

MODULE-4

Background

In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content to a user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition, the CGI programs were not platform-independent. Therefore, other techniques were introduced. Among these are servlets.

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

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init()**, **service()**, and **destroy()**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init()** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service()** method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The **service()** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the **destroy()** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To create servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Tomcat. Glassfish is from Oracle and is provided by the Java EE SDK. It is supported by NetBeans. Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse. The examples and descriptions in this chapter use Tomcat for reasons that will soon be apparent.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

Tomcat is used in this chapter because, in the opinion of this author, it makes it relatively easy to run the example servlets using only command-line tools and a text editor. It is also widely available in various programming environments. Furthermore, since only command-line tools are used, you don't need to download and install an IDE just to experiment with servlets. Understand, however, that even if you are developing in an environment that uses Glassfish, the concepts presented here still apply. It is just that the mechanics of preparing a servlet for testing will be slightly different.

REMEMBER The instructions for developing and deploying servlets in this chapter are based on Tomcat and use only command-line tools. If you are using an IDE and different servlet container/server, consult the documentation for your environment.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, several versions of Tomcat are available. The instructions that follow use 7.0.47. You can download Tomcat from **tomcat.apache.org**. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 7.0.47 was unpacked from the root directly, the default location is

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\
```

This is the location assumed by the examples in this book. If you load Tomcat in a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable **JAVA_HOME** to the top-level directory in which the Java Development Kit is installed.

NOTE All of the directories shown in this section assume Tomcat 7.0.47. If you install a different version of Tomcat, then you will need to adjust the directory names and paths to match those used by the version you installed.

Once installed, you start Tomcat by selecting **startup.bat** from the **bin** directly under the **apache-tomcat-7.0.47** directory. To stop Tomcat, execute **shutdown.bat**, also in the **bin** directory.

The classes and interfaces needed to build servlets are contained in **servlet-api.jar**, which is in the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib
```

To make **servlet-api.jar** accessible, update your **CLASSPATH** environment variable so that it includes

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib\servlet-api.jar
```

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib\servlet-api.jar"
```

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat's **webapps** directory and entering its name into a **web.xml** file. To keep things simple, the examples in this chapter use the directory and **web.xml** file that Tomcat supplies for its own example servlets. This way, you won't have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet's class file into the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\examples\WEB-INF\classes
```

Next, add the servlet's name and mapping to the **web.xml** file in the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\examples\WEB-INF
```

For instance, assuming the first example, called **HelloServlet**, you will add the following lines in the section that defines the servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servlet mappings:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.

2. Start Tomcat.
3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named **HelloServlet.java** that contains the following program:

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

public class HelloServlet extends GenericServlet {

    public void service(ServletRequest request,
        ServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello!");
        pw.close();
    }
}
```

Let's look closely at this program. First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the creation of a servlet. For example, it provides versions of **init()** and **destroy()**, which may be used as is. You need supply only the **service()** method.

Inside **HelloServlet**, the **service()** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client. Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType()** establishes the MIME type of the HTTP response. In this program, the MIME type is text/html. This indicates that the browser should interpret the content as HTML source code.

Next, the **getWriter()** method obtains a **PrintWriter**. Anything written to this stream is sent to the client as part of the HTTP response. Then **println()** is used to write some simple HTML source code as the HTTP response.

Compile this source code and place the **HelloServlet.class** file in the proper Tomcat directory as described in the previous section. Also, add **HelloServlet** to the **web.xml** file, as described earlier.

Start Tomcat

Start Tomcat as explained earlier. Tomcat must be running before you try to execute a servlet.

Start a Web Browser and Request the Servlet

Start a web browser and enter the URL shown here:

```
http://localhost:8080/examples/servlets/servlet/HelloServlet
```

Alternatively, you may enter the URL shown here:

```
http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet
```

This can be done because 127.0.0.1 is defined as the IP address of the local machine.

You will observe the output of the servlet in the browser display area. It will contain the string **Hello!** in bold type.

The Servlet API

Two packages contain the classes and interfaces that are required to build the servlets described in this chapter. These are **javax.servlet** and **javax.servlet.http**. They constitute the core of the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also provided by Java EE.

The Servlet API has been in a process of ongoing development and enhancement. The current servlet specification is version 3.1. However, because changes happen fast in the world of Java, you will want to check for any additions or alterations. This chapter discusses the core of the Servlet API, which will be available to most readers and works with all modern versions of the servlet specification.

The javax.servlet Package

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

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

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

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

Let us examine these interfaces and classes in more detail.

The Servlet Interface

All servlets must implement the **Servlet** interface. It declares the **init()**, **service()**, and **destroy()** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by **Servlet** are shown in Table 38-1.

The **init()**, **service()**, and **destroy()** methods are the life cycle methods of the servlet. These are invoked by the server. The **getServletConfig()** method is called by the servlet to obtain initialization parameters. A servlet developer overrides the **getServletInfo()** method to provide a string with useful information (for example, the version number). This method is also invoked by the server.

Method	Description
<code>void destroy()</code>	Called when the servlet is unloaded.
<code>ServletConfig getServletConfig()</code>	Returns a ServletConfig object that contains any initialization parameters.
<code>String getServletInfo()</code>	Returns a string describing the servlet.
<code>void init(ServletConfig sc) throws ServletException</code>	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from <i>sc</i> . A ServletException should be thrown if the servlet cannot be initialized.
<code>void service(ServletRequest req, ServletResponse res) throws ServletException, IOException</code>	Called to process a request from a client. The request from the client can be read from <i>req</i> . The response to the client can be written to <i>res</i> . An exception is generated if a servlet or IO problem occurs.

Table 38-1 The Methods Defined by **Servlet**

The ServletConfig Interface

The **ServletConfig** interface allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:

Method	Description
<code>ServletContext getServletContext()</code>	Returns the context for this servlet.
<code>String getInitParameter(String <i>param</i>)</code>	Returns the value of the initialization parameter named <i>param</i> .
<code>Enumeration<String> getInitParameterNames()</code>	Returns an enumeration of all initialization parameter names.
<code>String getServletName()</code>	Returns the name of the invoking servlet.

The ServletContext Interface

The **ServletContext** interface enables servlets to obtain information about their environment. Several of its methods are summarized in Table 38-2.

The ServletRequest Interface

The **ServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are summarized in Table 38-3.

The ServletResponse Interface

The **ServletResponse** interface enables a servlet to formulate a response for a client. Several of its methods are summarized in Table 38-4.

Method	Description
<code>Object getAttribute(String <i>attr</i>)</code>	Returns the value of the server attribute named <i>attr</i> .
<code>String getMimeType(String <i>file</i>)</code>	Returns the MIME type of <i>file</i> .
<code>String getRealPath(String <i>vpath</i>)</code>	Returns the real (i.e., absolute) path that corresponds to the relative path <i>vpath</i> .
<code>String getServerInfo()</code>	Returns information about the server.
<code>void log(String <i>s</i>)</code>	Writes <i>s</i> to the servlet log.
<code>void log(String <i>s</i>, Throwable <i>e</i>)</code>	Writes <i>s</i> and the stack trace for <i>e</i> to the servlet log.
<code>void setAttribute(String <i>attr</i>, Object <i>val</i>)</code>	Sets the attribute specified by <i>attr</i> to the value passed in <i>val</i> .

Table 38-2 Various Methods Defined by **ServletContext**

Method	Description
Object <code>getAttribute(String attr)</code>	Returns the value of the attribute named <i>attr</i> .
String <code>getCharacterEncoding()</code>	Returns the character encoding of the request.
int <code>getContentLength()</code>	Returns the size of the request. The value <code>-1</code> is returned if the size is unavailable.
String <code>getContentType()</code>	Returns the type of the request. A null value is returned if the type cannot be determined.
ServletInputStream <code>getInputStream()</code> throws <code>IOException</code>	Returns a ServletInputStream that can be used to read binary data from the request. An IllegalStateException is thrown if <code>getReader()</code> has been previously invoked on this object.
String <code>getParameter(String pname)</code>	Returns the value of the parameter named <i>pname</i> .
Enumeration<String> <code>getParameterNames()</code>	Returns an enumeration of the parameter names for this request.
String[] <code>getParameterValues(String name)</code>	Returns an array containing values associated with the parameter specified by <i>name</i> .
String <code>getProtocol()</code>	Returns a description of the protocol.
BufferedReader <code>getReader()</code> throws <code>IOException</code>	Returns a buffered reader that can be used to read text from the request. An IllegalStateException is thrown if <code>getInputStream()</code> has been previously invoked on this object.
String <code>getRemoteAddr()</code>	Returns the string equivalent of the client IP address.
String <code>getRemoteHost()</code>	Returns the string equivalent of the client host name.
String <code>getScheme()</code>	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String <code>getServerName()</code>	Returns the name of the server.
int <code>getServerPort()</code>	Returns the port number.

Table 38-3 Various Methods Defined by **ServletRequest**

Method	Description
String <code>getCharacterEncoding()</code>	Returns the character encoding for the response.
ServletOutputStream <code>getOutputStream()</code> throws <code>IOException</code>	Returns a ServletOutputStream that can be used to write binary data to the response. An IllegalStateException is thrown if <code>getWriter()</code> has been previously invoked on this object.
PrintWriter <code>getWriter()</code> throws <code>IOException</code>	Returns a PrintWriter that can be used to write character data to the response. An IllegalStateException is thrown if <code>getOutputStream()</code> has been previously invoked on this object.
void <code>setContentLength(int size)</code>	Sets the content length for the response to <i>size</i> .
void <code>setContentType(String type)</code>	Sets the content type for the response to <i>type</i> .

Table 38-4 Various Methods Defined by **ServletResponse**

The GenericServlet Class

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

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

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

The ServletInputStream Class

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

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

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

The ServletOutputStream Class

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

The Servlet Exception Classes

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

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
```

```
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/
            servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type="text" name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type="text" name="p" size="25" value=""></td>
</tr>
</table>
<input type="submit" value="Submit">
</body>
</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing. The **service()** method is overridden to process client requests. The **getParameterNames()** method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the **getParameter()** method.

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

public class PostParametersServlet
extends GenericServlet {

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

        // Get print writer.
        PrintWriter pw = response.getWriter();

        // Get enumeration of parameter names.
        Enumeration e = request.getParameterNames();

        // Display parameter names and values.
        while(e.hasMoreElements()) {
            String pname = (String)e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

After following these steps, the browser will display a response that is dynamically generated by the servlet.

The **javax.servlet.http** Package

The preceding examples have used the classes and interfaces defined in **javax.servlet**, such as **ServletRequest**, **ServletResponse**, and **GenericServlet**, to illustrate the basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in **javax.servlet.http**. As you will see, its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the interfaces used in this chapter:

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

The following table summarizes the classes used in this chapter. The most important of these is **HttpServlet**. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.

The **HttpServletRequest** Interface

The **HttpServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are shown in Table 38-5.

The **HttpServletResponse** Interface

The **HttpServletResponse** interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, **SC_OK** indicates that the HTTP

Method	Description
String getAuthType()	Returns authentication scheme.
Cookie[] getCookies()	Returns an array of the cookies in this request.
long getDateHeader(String <i>field</i>)	Returns the value of the date header field named <i>field</i> .
String getHeader(String <i>field</i>)	Returns the value of the header field named <i>field</i> .
Enumeration<String> getHeaderNames()	Returns an enumeration of the header names.
int getIntHeader(String <i>field</i>)	Returns the int equivalent of the header field named <i>field</i> .
String getMethod()	Returns the HTTP method for this request.
String getPathInfo()	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated()	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString()	Returns any query string in the URL.
String getRemoteUser()	Returns the name of the user who issued this request.
String getRequestedSessionId()	Returns the ID of the session.
String getRequestURI()	Returns the URI.
StringBuffer getRequestURL()	Returns the URL.
String getServletPath()	Returns that part of the URL that identifies the servlet.
HttpSession getSession()	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean <i>new</i>)	If <i>new</i> is true and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie()	Returns true if a cookie contains the session ID. Otherwise, returns false .
boolean isRequestedSessionIdFromURL()	Returns true if the URL contains the session ID. Otherwise, returns false .
boolean isRequestedSessionIdValid()	Returns true if the requested session ID is valid in the current session context.

Table 38-5 Various Methods Defined by **HttpServletRequest**

request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available. Several methods of this interface are summarized in Table 38-6.

The HttpSession Interface

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table 38-7. All of these methods throw an **IllegalStateException** if the session has already been invalidated.

Method	Description
<code>void addCookie(Cookie cookie)</code>	Adds <i>cookie</i> to the HTTP response.
<code>boolean containsHeader(String field)</code>	Returns true if the HTTP response header contains a field named <i>field</i> .
<code>String encodeURL(String url)</code>	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs generated by a servlet should be processed by this method.
<code>String encodeRedirectURL(String url)</code>	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs passed to sendRedirect() should be processed by this method.
<code>void sendError(int c)</code> throws <code>IOException</code>	Sends the error code <i>c</i> to the client.
<code>void sendError(int c, String s)</code> throws <code>IOException</code>	Sends the error code <i>c</i> and message <i>s</i> to the client.
<code>void sendRedirect(String url)</code> throws <code>IOException</code>	Redirects the client to <i>url</i> .
<code>void setDateHeader(String field, long msec)</code>	Adds <i>field</i> to the header with date value equal to <i>msec</i> (milliseconds since midnight, January 1, 1970, GMT).
<code>void setHeader(String field, String value)</code>	Adds <i>field</i> to the header with value equal to <i>value</i> .
<code>void setIntHeader(String field, int value)</code>	Adds <i>field</i> to the header with value equal to <i>value</i> .
<code>void setStatus(int code)</code>	Sets the status code for this response to <i>code</i> .

Table 38-6 Various Methods Defined by **HttpServletResponse**

The Cookie Class

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store.

A servlet can write a cookie to a user's machine via the **addCookie()** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some of the information that can be saved for each cookie includes the following:

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

Method	Description
Object <code>getAttribute(String attr)</code>	Returns the value associated with the name passed in <i>attr</i> . Returns null if <i>attr</i> is not found.
Enumeration<String> <code>getAttributeNames()</code>	Returns an enumeration of the attribute names associated with the session.
long <code>getCreationTime()</code>	Returns the creation time (in milliseconds since midnight, January 1, 1970, GMT) of the invoking session.
String <code>getId()</code>	Returns the session ID.
long <code>getLastAccessedTime()</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request on the invoking session.
void <code>invalidate()</code>	Invalidates this session and removes it from the context.
boolean <code>isNew()</code>	Returns true if the server created the session and it has not yet been accessed by the client.
void <code>removeAttribute(String attr)</code>	Removes the attribute specified by <i>attr</i> from the session.
void <code>setAttribute(String attr, Object val)</code>	Associates the value passed in <i>val</i> with the attribute name passed in <i>attr</i> .

Table 38-7 Various Methods Defined by **HttpSession**

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, it is not.

There is one constructor for **Cookie**. It has the signature shown here:

```
Cookie(String name, String value)
```

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the **Cookie** class are summarized in Table 38-8.

The **HttpServlet** Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the **HttpServlet** class are summarized in Table 38-9.

Method	Description
Object clone()	Returns a copy of this object.
String getComment()	Returns the comment.
String getDomain()	Returns the domain.
int getMaxAge()	Returns the maximum age (in seconds).
String getName()	Returns the name.
String getPath()	Returns the path.
boolean getSecure()	Returns true if the cookie is secure. Otherwise, returns false .
String getValue()	Returns the value.
int getVersion()	Returns the version.
boolean isHttpOnly()	Returns true if the cookie has the HttpOnly attribute.
void setComment(String <i>c</i>)	Sets the comment to <i>c</i> .
void setDomain(String <i>d</i>)	Sets the domain to <i>d</i> .
void setHttpOnly(boolean <i>httpOnly</i>)	If <i>httpOnly</i> is true , then the HttpOnly attribute is added to the cookie. If <i>httpOnly</i> is false , the HttpOnly attribute is removed.
void setMaxAge(int <i>secs</i>)	Sets the maximum age of the cookie to <i>secs</i> . This is the number of seconds after which the cookie is deleted.
void setPath(String <i>p</i>)	Sets the path to <i>p</i> .
void setSecure(boolean <i>secure</i>)	Sets the security flag to <i>secure</i> .
void setValue(String <i>v</i>)	Sets the value to <i>v</i> .
void setVersion(int <i>v</i>)	Sets the version to <i>v</i> .

Table 38-8 The Methods Defined by **Cookie**

Method	Description
void doDelete(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP OPTIONS request.

Table 38-9 The Methods Defined by **HttpServlet**

Method	Description
void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Handles an HTTP POST request.
void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Handles an HTTP PUT request.
void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Handles an HTTP TRACE request.
long getLastModified(HttpServletRequest req)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Table 38-9 The Methods Defined by **HttpServlet** (*continued*)

Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete()**, **doGet()**, **doHead()**, **doOptions()**, **doPost()**, **doPut()**, and **doTrace()**. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.html**, and a servlet is defined in **ColorGetServlet.java**. The HTML source code for **ColorGet.html** is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form name="Form1"
      action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
```

```

<option value="Blue">Blue</option>
</select>
<br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>

```

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

```

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

public class ColorGetServlet extends HttpServlet {

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

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}

```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

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

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

```
http://localhost:8080/examples/servlets/servlet/ColorGetServlet?color=Red
```

The characters to the right of the question mark are known as the *query string*.

Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.html**, and a servlet is defined in **ColorPostServlet.java**.

The HTML source code for **ColorPost.html** is shown in the following listing. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/servlet/ColorPostServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The **doPost()** method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the **getParameter()** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

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

public class ColorPostServlet extends HttpServlet {

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

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

Compile the servlet and perform the same steps as described in the previous section to test it.

NOTE Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is `http://localhost:8080/examples/servlets/servlet/ColorPostServlet`. The parameter names and values are sent in the body of the HTTP request.

Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

File	Description
AddCookie.html	Allows a user to specify a value for the cookie named MyCookie .
AddCookieServlet.java	Processes the submission of AddCookie.html .
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.html** is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type="text" name="data" size=25 value="">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie()** method. A feedback message is then written to the browser.

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

public class AddCookieServlet extends HttpServlet {

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



```
// Get parameter from HTTP request.
String data = request.getParameter("data");

// Create cookie.
Cookie cookie = new Cookie("MyCookie", data);

// Add cookie to HTTP response.
response.addCookie(cookie);

// Write output to browser.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>MyCookie has been set to");
pw.println(data);
pw.close();
}
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies()** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName()** and **getValue()** methods are called to obtain this information.

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

public class GetCookiesServlet extends HttpServlet {

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

        // Get cookies from header of HTTP request.
        Cookie[] cookies = request.getCookies();

        // Display these cookies.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = " + name +
                "; value = " + value);
        }
        pw.close();
    }
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

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

After completing these steps, you will observe that a feedback message is displayed by the browser.

Next, request the following URL via the browser:

```
http://localhost:8080/examples/servlets/servlet/GetCookiesServlet
```

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookie via the **setMaxAge()** method of **Cookie**. Therefore, the cookie expires when the browser session ends. You can experiment by using **setMaxAge()** and observe that the cookie is then saved on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the **getSession()** method of **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects. The **setAttribute()**, **getAttribute()**, **getAttributeNames()**, and **removeAttribute()** methods of **HttpSession** manage these bindings. Session state is shared by all servlets that are associated with a client.

The following servlet illustrates how to use session state. The **getSession()** method gets the current session. A new session is created if one does not already exist. The **getAttribute()** method is called to obtain the object that is bound to the name "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The **setAttribute()** method is called to bind the name "date" to this object.

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

public class DateServlet extends HttpServlet {
```

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

    // Get the HttpSession object.
    HttpSession hs = request.getSession(true);

    // Get writer.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>");

    // Display date/time of last access.
    Date date = (Date)hs.getAttribute("date");
    if(date != null) {
        pw.print("Last access: " + date + "<br>");
    }

    // Display current date/time.
    date = new Date();
    hs.setAttribute("date", date);
    pw.println("Current date: " + date);
}
}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.