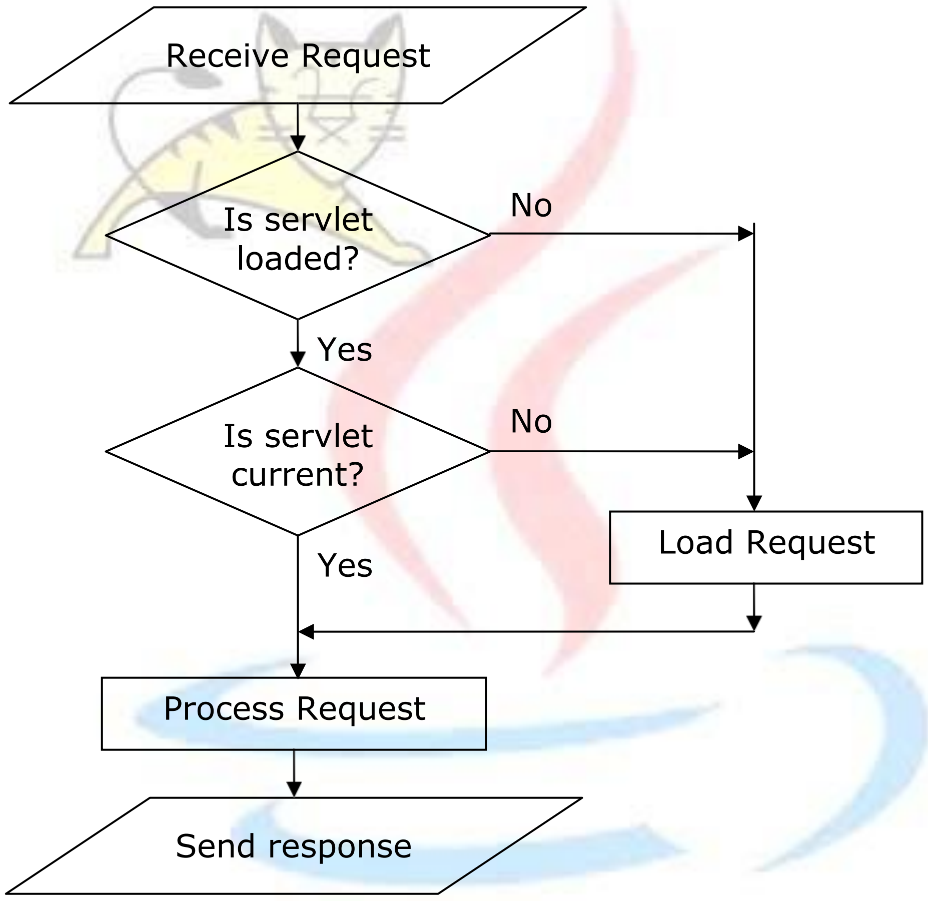


Fig. How servlet works?

The Tomcat Servlet Container [Ref.1]

A number of servlet containers are available today these are listed below:

- Apache Tomcat_
  http://jakarta.apache.org/tomcat/
- Allaire/Macromedia JRun_
  http://www.macromedia.com/software/jrun/
- New Atlanta ServletExec_
  http://www.servletexec.com/
- Gefion Software LiteWebServer_
  http://www.gefionsoftware.com/LiteWebServer/
- Caucho's Resin_
  http://www.caucho.com/

The most popular one—and the one recognized as the official servlet/JSP container—is Apache Tomcat. Originally designed by Sun Microsystems, Tomcat source code was handed over to the Apache Software Foundation in October 1999. In this new home, Tomcat was included as part of the Jakarta Project, one of the projects of the Apache Software Foundation. Working through the Apache process, Apache, Sun, and other companies—with the help of volunteer programmers worldwide—turned Tomcat into a world-class servlet reference implementation. Currently we are using Apache Tomcat version 6.0.18.

Tomcat by itself is a web server. This means that you can use Tomcat to service HTTP requests for servlets, as well as static files (HTML, image files, and so on). In practice, however, since it is faster for non-servlet, non-JSP requests, Tomcat normally is used as a module with another more robust web server, such as Apache web server or Microsoft Internet Information Server. Only requests for servlets or JSP pages are passed to Tomcat.

For writing a servlet, we need at Java Development Kit installed on our computer. Tomcat is written purely in Java.

Steps to Running Your First Servlet [Ref.1]

After we have installed and configured Tomcat, we can put it into service. Basically, we need to follow steps to go from writing our servlet to running it. These steps are summarized as follows:

- Write the servlet source code. We need to import the javax.servlet package and the javax.servlet.http package in your source file.
- Compile your source code.
- Create a deployment descriptor.
- Run Tomcat.
- Call your servlet from a web browser.

1. Write the Servlet Source Code

In this step, we prepare our source code. We can write the source code ourself using any text editor.

The following program shows a simple servlet called TestingServlet. The file is named TestingServlet.java. The servlet sends a few HTML tags and some text to the browser.

```
//TestingServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class TestingServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
    {
```

```
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Servlet Testing</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Welcome to the Servlet Testing Center");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

Now, save TestingServlet.java file to /bin directory of JDK.

2. Compiling the source code

For our servlet source code to compile, we need to include the path to the servlet-api.jar file in our CLASSPATH environment variable. The servlet-api.jar is located in the C:\Program Files\Apache Software Foundation\Tomcat 6.0\ directory. Here, the drive name depends upon our selection while installation of Tomcat 6.0 on computer. So, compile the file using following way:

```
javac TestingServlet.java -classpath "G:\Program Files\Apache
Software Foundation\Tomcat 6.0\lib\servlet-api.jar"
```

After successful compilation, we will get a class file named TestingServlet.class. Now, copy that class file into directory \classes under web-inf as shown in the following figure. All the servlet classes resides in this directory.

3. Create the Deployment Descriptor

A deployment descriptor is an optional component in a servlet application. The descriptor takes the form of an XML document called *web.xml* and must be located in the WEB-INF directory of the servlet application. When present, the deployment descriptor contains configuration settings specific to that application. In order to create the deployment descriptor, we now need to create or edit a web.xml file and place it under the WEB-INF directory. The *web.xml* for this example application must have the following content.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
     <servlet-name>Testing</servlet-name>
     <servlet-class>TestingServlet</servlet-class>
     <servlet-mapping>
          <servlet-name>TestingServlet</servlet-name>
          <url-pattern>/servlets/servlet/TestingServlet
               </url-pattern>
     </servlet-mapping>
  </servlet>
</web-app>
```

The web.xml file has one element—web-app. We should write all our servlets under <web-app>. For each servlet, we have a <servlet> element and we need the <servlet-name> and <servlet-class> elements. The <servlet-name> is the name for our servlet, by which it is known Tomcat. The <servlet-class> is the compiled file of your servlet without the .class extension.

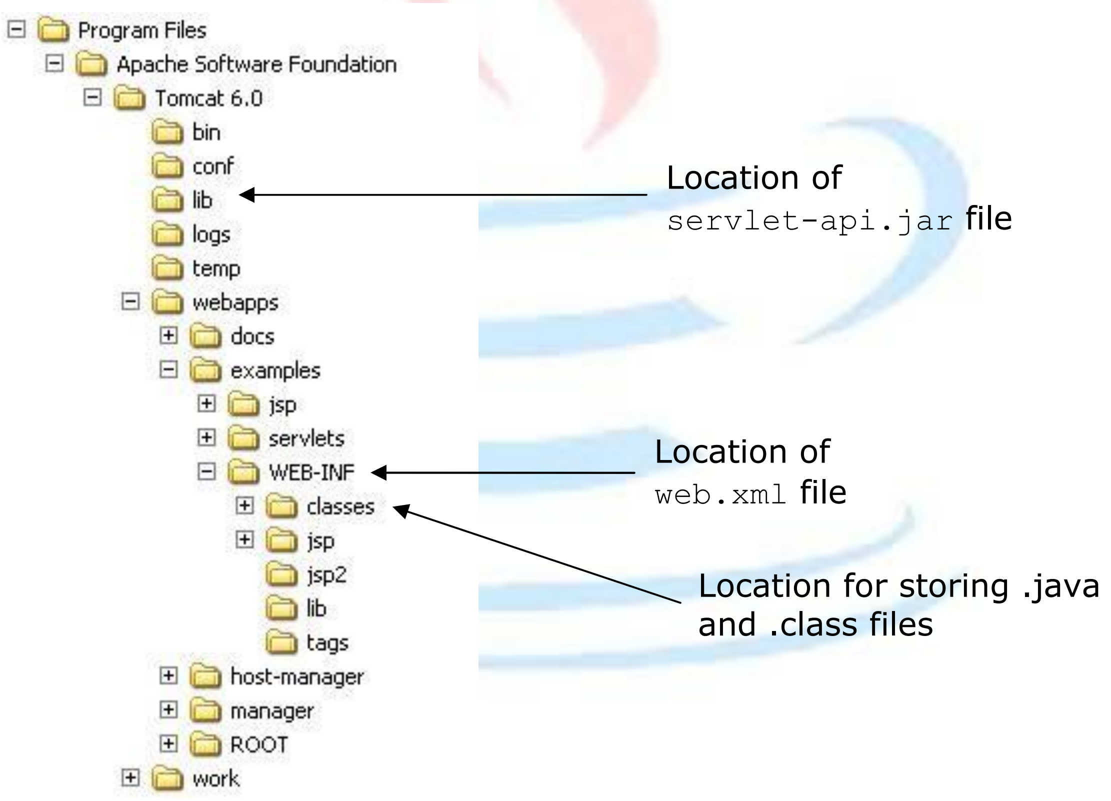


Fig. Apache Tomcat 6.0 Directory Structure.

We can also add multiple servlet names in our file using multiple servlet tags. The url-pattern suggests the url by which we are going to class our servlet in the web browser. Instead of doing this we can just modify the contents of web.xml by adding the names and mapping of our servlet code.

4. Run Tomcat

If Tomcat is not already running, we need to start it by selecting the option “monitor tomcat” from srart menu. We will find the icon of Apache Tomcat on the taskbar when it is running.

5. Call Your Servlet from a Web Browser

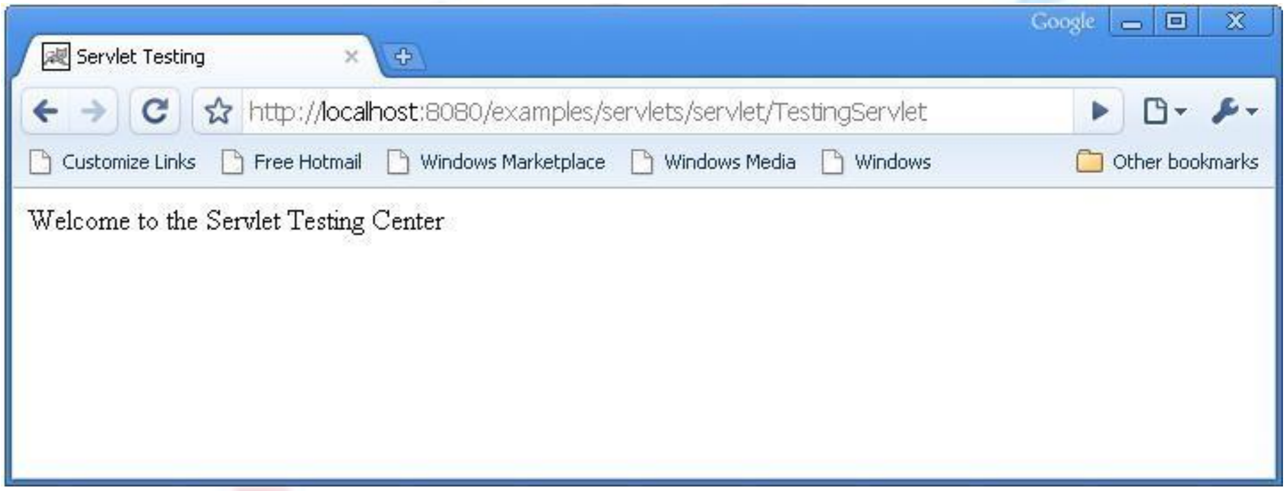
Now, we can call our servlet from a web browser. By default, Tomcat runs on port 8080. The URL for that servlet has the following format:

http://domain-name/virtual-directory/servlet/servlet-name

If we run the web browser from the same computer as Tomcat, you can replace the domain-name part with "localhost". In that case, the URL for your servlet is:

http://localhost:8080/examples/servlets/servlet/TestingServlet

Typing the URL in the Address or Location box of our web browser will give you the string "Welcome to the Servlet Testing Center," as shown in Figure 1.5.



Welcome to the Servlet Testing Center

The javax.servlet package [Ref.1]

The javax.servlet package contains seven interfaces, three classes, and two exceptions. The seven interfaces are as follows:

- `RequestDispatcher`
- `Servlet`
- `ServletConfig`
- `ServletContext`
- `ServletRequest`

- `ServletResponse`
- `SingleThreadModel`

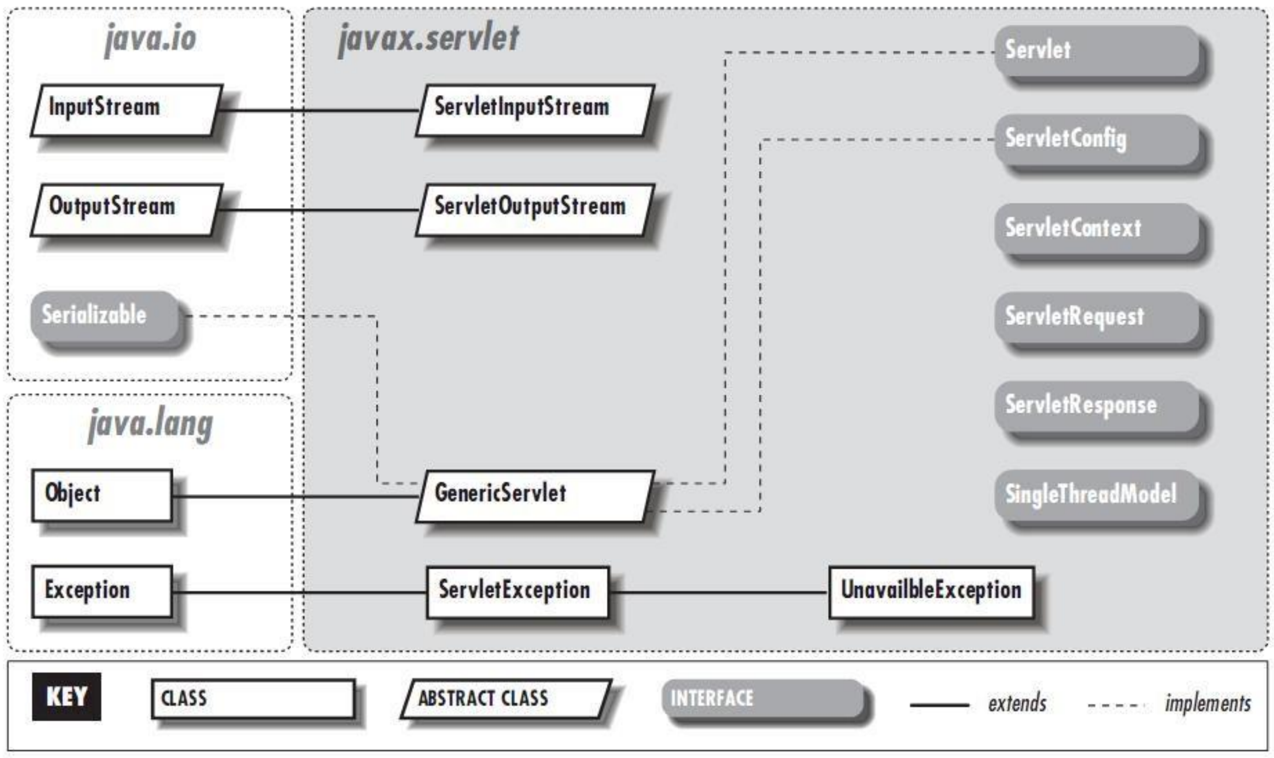
The three classes are as follows:

- `GenericServlet`
- `ServletInputStream`
- `ServletOutputStream`

And, finally, the exception classes are these:

- `ServletException`
- `UnavailableException`

The object model of the javax.servlet package is shown in figure below:



The javax.servlet package

The Servlet's Life Cycle

Applet life cycle contains methods: init( ), start( ), paint( ), stop( ), and destroy( ) – appropriate methods called based on user action. Similarly, servlets operate in the context of a request and response model managed by a servlet engine The engine does the following:

- Loads the servlet when it is first requested.
- Calls the servlet's init( ) method.
- Handles any number of requests by calling the servlet's service( ) method.
- When shutting down, calls each servlet's destroy( ) method.

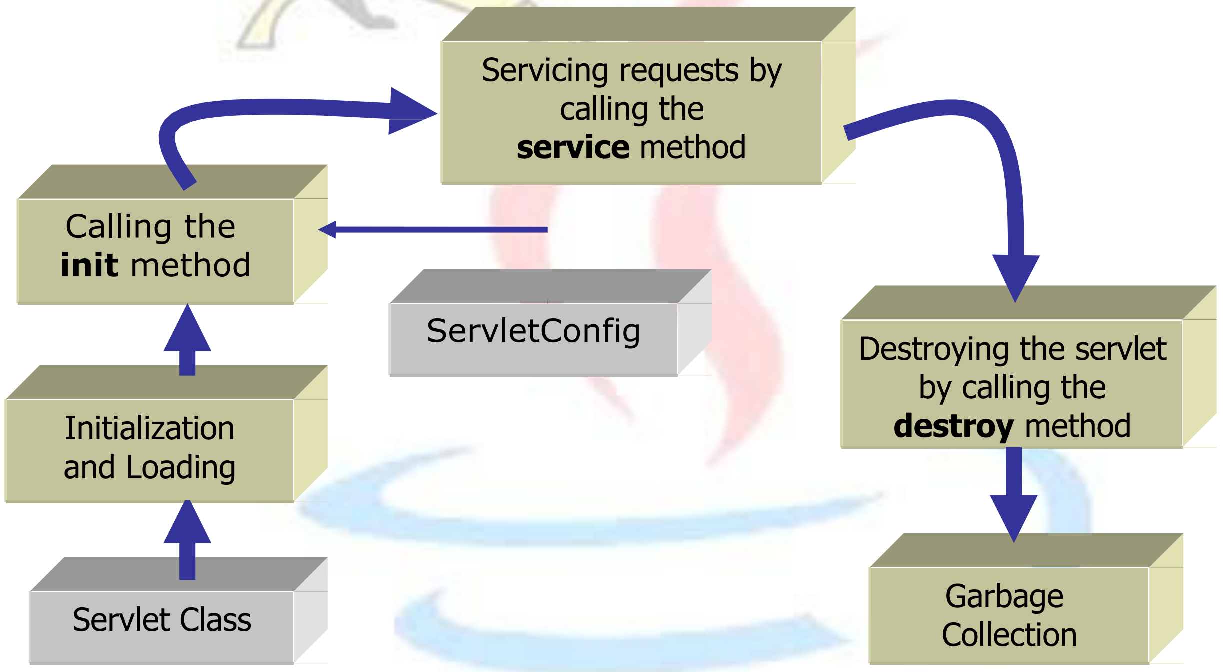

Servicing requests by calling the **service** method
Calling the **init** method
ServletConfig
Destroying the servlet by calling the **destroy** method
Initialization and Loading
Servlet Class
Garbage Collection

Fig. Servlet's Life Cycle

The init( ) method

- It request for a servlet received by the servlet engine.
- Checks to see if the servlet is already loaded.
- If not, uses a class loader to get the required servlet class and instantiates it by calling the constructor method.
- After the servlet is loaded, but before it services any requests, the init ( ) method is called.
- Inside init( ), the resources used by the servlet are initialized. E.g: establishing database connection.
- This method is called only once just before the servlet is placed into service.
- The init( ) method takes a ServletConfig object as a parameter. Its signature is:
  `public void init(ServletConfig config) throws ServletException`
- Most common way of doing this is to have it call the super.init( ) passing it the ServletConfig object.

The service( ) method

- The service( ) method handles all requests sent by a client.
- It cannot start servicing requests until the init( ) method has been executed.
- Only a single instance of the servlet is created and the servlet engine dispatches each request in a single thread.
- The service( ) method is used only when extending GenericServlet class.
- Since servlets are designed to operate in the HTTP environment, the HttpServlet class is extended.
- The service(HttpServletRequest, HttpServletResponse) method examines the request and calls the appropriate doGet() or doPost() method.
- A typical Http servlet includes overrides to one or more of these subsidiary methods rather than an override to service().

The destroy( ) method

- This method signifies the end of a servlet's life.
- The resources allocated during init( ) are released.
- Save persistent information that will be used the next time the servlet is loaded.
- The servlet engine unloads the servlet.
- Calling destroy( ) yourself will not acutally unload the servlet. Only the servlet engine can do this.

Demonstrating the Life Cycle of a Servlet [Ref.1]

The following program contains the code for a servlet named PrimitiveServlet, a very simple servlet that exists to demonstrate the life cycle of a servlet. The PrimitiveServlet class implements javax.servlet.Servlet (as all servlets must) and provides implementations for all the five methods of servlet. What it does is very simple. Each time any of the init, service, or destroy methods is called, the servlet writes the method's name to the console.

```
import javax.servlet.*;
import java.io.IOException;
public class PrimitiveServlet implements Servlet
{
  public void init(ServletConfig config) throws ServletException {
    System.out.println("init");
  }
  public void service(ServletRequest request,
                         ServletResponse response)
    throws ServletException, IOException {
    System.out.println("service");
  }
  public void destroy() {
    System.out.println("destroy");
  }
  public String getServletInfo() {
    return null;
  }
  public ServletConfig getServletConfig() {
    return null;
  }
}
```

The web.xml File for PrimitiveServlet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>PrimitiveServlet</servlet-name>
    <servlet-class>PrimitiveServlet</servlet-class>
  </servlet>
</web-app>
```

We should then be able to call this servlet from our browser by typing the following URL:

http://localhost:8080/examples/servlets/servlet/PrimitiveServlet

The first time the servlet is called, the console displays these two lines:

```
init
service
```

This tells us that the init method is called, followed by the service method. However, on subsequent requests, only the service method is called. The servlet adds the following line to the console:

```
service
```

This proves that the init method is called only once.

Requests and Responses [Ref.1]

Requests and responses are what a web application is all about. In a servlet application, a user using a web browser sends a request to the servlet container, and the servlet container passes the request to the servlet.

In a servlet paradigm, the user request is represented by the ServletRequest object passed by the servlet container as the first argument to the service method. The service method's second argument is a ServletResponse object, which represents the response to the user.

The ServletRequest Interface [Ref.1]

The ServletRequest interface defines an object used to encapsulate information about the user's request, including parameter name/value pairs, attributes, and an input stream.

The ServletRequest interface provides important methods that enable us to access information about the user. For example, the getParameterNames method returns an Enumeration containing the parameter names for the current request. In order to get the value of each parameter, the ServletRequest interface provides the getParameter method.

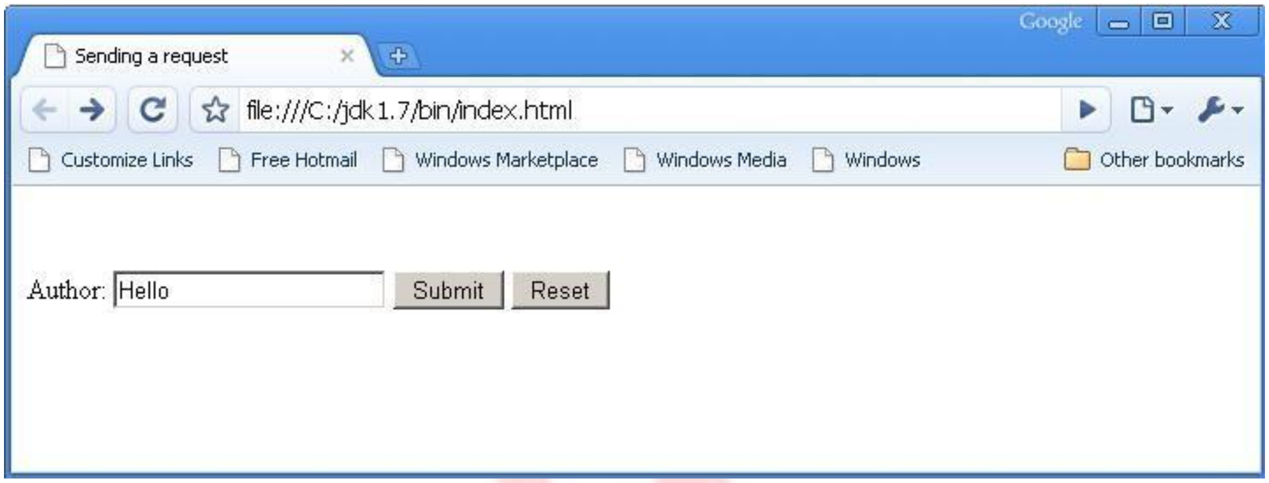
The getRemoteAddress and getRemoteHost methods are two methods that we can use to retrieve the user's computer identity. The first returns a string representing the IP address of the computer the client is using, and the second method returns a string representing the qualified host name of the computer.

The following example, shows a ServletRequest object in action. The example consists of an HTML form in a file named index.html and a servlet called RequestDemoServlet.

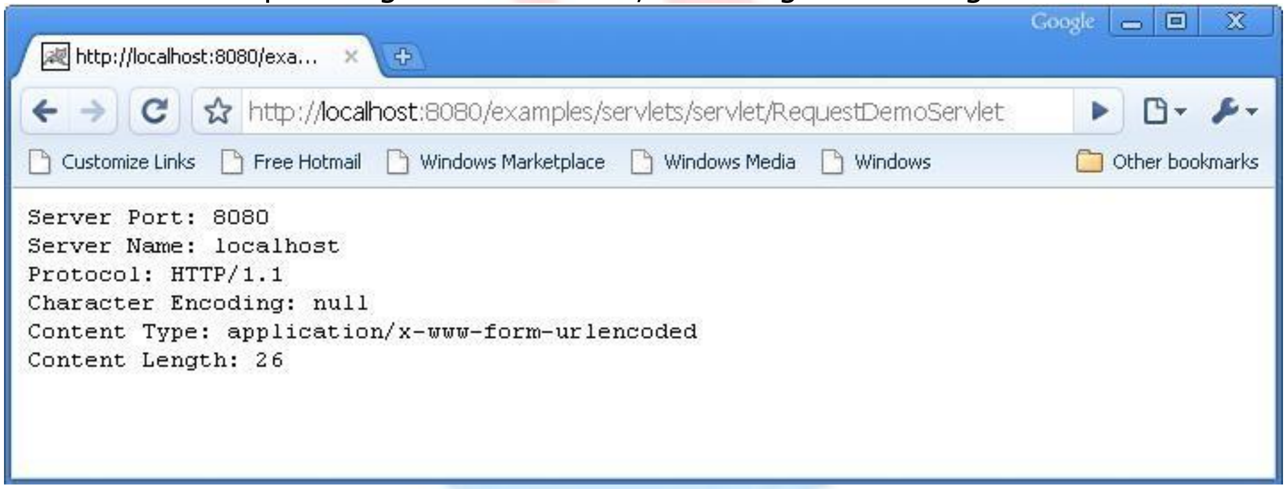
The index.html file:
```
<HTML>
<HEAD>
<TITLE>Sending a request</TITLE>
</HEAD>
<BODY>
<FORM ACTION =
http://localhost:8080/examples/servlets/servlet/RequestDemoServlet
METHOD="POST">
<BR><BR>
Author: <INPUT TYPE="TEXT" NAME="Author">
<INPUT TYPE="SUBMIT" NAME="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

The RequestDemoServlet.java file:
```
import javax.servlet.*;
import java.util.Enumeration;
import java.io.IOException;
public class RequestDemoServlet implements Servlet {
  public void init(ServletConfig config) throws ServletException {
  }
  public void destroy() {
  }
  public void service(ServletRequest request,
                            ServletResponse response)
    throws ServletException, IOException {
    System.out.println("Server Port: " + request.getServerPort());
    System.out.println("Server Name: " + request.getServerName());
    System.out.println("Protocol: " + request.getProtocol());
    System.out.println("Character Encoding: " +
      request.getCharacterEncoding());
    System.out.println("Content Type: " +
          request.getContentType());
    System.out.println("Content Length: " +
          request.getContentLength());
    System.out.println("Remote Address: " +
          request.getRemoteAddr());
    System.out.println("Remote Host: " + request.getRemoteHost());
    System.out.println("Scheme: " + request.getScheme());
    Enumeration parameters = request.getParameterNames();
    while (parameters.hasMoreElements()) {
      String parameterName = (String) parameters.nextElement();
      System.out.println("Parameter Name: " + parameterName);
      System.out.println("Parameter Value: " +
        request.getParameter(parameterName));
    }
    Enumeration attributes = request.getAttributeNames();
    while (attributes.hasMoreElements()) {
      String attribute = (String) attributes.nextElement();
      System.out.println("Attribute name: " + attribute);
      System.out.println("Attribute value: " +
        request.getAttribute(attribute));
    }
  }
  public String getServletInfo() {
    return null;
  }
  public ServletConfig getServletConfig() {
    return null;
  }
}
```

The snapshot of index.html:



After pressing submit button, we will get following window:


http://localhost:8080/exa...
http://localhost:8080/examples/servlets/servlet/RequestDemoServlet
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Other bookmarks

```
Server Port: 8080
Server Name: localhost
Protocol: HTTP/1.1
Character Encoding: null
Content Type: application/x-www-form-urlencoded
Content Length: 26
```

The ServletResponse Interface [Ref.1]

The ServletResponse interface represents the response to the user. The most important method of this interface is getWriter, from which we can obtain a java.io.PrintWriter object that we can use to write HTML tags and other text to the user.

The codes of the program given below offer an HTML file named index2.html and a servlet whose service method is overridden with code that outputs some HTML tags to the user. This servlet modifies the example below retrieves various information about the user. Instead of sending the information to the console, the service method sends it back to the user.

index2.html

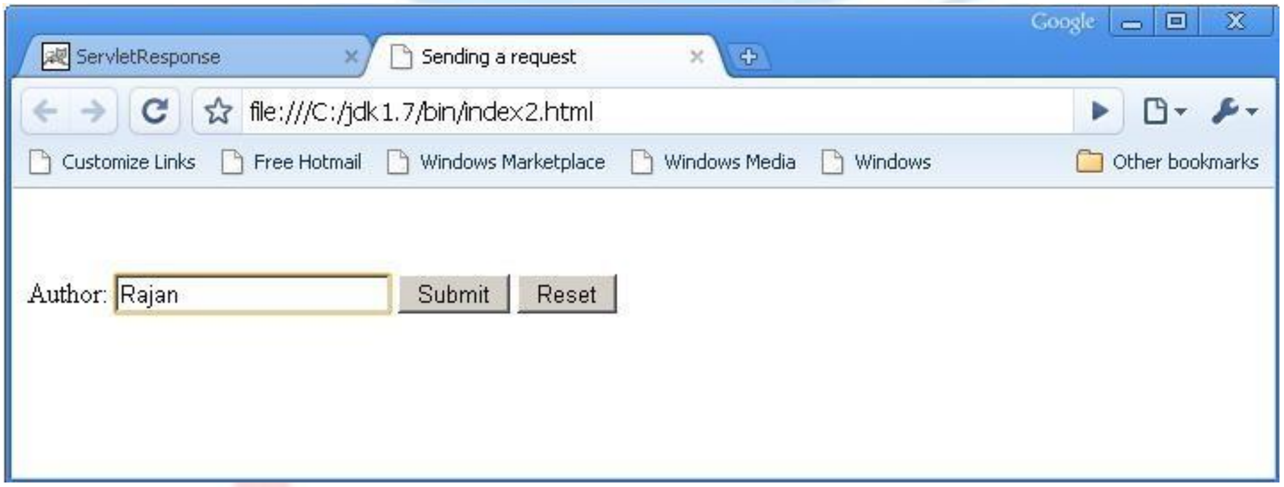
```
<HTML>
<HEAD>
<TITLE>Sending a request</TITLE>
</HEAD>
<BODY>
```

```
<FORM ACTION=
http://localhost:8080/examples/servlets/servlet/RequestDemoServlet
METHOD="POST">
<BR><BR>
Author: <INPUT TYPE="TEXT" NAME="Author">
<INPUT TYPE="SUBMIT" NAME="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

ResponseDemoServlet.java

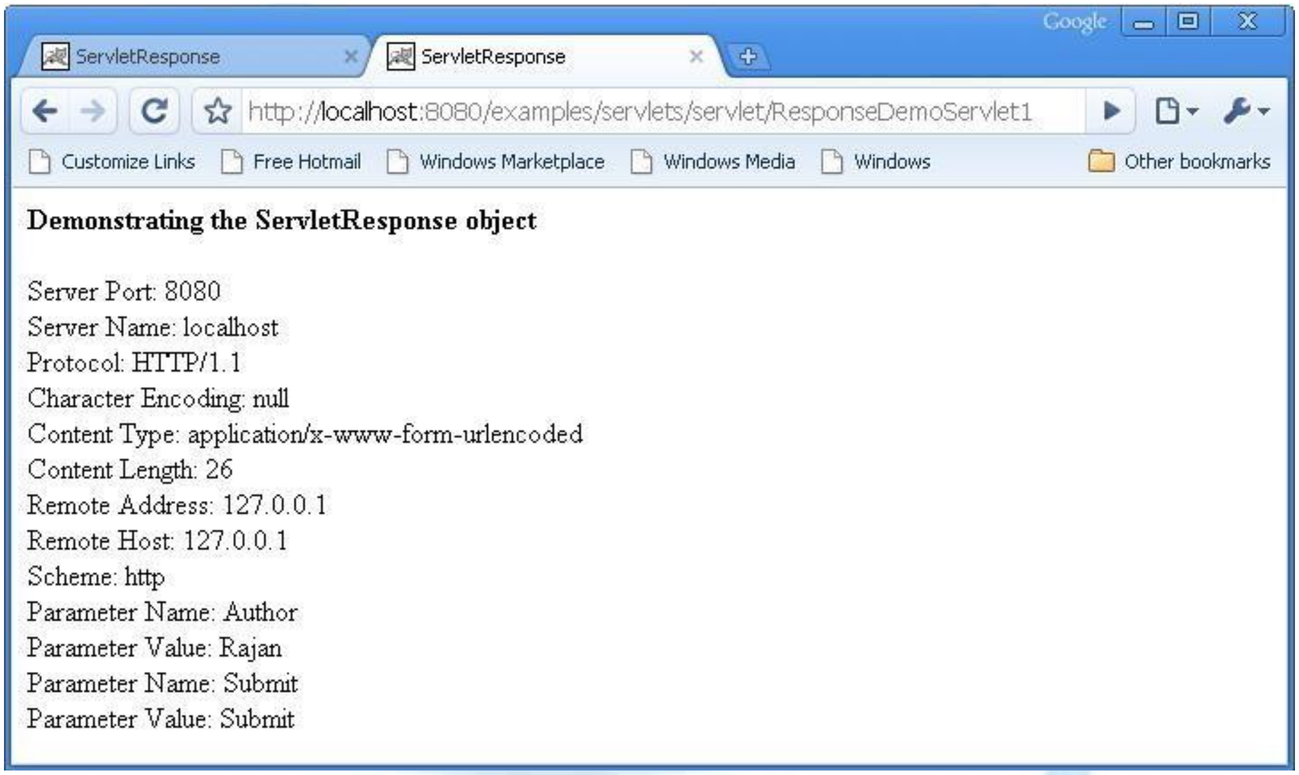
```
import javax.servlet.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Enumeration;

public class ResponseDemoServlet implements Servlet {
  public void init(ServletConfig config) throws ServletException {
  }
  public void destroy() {
  }
  public void service(ServletRequest request,
                            ServletResponse response)
    throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>");
    out.println("ServletResponse");
    out.println("</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<B>Demonstrating the ServletResponse object</B>");
    out.println("<BR>");
    out.println("<BR>Server Port: " + request.getServerPort());
    out.println("<BR>Server Name: " + request.getServerName());
    out.println("<BR>Protocol: " + request.getProtocol());
    out.println("<BR>Character Encoding: " +
                    request.getCharacterEncoding());
    out.println("<BR>Content Type: " + request.getContentType());
    out.println("<BR>Content Length: " +
                    request.getContentLength());
    out.println("<BR>Remote Address: " + request.getRemoteAddr());
    out.println("<BR>Remote Host: " + request.getRemoteHost());
    out.println("<BR>Scheme: " + request.getScheme());
    Enumeration parameters = request.getParameterNames();
    while (parameters.hasMoreElements()) {
      String parameterName = (String) parameters.nextElement();
      out.println("<br>Parameter Name: " + parameterName);
      out.println("<br>Parameter Value: " +
        request.getParameter(parameterName));
    }
    Enumeration attributes = request.getAttributeNames();
    while (attributes.hasMoreElements()) {
      String attribute = (String) attributes.nextElement();
      out.println("<BR>Attribute name: " + attribute);
      out.println("<BR>Attribute value: " +
                    request.getAttribute(attribute));
    }
    out.println("</BODY>");
    out.println("</HTML>");
  }
  public String getServletInfo() {
    return null;
  }
  public ServletConfig getServletConfig() {
    return null;
  }
}
```

Window of index2.html



Author: Rajan

Window after clicking the 'submit' button



```
Demonstrating the ServletResponse object

Server Port: 8080
Server Name: localhost
Protocol: HTTP/1.1
Character Encoding: null
Content Type: application/x-www-form-urlencoded
Content Length: 26
Remote Address: 127.0.0.1
Remote Host: 127.0.0.1
Scheme: http
Parameter Name: Author
Parameter Value: Rajan
Parameter Name: Submit
Parameter Value: Submit
```

GenericServlet [Ref.1]

Till this point, we have been creating servlet classes that implement the javax.servlet.Servlet interface. Everything works fine, but there are two annoying things that we've probably noticed:

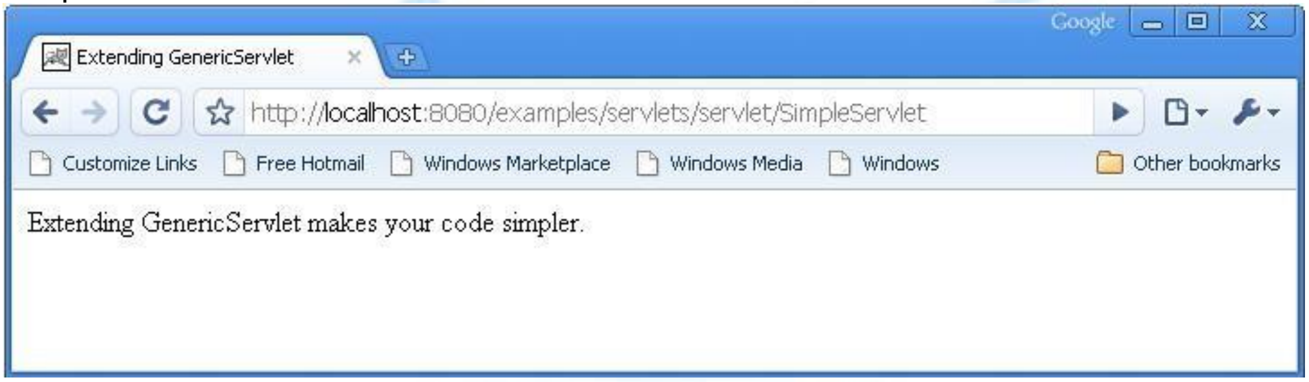
1. We have to provide implementations for all five methods of the Servlet interface, even though most of the time we only need one. This makes your code look unnecessarily complicated.
2. The ServletConfig object is passed to the init method. We need to preserve this object to use it from other methods. This is not difficult, but it means extra work.

The javax.servlet package provides a wrapper class called GenericServlet that implements two important interfaces from the javax.servlet package: Servlet and ServletConfig, as well as the java.io.Serializable interface. The GenericServlet class provides implementations for all methods, most of which are blank. We can extend GenericServlet and override only methods that we need to use. Clearly, this looks like a better solution.

The program given below called SimpleServlet that extends GenericServlet. The code provides the implementation of the service method that sends some output to the browser. Because the service method is the only method we need, only this method needs to appear in the class. Compared to all servlet classes that implement the javax.servlet.Servlet interface directly, SimpleServlet looks much cleaner and clearer.

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;
public class SimpleServlet extends GenericServlet {
  public void service(ServletRequest request,
                      ServletResponse response)
    throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>");
    out.println("Extending GenericServlet");
    out.println("</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Extending GenericServlet makes your code
                           simpler.");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

Output window:



Extending GenericServlet makes your code simpler.

The HttpServlet Class [Ref.1]

The HttpServlet class extends the javax.servlet.GenericServlet class. The HttpServlet class also adds a number of interesting methods to use. The most important are the six doxxx methods that get called when a related HTTP request method is used. The six methods are doPost, doPut, doGet, doDelete, doOptions and doTrace. Each doxxx method is invoked when a corresponding HTTP method is used. For instance, the doGet method is invoked when the servlet receives an HTTP request that was sent using the GET method. Of the six doxxx methods, the doPost and the doGet methods are the most frequently used.

The doPost method is called when the browser sends an HTTP request using the POST method. The POST method is one of the two methods that can be used by an HTML form. Consider the following HTML form at the client side:

```
<FORM ACTION="Register" METHOD="POST">
<INPUT TYPE=TEXT Name="firstName">
<INPUT TYPE=TEXT Name="lastName">
<INPUT TYPE=SUBMIT>
</FORM>
```

When the user clicks the Submit button to submit the form, the browser sends an HTTP request to the server using the POST method. The web server then passes this request to the Register servlet and the doPost method of the servlet is invoked. Using the POST method in a form, the parameter name/value pairs of the form are sent in the request body. For example, if we use the preceding form as an example and enter ‘Sunil’ as the value for firstName and ‘Go’ as the value for lastName, we will get the following result in the request body:

```
firstName=Sunil
lastName=Go
```

An HTML form can also use the GET method; however, POST is much more often used with HTML forms.

The doGet method is invoked when an HTTP request is sent using the GET method. GET is the default method in HTTP. When we type a URL, such as www.yahoo.com, our request is sent to Yahoo! using the GET method. If we use the GET method in a form, the parameter name/value pairs are appended to the URL. Therefore, if we have two parameters named firstName and lastName in our form, and the user enters Sunil and Go, respectively, the URL to our servlet will become something like the following:

```
http://yourdomain/myApp/Register?firstName=Sunil&lastName=Go
```

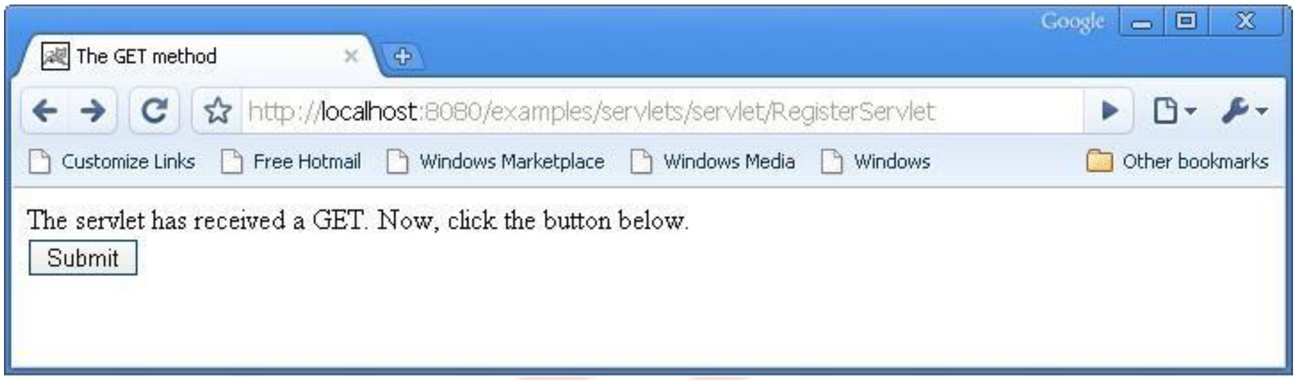
Upon receiving a GET method, the servlet will call its doGet method. The service method of HttpServlet class is as follows:

```
protected void service(HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException
```

//Demonstration of doGet( ) and doPost( ) methods.
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

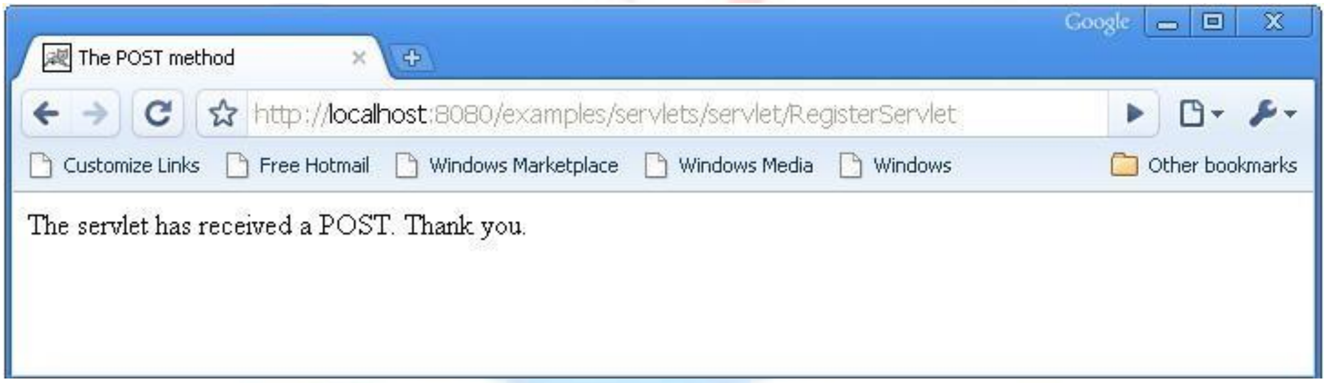
```
public class RegisterServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>The GET method</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("The servlet has received a GET. " +
      "Now, click the button below.");
    out.println("<BR>");
    out.println("<FORM METHOD=POST>");
    out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");

  }
  public void doPost(
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>The POST method</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("The servlet has received a POST. Thank you.");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

When the servlet is first called from a web browser by typing the URL to the servlet in the Address or Location box, GET is used as the request method. At the server side, the doGet method is invoked. The servlet sends a string saying "The servlet has received a GET. Now, click the button below." plus an HTML form. The output is shown in Figure below:



The servlet has received a GET. Now, click the button below.

The form sent to the browser uses the POST method. When the user clicks the button to submit the form, a POST request is sent to the server. The servlet then invokes the doPost method, sending a String saying, "The servlet has received a POST. Thank you," to the browser. The output of doPost is shown in Figure below:



The servlet has received a POST. Thank you.

HttpServletRequest Interface [Ref.1]

In addition to providing several more protocol-specific methods in the HttpServlet class, the javax.servlet.http package also provides more sophisticated request and response interfaces.

Obtaining HTTP Request Headers from HttpServletRequest [Ref.1]

The HTTP request that a client browser sends to the server includes an HTTP request header with important information, such as cookies and the referer. We can access these headers from the HttpServletRequest object passed to a doxxx method.

The following example demonstrates how we can use the HttpServletRequest interface to obtain all the header names and sends the header name/value pairs to the browser.

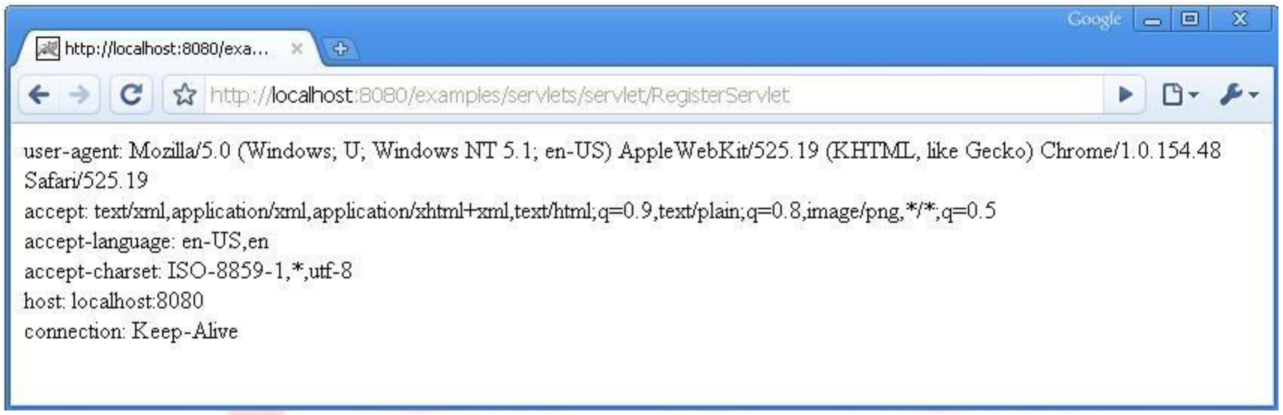
//Obtaining HTTP request Headers

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class RegisterServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
          HttpServletResponse response)
    throws ServletException, IOException
   {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Enumeration enumeration = request.getHeaderNames();
    while(enumeration.hasMoreElements())
    {
      String header = (String) enumeration.nextElement();
      out.println(header + ": " + request.getHeader(header) +
                              "<BR>");
    }
  }
}
```

The RegisterServlet given above uses the getHeaderNames and the getHeader methods. The getHeaderNames is first called to obtain an Enumeration containing all the header names found in the client request. The value of each header then is retrieved by using the getHeader method, passing a header name.

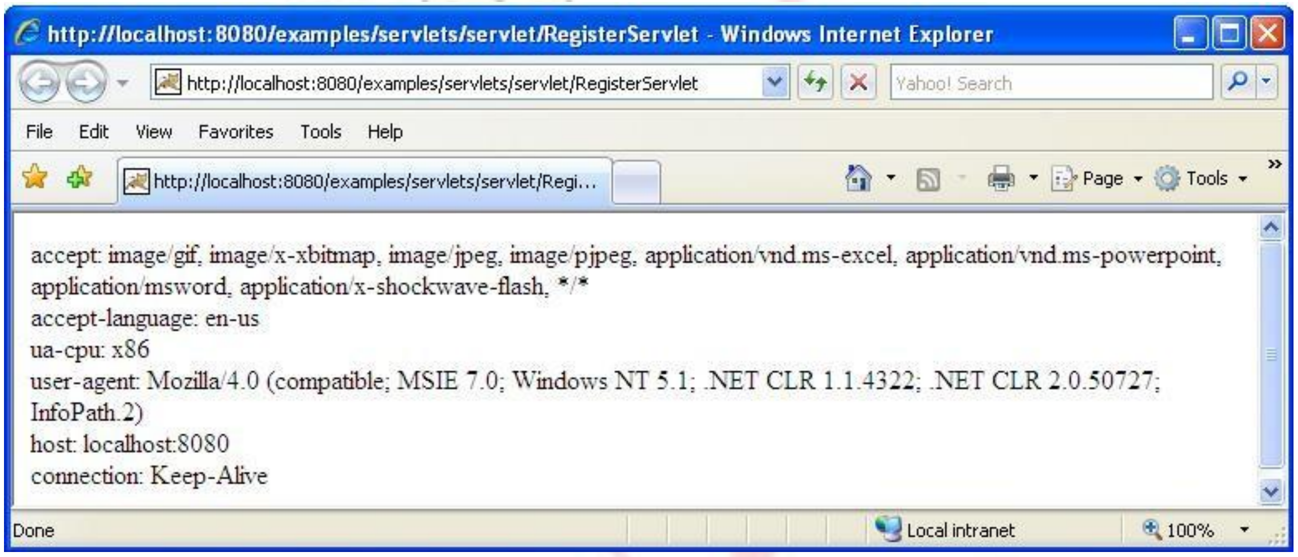
The output of the code depends on the client environment, such as the browser used and the operating system of the client's machine. For example, some browsers might send cookies to the server. Also, whether the servlet is requested by the user typing the URL in the Address/Location box or by clicking a hyperlink also accounts for the presence of an HTTP request header called referer.

The output of the code above is shown in Figure below:
Output obtained in Google Chrome:


http://localhost:8080/exa...
http://localhost:8080/examples/servlets/servlet/RegisterServlet
Google

```
user-agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.19 (KHTML, like Gecko) Chrome/1.0.154.48
Safari/525.19
accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
accept-language: en-US,en
accept-charset: ISO-8859-1,*,utf-8
host: localhost:8080
connection: Keep-Alive
```

Output obtained in Internet Explorer 7:


http://localhost:8080/examples/servlets/servlet/RegisterServlet - Windows Internet Explorer
http://localhost:8080/examples/servlets/servlet/RegisterServlet
Yahoo! Search
File
Edit
View
Favorites
Tools
Help
http://localhost:8080/examples/servlets/servlet/Regi...
Page
Tools
Done
Local intranet
100%

```
accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-shockwave-flash, */*
accept-language: en-us
ua-cpu: x86
user-agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727;
InfoPath.2)
host: localhost:8080
connection: Keep-Alive
```

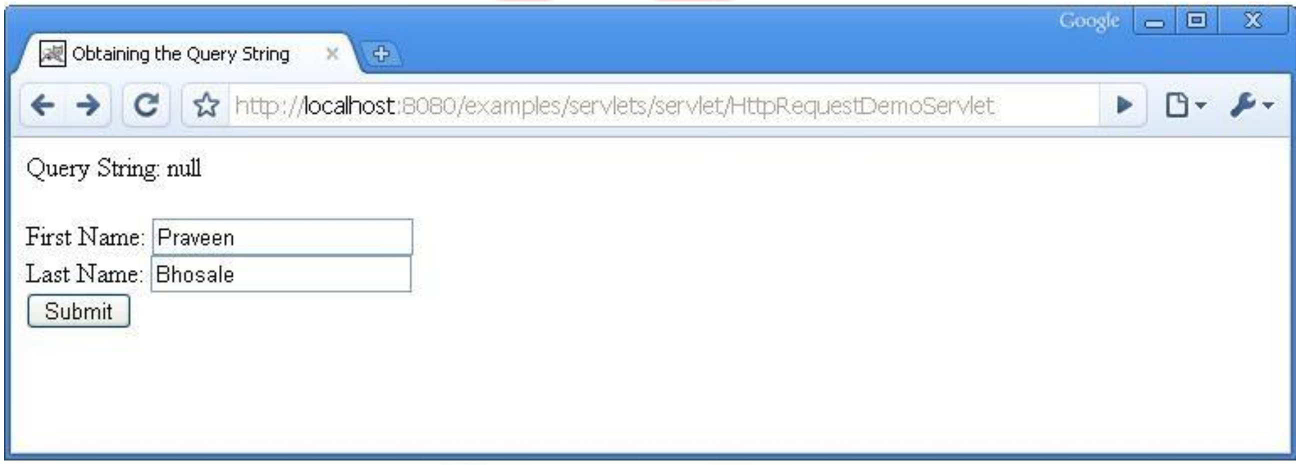
Obtaining the Query String from HttpServletRequest [Ref.1]

The next important method is the getQueryString method, which is used to retrieve the query string of the HTTP request. A query string is the string on the URL to the right of the path to the servlet.

If we use the GET method in an HTML form, the parameter name/value pairs will be appended to the URL. The code in Listing 3.3 is a servlet named HttpRequestDemoServlet that displays the value of the request's query string and a form.

```
//Obtaining the Query String
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class HttpRequestDemoServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Obtaining the Query String</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Query String: " + request.getQueryString() +
                "<BR>");
    out.println("<FORM METHOD=GET>");
    out.println("<BR>First Name: <INPUT TYPE=
                    TEXT NAME=FirstName>");
    out.println("<BR>Last Name: <INPUT TYPE=TEXT NAME=LastName>");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

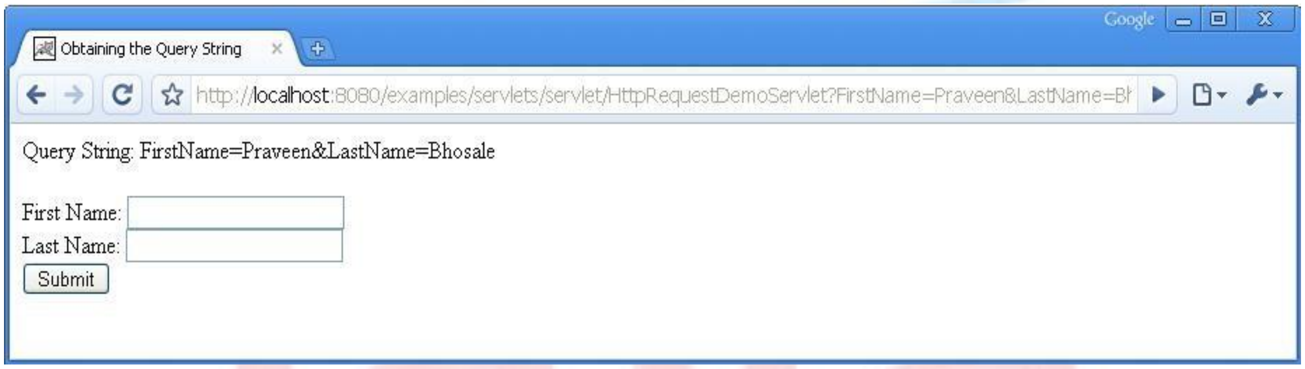
When the user enters the URL to the servlet in the web browser and the servlet is first called, the query string is null, as shown in Figure below:


Obtaining the Query String
http://localhost:8080/examples/servlets/servlet/HttpRequestDemoServlet

Query String: null

First Name: Praveen
Last Name: Bhosale
Submit

After we enter some values into the HTML form and submit the form, the page is redisplayed. Note that now there is a string added to the URL. The query string has a value of the parameter name/value pairs separated by an ampersand (&). The page is shown in Figure below:


Obtaining the Query String
http://localhost:8080/examples/servlets/servlet/HttpRequestDemoServlet?FirstName=Praveen&LastName=Bh

Query String: FirstName=Praveen&LastName=Bhosale

First Name:
Last Name:
Submit

Obtaining the Parameters from HttpServletRequest [Ref.1]

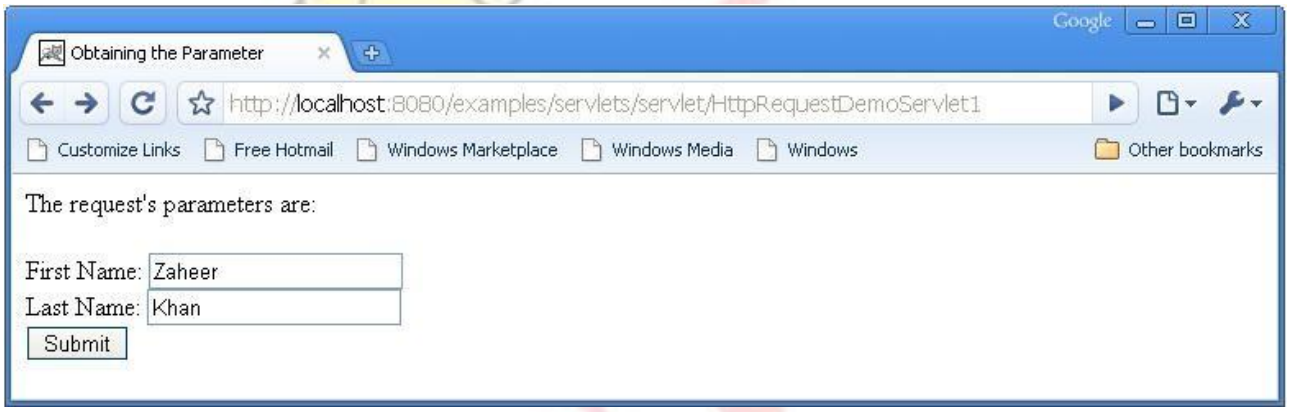
We have seen that we can get the query string containing a value. This means that we can get the form parameter name/value pairs or other values from the previous page. We should not use the getQueryString method to obtain a form's parameter name/value pairs, however, because this means we have to parse the string ourselves. We can use some other methods in HttpServletRequest to get the parameter names and values: the getParameterNames and the getParameter methods.

The getParameterNames method returns an Enumeration containing the parameter names. In many cases, however, we already know the parameter names, so we don't need to use this method. To get a parameter value, we use the getParameter method, passing the parameter name as the argument.

The following example demonstrates how we can use the getParameterNames and the getParameter methods to display all the parameter names and values from the HTML form from the previous page. The code is given below:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class HttpRequestDemoServlet1 extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
  throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Obtaining the Parameter</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("The request's parameters are:<BR>");
    Enumeration enumeration = request.getParameterNames();
    while (enumeration.hasMoreElements())
    {
      String parameterName = (String) enumeration.nextElement();
      out.println(parameterName + ": " +
        request.getParameter(parameterName) + "<BR>" );
    }
    out.println("<FORM METHOD=GET>");
    out.println("<BR>First Name: <INPUT TYPE=TEXT
                          NAME=FirstName>");
    out.println("<BR>Last Name: <INPUT TYPE=TEXT NAME=LastName>");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

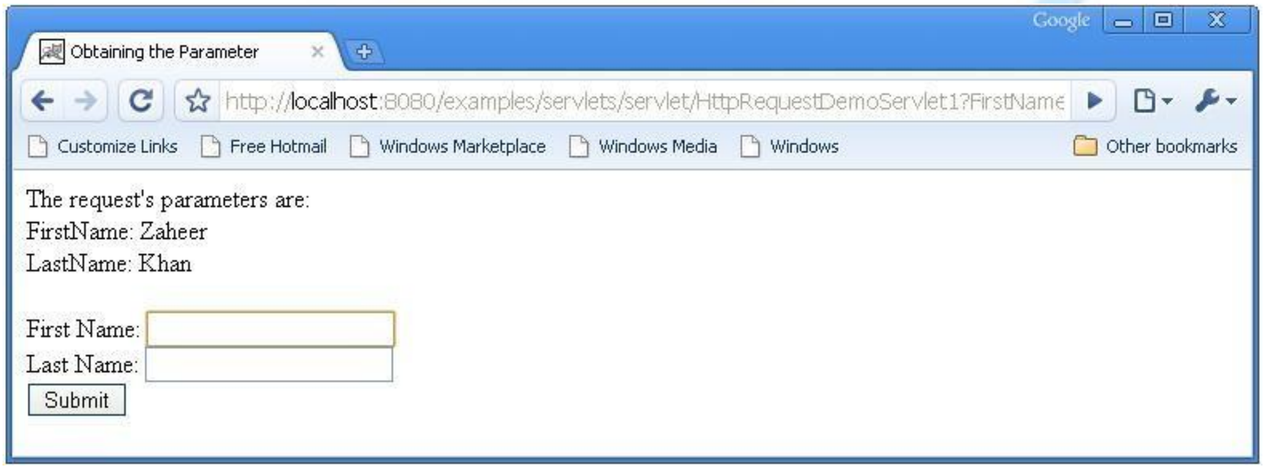
When the servlet is first called, it does not have any parameter from the previous request. Therefore, the no parameter name/value pair is displayed, as shown in Figure below:



The request's parameters are:

First Name: Zaheer
Last Name: Khan

On subsequent requests, the user should enter values for both the firstName and lastName parameters. This is reflected on the next page, which is shown in Figure below:



The request's parameters are:
FirstName: Zaheer
LastName: Khan

First Name:
Last Name:

Manipulating Multi-Value Parameters [Ref.1]

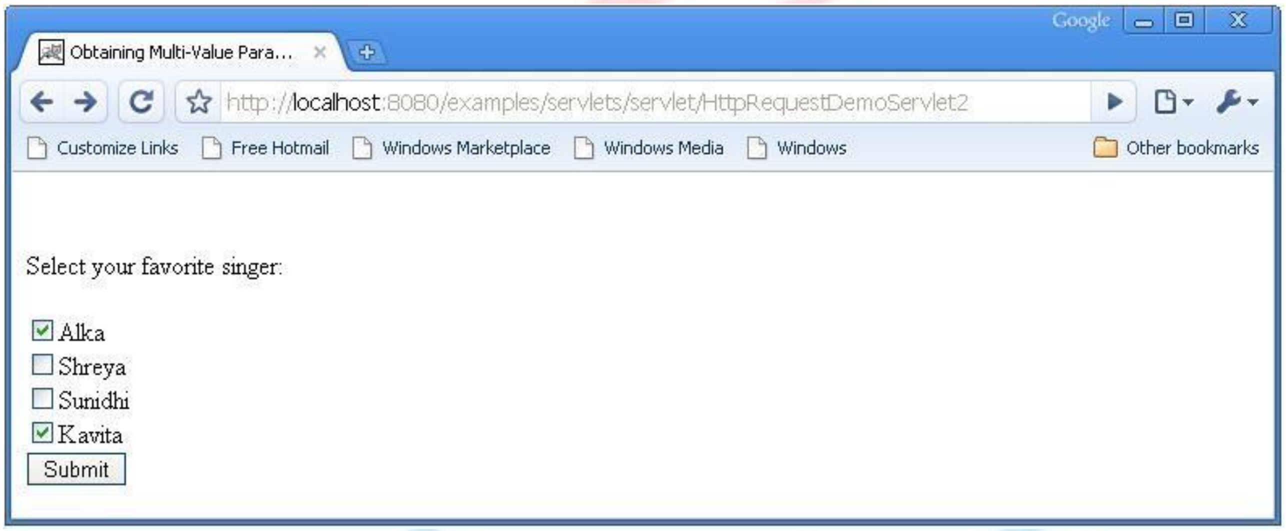
We may have a need to use parameters with the same name in our form. This case might arise, for example, when we are using check box controls that can accept multiple values or when we have a multiple-selection HTML select control. In situations like these, we can't use the getParameter method because it will give us only the first value. Instead, we use the getParameterValues method.

The getParameterValues method accepts one argument: the parameter name. It returns an array of string containing all the values for that parameter. If the parameter of that name is not found, the getParameterValues method will return a null.

The following example illustrates the use of the getParameterValues method to get all favorite music selected by the user. The code for this servlet is given in program below:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class HttpRequestDemoServlet2 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Obtaining Multi-Value Parameters</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<BR>");
    out.println("<BR>Select your favorite singer:");
    out.println("<BR><FORM METHOD=POST>");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Alka>Alka");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Shreya>Shreya");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Sunidhi>Sunidhi");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Kavita>Kavita");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException {
    String[] values = request.getParameterValues("favoriteMusic");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if (values != null ) {
      int length = values.length;
      out.println("You have selected: ");
      for (int i=0; i<length; i++) {
        out.println("<BR>" + values[i]);
      }
    }
    }
}
```

When the servlet is first called, the doGet method is invoked and the method sends a form to the web browser. The form has four check box controls with the same name: favoriteMusic. Their values are different, however. This is shown in Figure below:

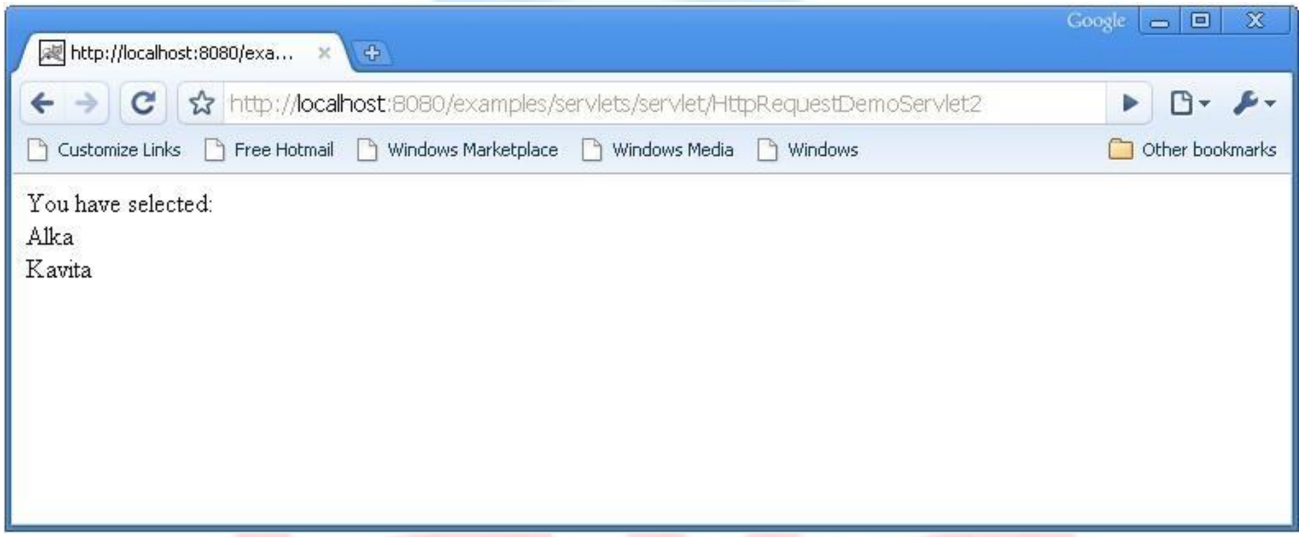

Obtaining Multi-Value Para...
http://localhost:8080/examples/servlets/servlet/HttpRequestDemoServlet2
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Other bookmarks

Select your favorite singer:

☑ Alka
☐ Shreya
☐ Sunidhi
☑ Kavita

Submit

When the user selects the value(s) of the check boxes, the browser sends all selected values. In the server side, we use the getParameterValues to retrieve all values sent in the request. This is shown in Figure below:


http://localhost:8080/exa...
http://localhost:8080/examples/servlets/servlet/HttpRequestDemoServlet2
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Other bookmarks

```
You have selected:
Alka
Kavita
```

Note that we use the POST method for the form; therefore, the parameter name/value pairs are retrieved in the doPost method.

HttpServletResponse [Ref.1]

The HttpServletResponse interface provides several protocol-specific methods not available in the javax.servlet.ServletResponse interface.

The HttpServletResponse interface extends the ServletResponse interface. Till in the examples, we have seen that we always use two of the methods in HttpServletResponse when sending output to the browser: setContentType and getWriter.

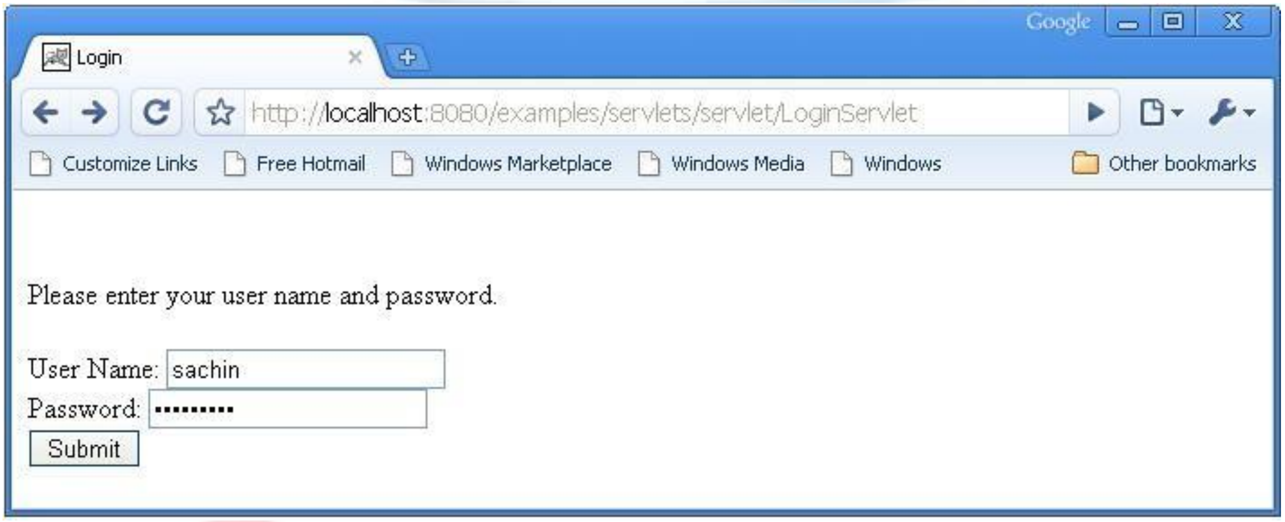
```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

There is more to it, however. The addCookie method sends cookies to the browser. We also use methods to manipulate the URLs sent to the browser. Another interesting method in the HttpServletResponse interface is the setHeader method. This method allows us to add a name/value field to the response header.

We can also use a method to redirect the user to another page: sendRedirect. When we call this method, the web server sends a special message to the browser to request another page. Therefore, there is always a round trip to the client side before the other page is fetched. This method is used frequently and its use is illustrated in the following example. The example below shows a Login page that prompts the user to enter a user name and a password. If both are correct, the user will be redirected to a Welcome page. If not, the user will see the same Login page.

When the servlet is first requested, the servlet's doGet method is called. The doGet method then outputs the form. The user can then enter the user name and password, and submit the form. Note that the form uses the POST method, which means that at the server side, the doPost method is invoked, and the user name and password are checked against some predefined values. If the user name and password match, the user is redirected to a Welcome page. If not, the doPost method outputs the Login form again along with an error message.

```
public class LoginServlet extends HttpServlet {
  private void sendLoginForm(HttpServletResponse response,
    boolean withErrorMessage)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Login</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    if (withErrorMessage)
      out.println("Login failed. Please try again.<BR>");

    out.println("<BR>");
    out.println("<BR>Please enter your user name and password.");
    out.println("<BR><FORM METHOD=POST>");
    out.println("<BR>User Name: <INPUT TYPE=TEXT NAME=userName>");
    out.println("<BR>Password: <INPUT TYPE=PASSWORD
                    NAME=password>");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    sendLoginForm(response, false);
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException {
    String userName = request.getParameter("userName");
    String password = request.getParameter("password");
    if (userName!=null && password!=null &&
      userName.equals("james.bond") && password.equals("007")) {
      response.sendRedirect("http://domain/app/WelcomePage");
    }
    else
      sendLoginForm(response, true);
  }
}
```

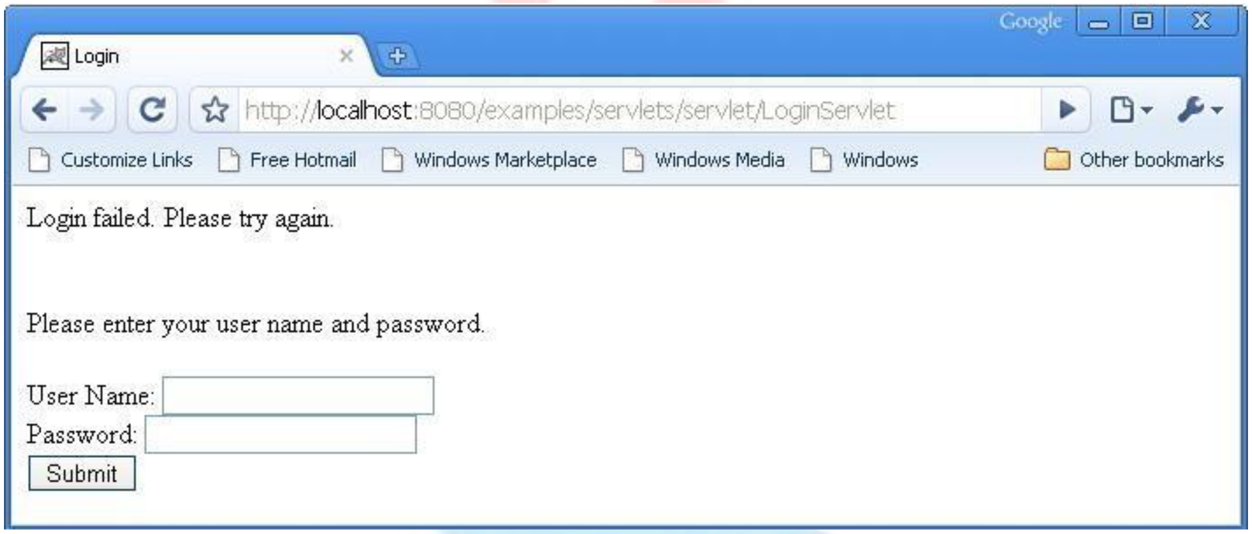


Please enter your user name and password.

User Name: sachin
Password: •••••••••
Submit

In the code given above, private method called sendLoginForm that accepts an HttpServletResponse object and a boolean that signals whether an error message be sent along with the form. This sendLoginForm method is called both from the doGet and the doPost methods. When called from the doGet method, no error message is given, because this is the first time the user requests the page. The withErrorMessage flag is therefore false. When called from the doPost method, this flag is set to true because the sendLoginForm method is only invoked from doPost if the user name and password did not match.

The Login page, when it is first requested, is shown in Figure above. The Login page, after a failed attempt to log in, is shown in Figure below:



Login failed. Please try again.

Please enter your user name and password.

User Name:
Password:
Submit

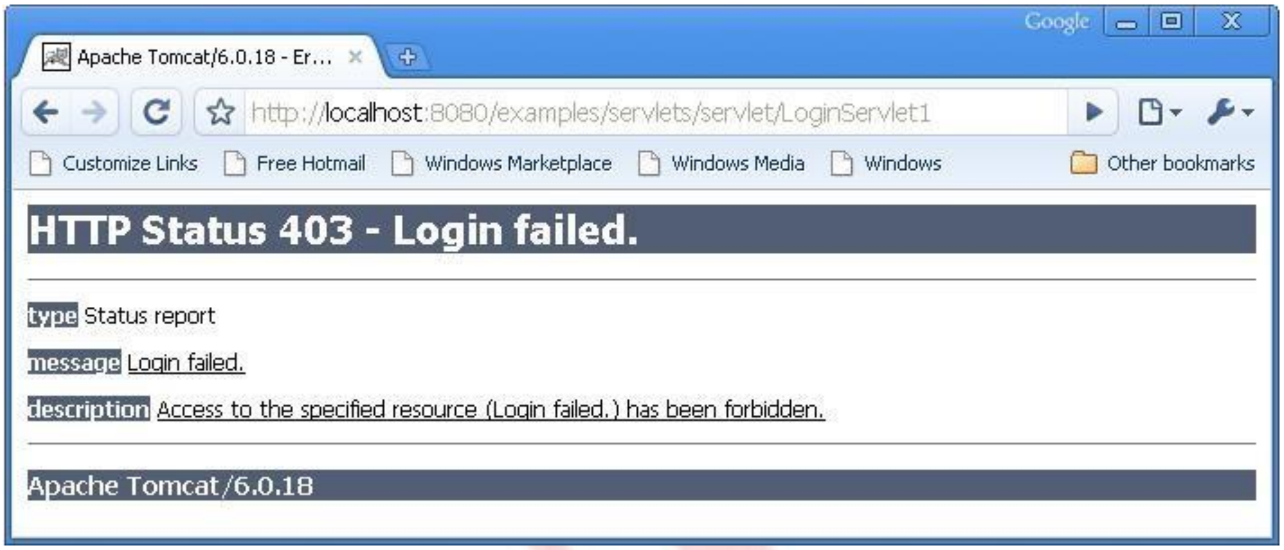
Sending an Error Code [Ref.1]

The HttpServletResponse also allows us to send pre-defined error messages. The interface defines a number of public static final integers that all start with SC_. For example, SC_FORBIDDEN will be translated into an HTTP error 403.

Along with the error code, we also can send a custom error message. Instead of redisplaying the Login page when a failed login occurs, we can send an HTTP error 403 plus our error message. To do this, replace the call to the sendLoginForm in the doPost method with the following:

```
response.sendError(response.SC_FORBIDDEN, "Login failed.");
```

The user will see the screen in following Figure when a login fails.


Apache Tomcat/6.0.18 - Er...
http://localhost:8080/examples/servlets/servlet/LoginServlet1
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Other bookmarks

HTTP Status 403 - Login failed.

type Status report

message Login failed.

description Access to the specified resource (Login failed.) has been forbidden.

Apache Tomcat/6.0.18

Request Dispatching [Ref.1]

In some circumstances, we may want to include the content from an HTML page or the output from another servlet. Additionally, there are cases that require that we pass the processing of an HTTP request from our servlet to another servlet. The current servlet specification responds to these needs with an interface called RequestDispatcher, which is found in the javax.servlet package. This interface has two methods, which allow you to delegate the request-response processing to another resource: include and forward. Both methods accept a ServletRequest object and a ServletResponse object as arguments.

As the name implies, the include method is used to include content from another resource, such as another servlet, a JSP page, or an HTML page. The method has the following signature:

```
public void include(javax.servlet.ServletRequest request,
  javax.servlet.ServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException
```

The forward method is used to forward a request from one servlet to another. The original servlet can perform some initial tasks on the ServletRequest object before forwarding it. The signature of the forward method is as follows:

```
public void forward(javax.servlet.ServletRequest request,
  javax.servlet.ServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException
```

The Difference Between sendRedirect and forward:

The sendRedirect method works by sending a status code that tells the browser to request another URL. This means that there is always a round trip to the client side. Additionally, the previous HttpServletRequest object is lost. To pass information between the original servlet and the next request, we normally pass the information as a query string appended to the destination URL.

The forward method, on the other hand, redirects the request without the help from the client's browser. Both the HttpServletRequest object and the HttpServletResponse object also are passed to the new resource.

In order to perform a servlet include or forward, we first need to obtain a RequestDispatcher object. We can obtain a RequestDispatcher object three different ways, as follows:

- Use the getRequestDispatcher method of the ServletContext interface, passing a String containing the path to the other resource. The path is relative to the root of the ServletContext.
- Use the getRequestDispatcher method of the ServletRequest interface, passing a String containing the path to the other resource. The path is relative to the current HTTP request.
- Use the getNamedDispatcher method of the ServletContext interface, passing a String containing the name of the other resource.

Including Static Content [Ref.1]

Sometimes we need to include static content, such as HTML pages or image files that are prepared by a web graphic designer. We can do this by using the same technique for including dynamic resources.

The following example shows a servlet named FirstServlet that includes an HTML file named main.html. The servlet class file is located in the WEB-INF\classes directory, whereas the AdBanner.html file, like other HTML files, resides in the \examples directory. The servlet is given in program below and the HTML file is also given.

```
//Including Static Content
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FirstServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher rd
                    = request.getRequestDispatcher("/main.html");
    rd.include(request, response);
  }
}
```

//main.html File

```
<HTML>
<HEAD>
<TITLE>Banner</TITLE>
</HEAD>
<BODY>
<IMG SRC=quote01.jpg>
</BODY>
</HTML>
```

Including another Servlet [Ref.1]

The second example shows a servlet (FirstServlet) that includes another servlet (SecondServlet). The second servlet simply sends the included request parameter to the user. The FirstServlet and the SecondServlet is presented below.

//FirstServlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FirstServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Included Request Parameters</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<B>Included Request Parameters</B><BR>");
    RequestDispatcher rd =
  request.getRequestDispatcher("/servlet/SecondServlet?name=budi");
    rd.include(request, response);
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

//SecondServlet

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

```
public class SecondServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Enumeration enum = request.getAttributeNames();
    while (enum.hasMoreElements()) {
      String attributeName = (String) enum.nextElement();
      out.println(attributeName + ": " +
        request.getAttribute(attributeName) + "<BR>");
    }
  }
}
```

Session Tracking and Management [Ref.1]

The Hypertext Transfer Protocol (HTTP) is the network protocol that web servers and client browsers use to communicate with each other. HTTP is the language of the web. HTTP connections are initiated by a client browser that sends an HTTP request. The web server then responds with an HTTP response and closes the connection. If the same client requests another resource from the server, it must open another HTTP connection to the server. The server always closes the connection as soon as it sends the response, whether or not the browser user needs some other resource from the server.

This process is similar to a telephone conversation in which the receiver always hangs up after responding to the last remark/question from the caller. For example, a call goes something like this:

Caller dials. Caller gets connected.

Caller: "Hi, good morning."

Receiver: "Good morning."

Receiver hangs up.

Caller dials again. Caller gets connected.

Caller: "May I speak to Dr. Divakar, please?"

Receiver: "Sure."

Receiver hangs up.

Caller dials again, and so on, and so on.

Putting this in a web perspective, because the web server always disconnects after it responds to a request, the web server does not know whether a request comes from a user who has just requested the first page or from a user who has requested nine other pages before. As such, HTTP is said to be *stateless*.

Being stateless has huge implications. Consider, for example, a user who is shopping at an online store. As usual, the process starts with the user searching for a product. If the product is found, the user then enters the quantity of that product into the shopping cart form and submits it to the server. But, the user is not yet checking out—he still wants to buy something else. So he searches the catalog again for the second product. The first product order has now been lost, however, because the previous connection was closed and the web server does not remember anything about the previous connection.

The good news is that web programmers can work around this. The solution is called user session management. The web server is forced to associate HTTP requests and client browsers. There are four different ways of session tracking:

- User Authentication
- Hidden from fields
- URL Re-writing
- Persistent cookies

User Authentication [Ref.2]

One way to perform session tracking is to leverage the information that comes with user authentication. It occurs when a web server restricts access to some of its resources to only those clients that log in using a recognized username and password. After the client logs in, the username is available to a servlet through getRemoteUser( ).

We can use the username to track a client session. Once a user has logged in, the browser remembers her username and resends the name and password as the user views new pages on the site. A servlet can identify the user through her username and thereby track his session. For example, if the user adds an item to his virtual shopping cart, that fact can be remembered (in a shared class or external database, perhaps) and used later by another servlet when the user goes to the check-out page.

For example, a servlet that utilizes user authentication might add an item to a user's shopping cart with code like the following:

```
String name = req.getRemoteUser();
if (name == null) {
  // Explain that the server administrator should protect this page
}
else {
  String[] items = req.getParameterValues("item");
  if (items != null) {
    for (int i = 0; i < items.length; i++) {
      addItemToCart(name, items[i]);
    }
  }
}
```

Another servlet can then retrieve the items from a user's cart with code like this:

```
String name = req.getRemoteUser();
if (name == null) {
  // Explain that the server administrator should protect this page
}
else {
  String[] items = getItemsFromCart(name);
}
```

The biggest advantage of using user authentication to perform session tracking is that it's easy to implement. Simply tell the server to protect a set of pages, and use getRemoteUser( ) to identify each client. Another advantage is that the technique works even when the user accesses our site from different machines. It also works even if the user strays from our site or exits his browser before coming back.

The biggest disadvantage of user authentication is that *it requires each user to register for an account and then log in each time he starts visiting our site.* Most users will tolerate registering and logging in as a necessary evil when they are accessing sensitive information, but it's overkill for simple session tracking. Another downside is that HTTP's basic authentication provides *no logout mechanism*; the user has to exit his browser to log out. A final problem with user authentication is that *a user cannot simultaneously maintain more than one session at the same site*. We clearly need alternative approaches to support anonymous session tracking and to support authenticated session tracking with logout.

Hidden Form Fields [Ref.2]

One way to support anonymous session tracking is to use hidden form fields. As the name implies, these are fields added to an HTML form that are not displayed in the client's browser. They are sent back to the server when the form that contains them is submitted. You include hidden form files with HTML like this:

```
<FORM ACTION="/servlet/MovieFinder" METHOD="POST">
```

```
     ...
     <INPUT TYPE=hidden NAME="zip" VALUE="94040">
     <INPUT TYPE=hidden NAME="level" VALUE="expert">
     ...
     </FORM>
```

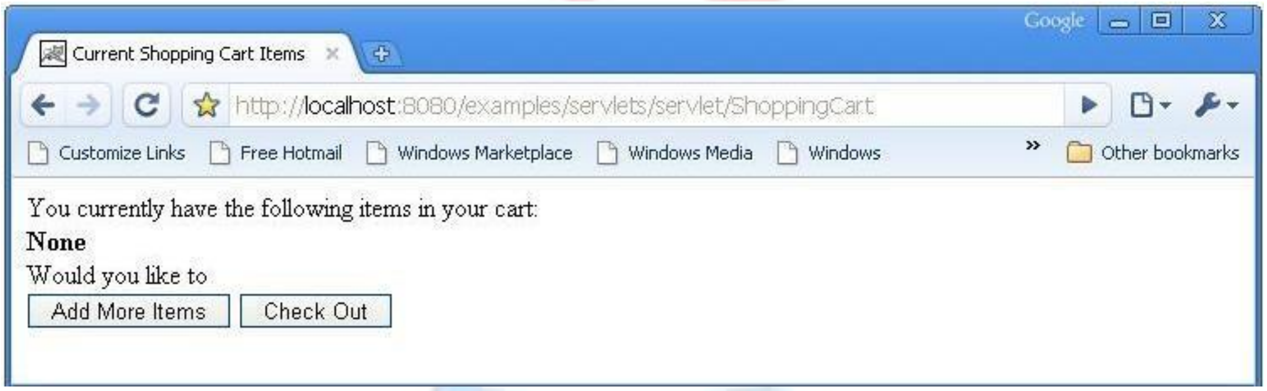
In a sense, hidden form fields define constant variables for a form. To a servlet receiving a submitted form, there is no difference between a hidden field and a visible field.

With hidden form fields, we can rewrite our shopping cart servlets so that users can shop anonymously until checkout time. Example given below demonstrates the technique with a servlet that displays the user's shopping cart contents and lets the user choose to add more items or check out.

//Session Tracking Using Hidden Form Fields

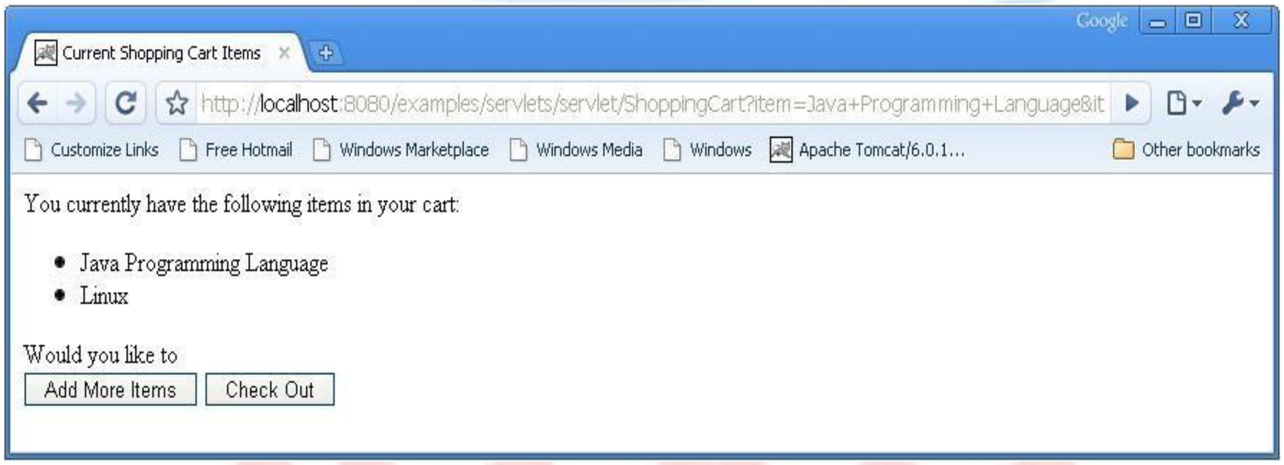
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShoppingCart extends HttpServlet {
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res) throws
               ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HEAD><TITLE>Current Shopping Cart
               Items</TITLE></HEAD><BODY>");

    // Cart items are passed in as the item parameter.
    String[] items = req.getParameterValues("item");

    // Print the current cart items.
    out.println("You currently have the following items in
                    your cart:<BR>");
    if (items == null)
      out.println("<B>None</B>");
    else
    {
      out.println("<UL>");
      for (int i = 0; i < items.length; i++)
        out.println("<LI>" + items[i]);
      out.println("</UL>");
    }

    // Ask if the user wants to add more items or check out.
    // Include the current items as hidden fields so they'll
    // be passed on.
    out.println("<FORM ACTION =
     \"/examples/servlets/servlet/ShoppingCart\" METHOD = POST>");
    if (items != null) {
      for (int i = 0; i < items.length; i++)
        out.println("<INPUT TYPE=HIDDEN NAME=\"item\" VALUE=\"" +
          items[i] + "\">");
    }
    out.println("Would you like to<BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Add More Items \">");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Check Out \">");
    out.println("</FORM></BODY></HTML>");
  }
}
```

First view of the window:


Current Shopping Cart Items
http://localhost:8080/examples/servlets/servlet/ShoppingCart
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Other bookmarks

You currently have the following items in your cart:
None
Would you like to
Add More Items | Check Out

After entering the URL as :
http://localhost:8080/examples/servlets/servlet/ShoppingCart?item=Java+Programming+Language&item=Linux


Current Shopping Cart Items
http://localhost:8080/examples/servlets/servlet/ShoppingCart?item=Java+Programming+Language&it
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Apache Tomcat/6.0.1...
Other bookmarks

You currently have the following items in your cart:

- Java Programming Language
- Linux

Would you like to
Add More Items | Check Out

This servlet first reads the items that are already in the cart using getParameterValues("item"). Presumably, the item parameter values were sent to this servlet using hidden fields. The servlet then displays the current items to the user and asks if he wants to add more items or check out. The servlet asks its question with a form that includes hidden fields, so the form's target (the ShoppingCart servlet) receives the current items as part of the submission.

As more and more information is associated with a client's session, it can become burdensome to pass it all using hidden form fields. In these situations, it's possible to pass on just a unique session ID that identifies a particular client's session. That session ID can be associated with complete information about the session that is stored on the server.

Beware that session IDs must be held as a server secret because any client with knowledge of another client's session ID can, with a forged hidden form field, assume the second client's identity. Consequently, session IDs should be generated so as to be difficult to guess or forge, and active session IDs should be protected—for example, don't make public the server's access log because the logged URLs may contain session IDs for forms submitted with GET requests.

Hidden form fields can be used to implement authentication with logout. Simply present an HTML form as the logon screen, and once the user has been authenticated by the server her identity can be associated with her particular session ID. On logout the session ID can be deleted (by not sending the ID to the client on later forms), or the association between ID and user can simply be forgotten.

The advantages of hidden form fields are *their ubiquity and support for anonymity*. Hidden fields are supported in all the popular browsers, they demand no special server requirements, and they can be used with clients that haven't registered or logged in. The major disadvantage with this technique, however, is that the session persists only through sequences of dynamically generated forms. The session cannot be maintained with static documents, emailed documents, bookmarked documents, or browser shutdowns.

URL Rewriting [Ref.2]

URL rewriting is another way to support anonymous session tracking. With URL rewriting, every local URL the user might click on is dynamically modified, or rewritten, to include extra information. The extra information can be in the form of extra path information, added parameters, or some custom, server-specific URL change. Due to the limited space available in rewriting a URL, the extra information is usually limited to a unique session ID. For example, the following URLs have been rewritten to pass the session ID 123.

`http://server:port/servlet/Rewritten`	original
`http://server:port/servlet/Rewritten/123`	extra path information
`http://server:port/servlet/Rewritten?sessionid=123`	added parameter
`http://server:port/servlet/Rewritten;jsessionid=123`	custom change

Each rewriting technique has its advantages and disadvantages. *Using extra path information works on all servers, but it doesn't work well if a servlet has to use the extra path information as true path information*. Using an added parameter works on all servers too, but it can cause parameter naming collisions. Using a custom, server-specific change works under all conditions for servers that support the change. Unfortunately, it doesn't work at all for servers that don't support the change.

Cookies [Ref.1]

The fourth technique that we can use to manage user sessions is by using cookies. A cookie is a small piece of information that is passed back and forth in the HTTP request and response. Even though a cookie can be created on the client side using some scripting language such as JavaScript, it is usually created by a server resource, such as a servlet. The cookie sent by a servlet to the client will be passed back to the server when the client requests another page from the same application.

Cookies were first specified by Netscape (see http://home.netscape.com/newsref/std/cookie_spec.html) and are now part of the Internet standard as specified in RFC 2109: The HTTP State Management Mechanism. Cookies are transferred to and from the client in the HTTP headers.

In servlet programming, a cookie is represented by the Cookie class in the javax.servlet.http package. We can create a cookie by calling the Cookie class constructor and passing two String objects: the name and value of the cookie. For instance, the following code creates a cookie object called c1. The cookie has the name "myCookie" and a value of "secret":

```
Cookie c1 = new Cookie("myCookie", "secret");
```

We then can add the cookie to the HTTP response using the addCookie method of the HttpServletResponse interface:

```
response.addCookie(c1);
```

Note that because cookies are carried in the request and response headers, we must not add a cookie after an output has been written to the HttpServletResponse object. Otherwise, an exception will be thrown.

The following example shows how we can create two cookies called userName and password and illustrates how those cookies are transferred back to the server. The servlet is called CookieServlet, and its code is given in program below:

When it is first invoked, the doGet method of the servlet is called. The method creates two cookies and adds both to the HttpServletResponse object, as follows:

```
    Cookie c1 = new Cookie("userName", "Helen");
    Cookie c2 = new Cookie("password", "Keppler");
    response.addCookie(c1);
    response.addCookie(c2);
```

Next, the doGet method sends an HTML form that the user can click to send another request to the servlet:

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Cookie Test</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("Please click the button to see
                        the cookies sent to you.");
        out.println("<BR>");
        out.println("<FORM METHOD=POST>");
        out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
        out.println("</FORM>");
        out.println("</BODY>");
        out.println("</HTML>");
```

The form does not have any element other than a submit button. When the form is submitted, the doPost method is invoked. The doPost method does two things: It iterates all the headers in the request to show how the cookies are conveyed back to the server, and it retrieves the cookies and displays their values.

For displaying all the headers in the HttpServletRequest method, it first retrieves an Enumeration object containing all the header names. The method then iterates the Enumeration object to get the next header name and passes the header name to the getHeader method to display the value of that header, as you see here:

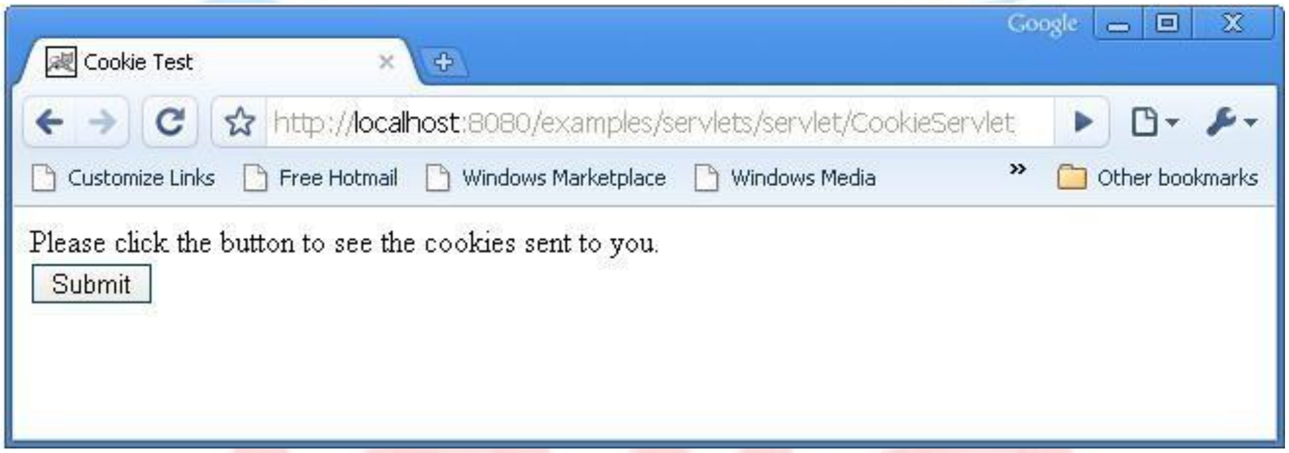
```
    Enumeration enumr = request.getHeaderNames();
    while (enumr.hasMoreElements())
    {
      String header = (String) enumr.nextElement();
      out.print("<B>" + header + "</B>: ");
      out.print(request.getHeader(header) + "<BR>");
    }
```

In order to retrieve cookies, we use the getCookies method of the HttpServletRequest interface. This method returns a Cookie array containing all cookies in the request. It is our responsibility to loop through the array to get the cookie we want, as follows:

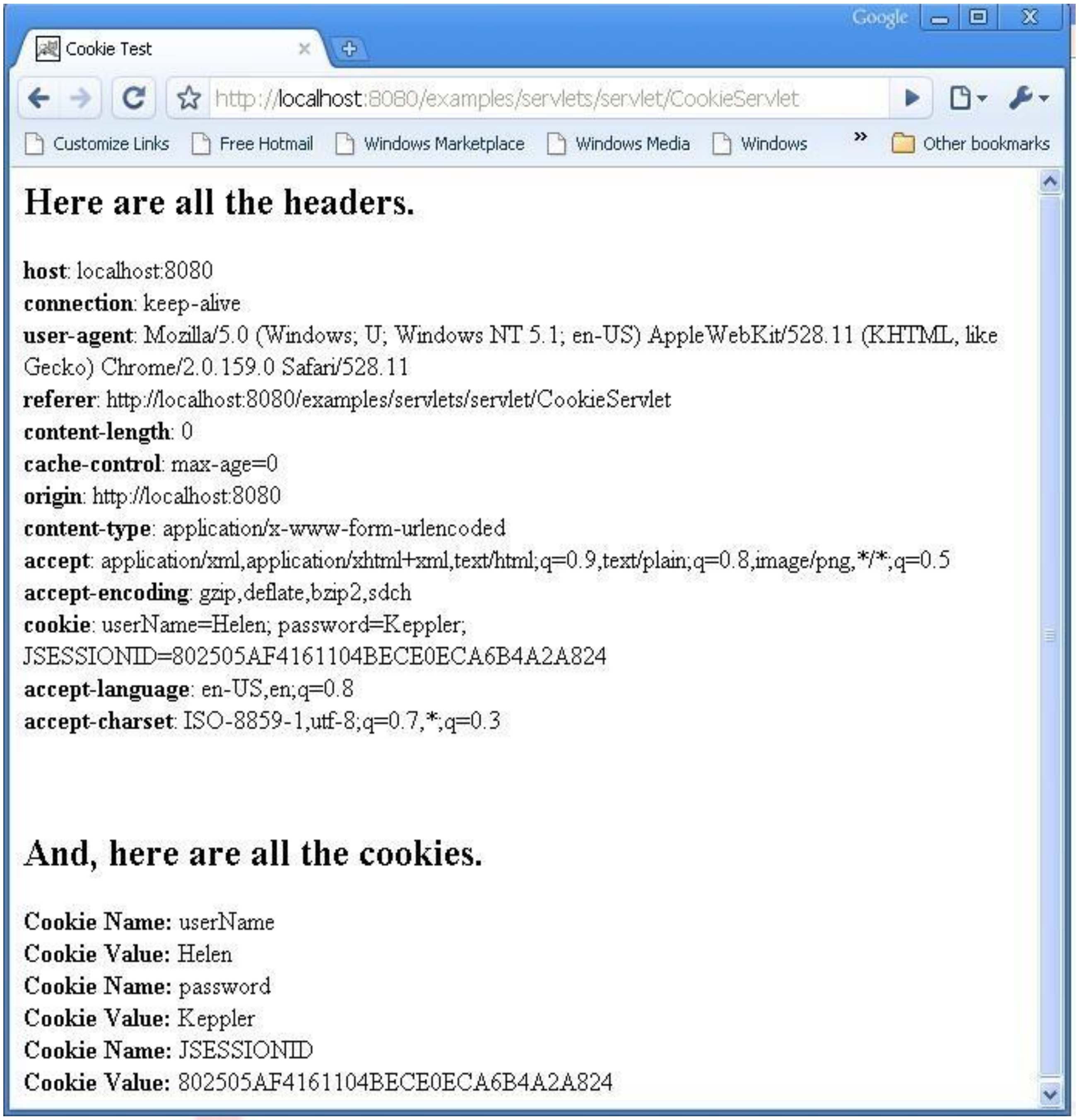
```
     Cookie[] cookies = request.getCookies();
     int length = cookies.length;
     for (int i=0; i<length; i++)
     {
       Cookie cookie = cookies[i];
       out.println("<B>Cookie Name:</B> " +
         cookie.getName() + "<BR>");
       out.println("<B>Cookie Value:</B> " +
         cookie.getValue() + "<BR>");
     }
```

//The Cookie Servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class CookieServlet extends HttpServlet {
  /**Process the HTTP Get request*/
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    Cookie c1 = new Cookie("userName", "Helen");
    Cookie c2 = new Cookie("password", "Keppler");
    response.addCookie(c1);
    response.addCookie(c2);

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Cookie Test</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Please click the button to see
                    the cookies sent to you.");
    out.println("<BR>");
    out.println("<FORM METHOD=POST>");
    out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
  /**Process the HTTP Post request*/
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response) throws
                         ServletException,IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Cookie Test</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<H2>Here are all the headers.</H2>");

    Enumeration enumr = request.getHeaderNames();
    while (enumr.hasMoreElements()) {
      String header = (String) enumr.nextElement();
      out.print("<B>" + header + "</B>: ");
      out.print(request.getHeader(header) + "<BR>");
    }
    out.println("<BR><BR><H2>And, here are all the cookies.</H2>");
    Cookie[] cookies = request.getCookies();
    int length = cookies.length;
    for (int i=0; i<length; i++)
    {
      Cookie cookie = cookies[i];
      out.println("<B>Cookie Name:</B> " + cookie.getName()
                              + "<BR>");
      out.println("<B>Cookie Value:</B> " + cookie.getValue()
                              + "<BR>");
    }
     out.println("</BODY>");
     out.println("</HTML>");
   }
}
```



Please click the button to see the cookies sent to you.



Here are all the headers.

host: localhost:8080
connection: keep-alive
user-agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/528.11 (KHTML, like Gecko) Chrome/2.0.159.0 Safari/528.11
referer: http://localhost:8080/examples/servlets/servlet/CookieServlet
content-length: 0
cache-control: max-age=0
origin: http://localhost:8080
content-type: application/x-www-form-urlencoded
accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
accept-encoding: gzip,deflate,bzip2,sdch
cookie: userName=Helen; password=Keppler; JSESSIONID=802505AF4161104BECE0ECA6B4A2A824
accept-language: en-US,en;q=0.8
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

And, here are all the cookies.

Cookie Name: userName
Cookie Value: Helen
Cookie Name: password
Cookie Value: Keppler
Cookie Name: JSESSIONID
Cookie Value: 802505AF4161104BECE0ECA6B4A2A824

Simple Cookie operations

Setting Cookie Attributes:

The Cookie class provides a number of methods for setting a cookie's values and attributes. Using these methods is straightforward. The following example sets the comment field of the Servlet's cookie. The comment field describes the purpose of the cookie.

```
  public void doGet (HttpServletRequest request,
                          HttpServletResponse response)
      throws ServletException, IOException
     {
         ...
         // If the user wants to add a book, remember it
         // by adding a cookie
         if (values != null)
         {
             bookId = values[0];
             Cookie getBook = new Cookie("Buy", bookId);
             getBook.setComment("User wants to buy this book " +
                                "from the bookstore.");
         }
         ...
     }
```

We can also set the maximum age of the cookie. This attribute is useful, for example, for deleting a cookie. Once again, if Duke's Bookstore kept track of a user's order with cookies, the example would use this attribute to delete a book from the user's order. The user removes a book from the shopping cart in the Servlet; its code would look something like this:

```
    public void doGet (HttpServletRequest request,
                          HttpServletResponse response)
      throws ServletException, IOException
     {
      ...
         /* Handle any pending deletes from the shopping cart */
         String bookId = request.getParameter("Remove");
         ...
         if (bookId != null) {
             // Find the cookie that pertains to the book to remove
             ...
             // Delete the cookie by setting its maximum age to zero
             thisCookie.setMaxAge(0);
             ...
         }
      // also set content type header before accessing the Writer
         response.setContentType("text/html");
         PrintWriter out = response.getWriter();
         //Print out the response
         out.println("<html> <head>" +
                     "<title>Your Shopping Cart</title>" + ...);
```

Sending the Cookie

Cookies are sent as headers of the response to the client; they are added with the addCookie method of the HttpServletResponse class. If we are using a

Writer to return text data to the client, we must call the addCookie method before calling the HttpServletResponse's getWriter method.
The following is code for sending the cookie:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
 throws ServletException, IOException
{
    ...
  //If user wants to add a book, remember it by adding a cookie
    if (values != null) {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User has indicated a desire " +
                      "to buy this book from the bookstore.");
        response.addCookie(getBook);
    }
    ...
}
```

Retrieving Cookies

Clients return cookies as fields added to HTTP request headers. To retrieve any cookie, we must retrieve all the cookies using the getCookies method of the HttpServletRequest class.
The getCookies method returns an array of Cookie objects, which we can search to find the cookie or cookies that we want. (Remember that multiple cookies can have the same name. In order to get the name of a cookie, use its getName method.)
For example:

```
public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
   throws ServletException, IOException
  {
   ...
      /* Handle any pending deletes from the shopping cart */
      String bookId = request.getParameter("Remove");
      ...
      if (bookId != null) {
         // Find the cookie that pertains to the book to remove
          Cookie[] cookies = request.getCookies();
          ...
          // Delete the book's cookie by setting its max age to 0
          thisCookie.setMaxAge(0);
      }
```

```
     // also set content type header before accessing the Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //Print out the response
        out.println("<html> <head>" +
                    "<title>Your Shopping Cart</title>" + ...);
```

Getting the Value of a Cookie

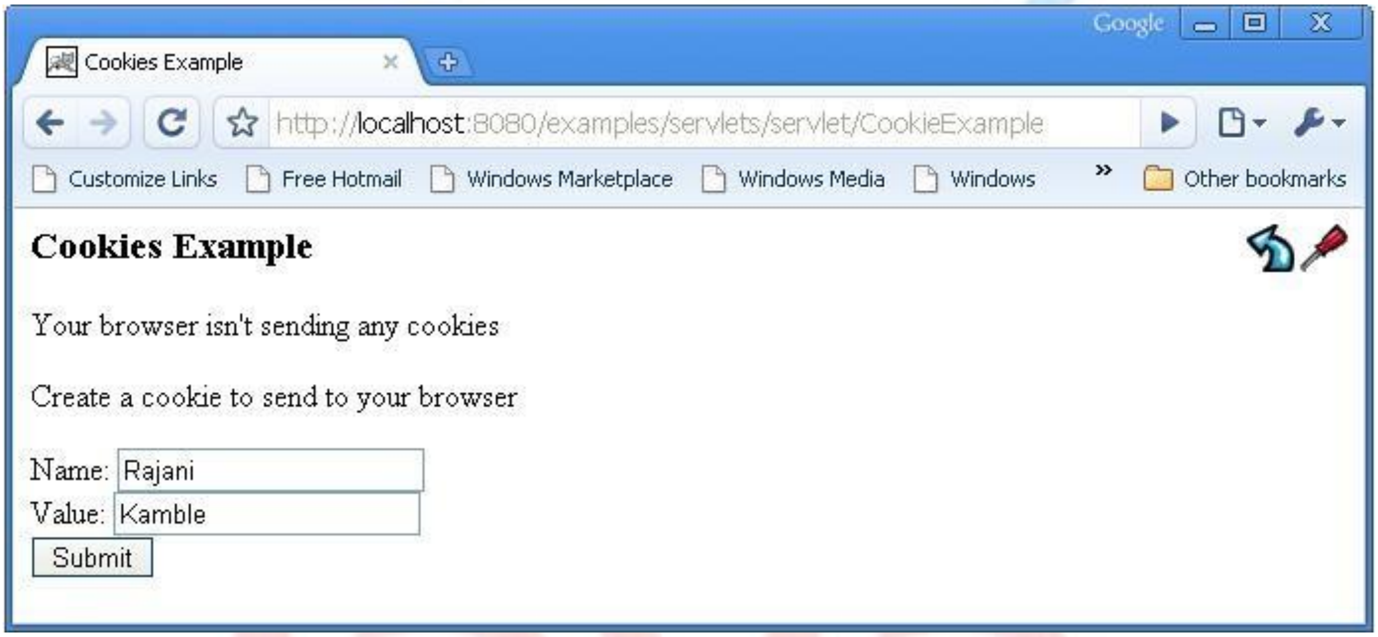
To find the value of a cookie, use its getValue method. For example:

```
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
     throws ServletException, IOException
    {
        ...
        /* Handle any pending deletes from the shopping cart */
        String bookId = request.getParameter("Remove");
        ...
        if (bookId != null) {
            // Find the cookie that pertains to that book
            Cookie[] cookies = request.getCookies();
            for(i=0; i < cookies.length; i++) {
                Cookie thisCookie = cookie[i];
                if (thisCookie.getName().equals("Buy") &&
                    thisCookie.getValue().equals(bookId)) {
                // Delete cookie by setting its maximum age to zero
                    thisCookie.setMaxAge(0);
                }
            }
        }
     // also set content type header before accessing the Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        //Print out the response
        out.println("<html> <head>" +
                    "<title>Your Shopping Cart</title>" + ...);
```

//Example of cookie:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieExample extends HttpServlet {
    public void doGet(   HttpServletRequest request,
                         HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // print out cookies
        Cookie[] cookies = request.getCookies();
        for (int i = 0; i < cookies.length; i++) {
            Cookie c = cookies[i];
            String name = c.getName();
            String value = c.getValue();
            out.println(name + " = " + value);
        }
        // set a cookie
        String name = request.getParameter("cookieName");
        if (name != null && name.length() > 0) {
            String value = request.getParameter("cookieValue");
            Cookie c = new Cookie(name, value);
            response.addCookie(c);
        }
    }
}
```

Cookies output windows:



Cookies Example

Your browser isn't sending any cookies

Create a cookie to send to your browser

Name: Rajani
Value: Kamble
Submit

Cookies E

Customize

Other bookmarks

Cookies

Your brows

You just ser
Name: Raja
Value: Kam

Create a co

Name: Sacl
Value: 1563
Submit

Cookies E

Customize

Other bookmarks

Cookies

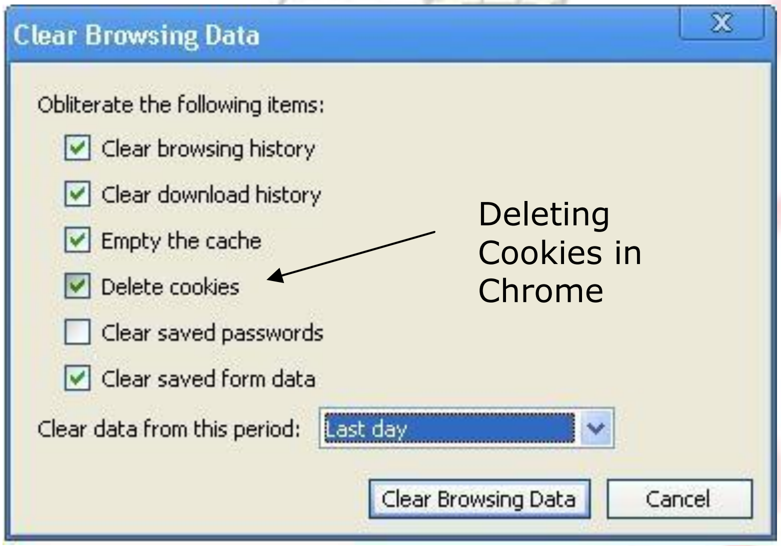
Your brows
Cookie Nar
Cookie Valı

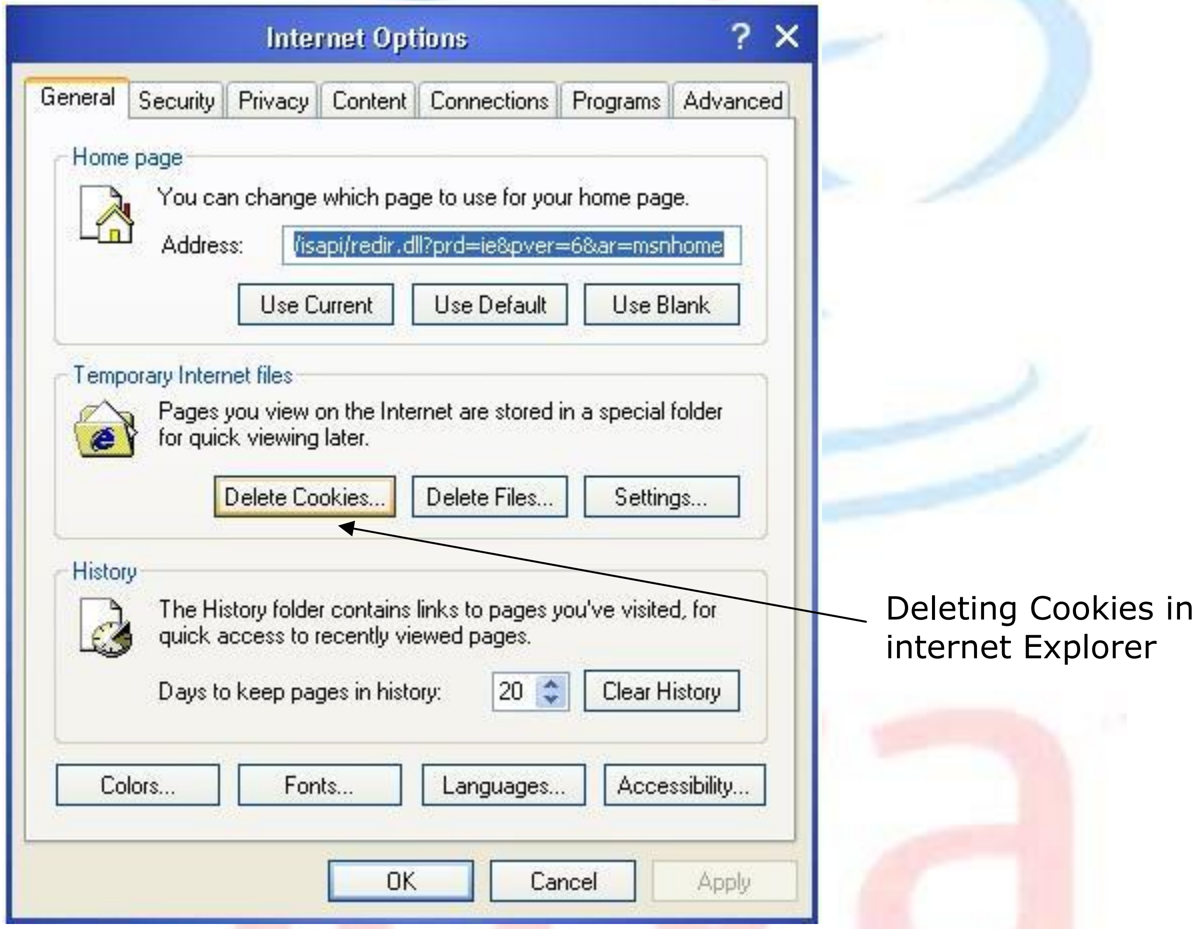
You just ser
Name: Sach
Value: 1563

Create a co

Name: India
Value: 112.
Submit

For deleting the persistent cookies, use following windows of Google Chrome and Internet Explorer respectively:




Internet Options
General
Security
Privacy
Content
Connections
Programs
Advanced
Home page
You can change which page to use for your home page.
Address: /isapi/redir.dll?prd=ie&pver=6&ar=msnhome
Use Current
Use Default
Use Blank
Temporary Internet files
Pages you view on the Internet are stored in a special folder for quick viewing later.
Delete Cookies...
Delete Files...
Settings...
History
The History folder contains links to pages you've visited, for quick access to recently viewed pages.
Days to keep pages in history: 20
Clear History
Colors...
Fonts...
Languages...
Accessibility...
OK
Cancel
Apply
Deleting Cookies in internet Explorer

Session Objects

Out of the four techniques for session management, the Session object, represented by the javax.servlet.http.HttpSession interface, is the easiest to use and the most powerful. For each user, the servlet can create an HttpSession object that is associated with that user only and can only be accessed by that particular user. The HttpSession object acts like a Hashtable into which we can store any number of key/object pairs. The HttpSession object is accessible from other servlets in the same application. To retrieve an object previously stored, we need only to pass the key.

An HttpSession object uses a cookie or URL rewriting to send a token to the client. If cookies are used to convey session identifiers, the client browsers are required to accept cookies.

Unlike previous techniques, however, the server does not send any value. What it sends is simply a unique number called the session identifier. This session identifier is used to associate a user with a Session object in the server. Therefore, if there are 10 simultaneous users, 10 Session objects will be created in the server and each user can access only his/her own HttpSession object.

The way an HttpSession object is created for a user and retrieved in the next requests is illustrated in Figure below:

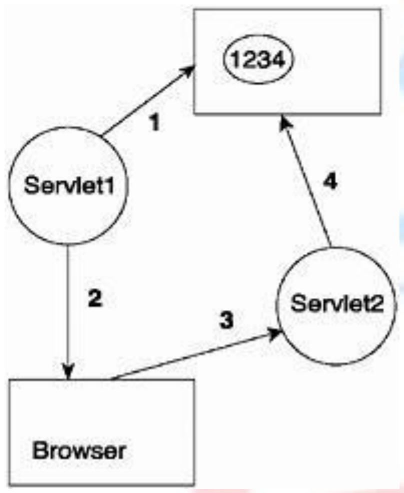


Figure above shows that there are four steps in session tracking using the HttpSession object:

1. An HttpSession object is created by a servlet called Servlet1. A session identifier is generated for this HttpSession object. In this example, the session identifier is 1234, but in reality, the servlet container will generate a longer random number that is guaranteed to be unique. The HttpSession object then is stored in the server and is associated with the

generated session identifier. Also the programmer can store values immediately after creating an HttpSession.

2. In the response, the servlet sends the session identifier to the client browser.

3. When the client browser requests another resource in the same application, such as Servlet2, the session identifier is sent back to the server and passed to Servlet2 in the HttpServletRequest object.

4. For Servlet2 to have access to the HttpSession object for this particular client, it uses the getSession method of the HttpServletRequest interface. This method automatically retrieves the session identifier from the request and obtains the HttpSession object associated with the session identifier.

The getSession method of the HttpServletRequest interface has two overloads. They are as follows:

```
HttpSession getSession()
HttpSession getSession(boolean create)
```

The first overload returns the current session associated with this request, or if the request does not have a session identifier, it creates a new one.

The second overload returns the HttpSession object associated with this request if there is a valid session identifier in the request. If no valid session identifier is found in the request, whether a new HttpSession object is created depends on the create value. If the value is true, a new HttpSession object is created if no valid session identifier is found in the request. Otherwise, the getSession method will return null.

The HttpSession interface [Ref.1]

getAttribute

This method retrieves an attribute from the HttpSession object. The return value is an object of type Object; therefore we may have to downcast the attribute to its original type. To retrieve an attribute, we pass the name associated with the attribute. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.
The signature:

```
public Object getAttribute(String name)
     throws IllegalStateException
```

getAttributeNames

The getAttributeNames method returns a java.util.Enumeration containing all attribute names in the HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.
The signature is as follows:

```
public java.util.Enumeration getAttributeNames()
     throws IllegalStateException
```

getCreationTime

The getCreationTime method returns the time that the HttpSession object was created, in milliseconds since January 1, 1970 00:00:00 GMT. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.
The signature is as follows:

```
public long getCreationTime() throws IllegalStateException
```

getId

The getID method returns the session identifier. The signature for this method is as follows:

```
public String getId()
```

getLastAccessedTime

The getLastAccessedTime method returns the time the HttpSession object was last accessed by the client. The return value is the number of milliseconds lapsed since January 1, 1970 00:00:00 GMT. The following is the method signature:

```
public long getLastAccessedTime()
```

getMaxInactiveInterval

The getMaxInactiveInterval method returns the number of seconds the HttpSession object will be retained by the servlet container after it is last accessed before being removed. The signature of this method is as follows:

```
public int getMaxInactiveInterval()
```

getServletContext

The getServletContext method returns the ServletContext object the HttpSession object belongs to. The signature is as follows:

```
public javax.servlet.ServletContext getServletContext
```

invalidate

The invalidate method invalidates the HttpSession object and unbinds any object bound to it. This method throws an IllegalStateException if this method is called upon an already invalidated HttpSession object. The signature is as follows:

```
public void invalidate() throws IllegalStateException
```

isNew

The isNew method indicates whether the HttpSession object was created with this request and the client has not yet joined the session tracking. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. Its signature is as follows:

```
public boolean isNew() throws IllegalStateException
```

removeAttribute

The removeAttribute method removes an attribute bound to this HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. Its signature is as follows:

```
public void removeAttribute(String name)
     throws IllegalStateException
```

setAttribute

The setAttribute method adds a name/attribute pair to the HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. The method has the following signature:

```
public void setAttribute(String name, Object attribute)
     throws IllegalStateException
```

setMaxInactiveInterval

The setMaxInactiveInterval method sets the number of seconds from the time the HttpSession object is accessed the servlet container will wait before removing the HttpSession object. The signature is as follows:

```
public void setMaxInactiveInterval(int interval)
```

Passing a negative number to this method will make this HttpSession object never expires.

Example:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SessionExample extends HttpServlet {
    public void doGet(   HttpServletRequest request,
                         HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        HttpSession session = request.getSession(true);

        // print session info
        Date created = new Date(session.getCreationTime());
        Date accessed = new Date(session.getLastAccessedTime());
        out.println("ID " + session.getId());
        out.println("Created: " + created);
        out.println("Last Accessed: " + accessed);

        // set session info if needed
        String dataName = request.getParameter("dataName");
        if (dataName != null && dataName.length() > 0) {
            String dataValue = request.getParameter("dataValue");
            session.setAttribute(dataName, dataValue);
        }

        // print session contents
        Enumeration e = session.getAttributeNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = session.getAttribute(name).toString();
            out.println(name + " = " + value);
        }
    }
}
```

References

1. **Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions,**
   First Edition by Budi Kurniawan, 2002, New Riders Publishing
   Chapter 1: The Servlet Technology
   Chapter 2: Inside Servlets
   Chapter 3: Writing Servlet Applications
   Chapter 5: Session Management
   (Most of the data is referred from this book)

2. **Java Servlet Programming,**
   Second Edition by Jason Hunter, William Crawford, 2001, O'Reilly

3. **Java the Complete Reference**,
   Seventh Edition by Herbert Schildt, 2001 Osborne McGraw Hill
   Chapter 31: Servlets
