

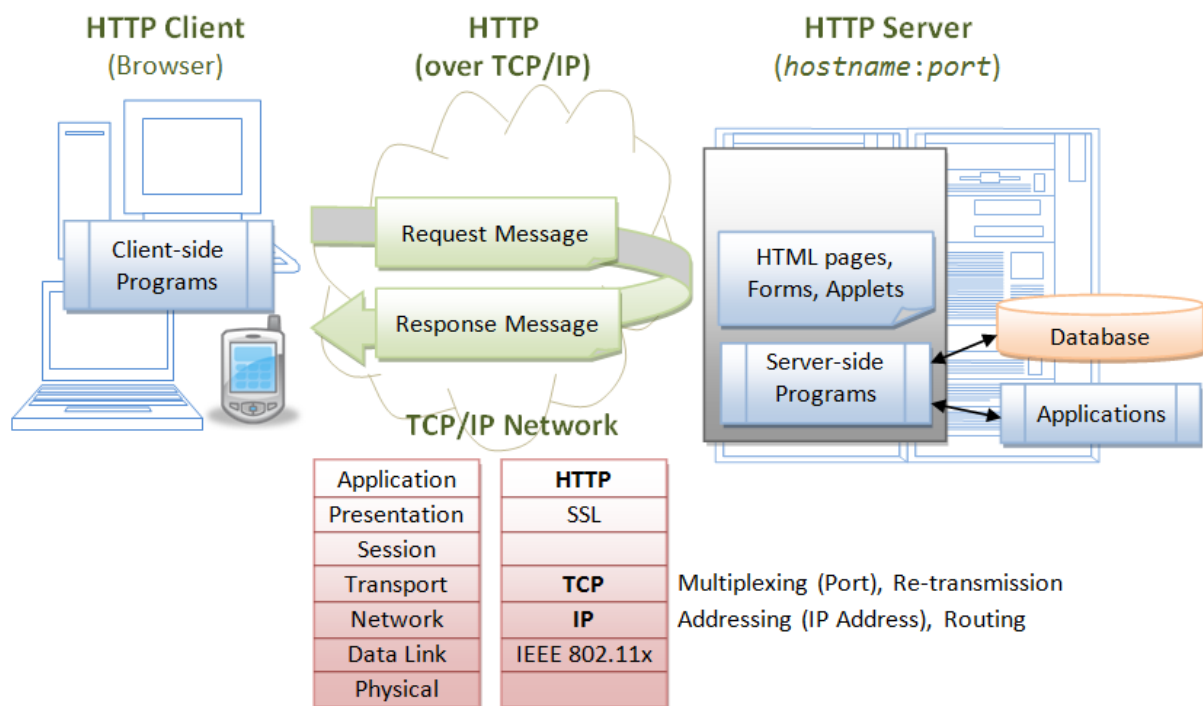
1. INTRODUCCIÓN	2
2. CICLO DE VIDA.....	3
3. VISUALIZACIÓN DEL TEXTO “HOLA MUNDO”	5
4. PROCESAMIENTO DE DATOS DE UN FORMULARIO HTML	8
4.1. CONSTRUCCIÓN DE UN FORMULARIO HTML	8
4.2. LECTURA DE LOS DATOS DE UN FORMULARIO HTML.....	10
5. MENSAJE DE PETICIÓN DEL CLIENTE.....	16
5.1. CABECERAS DE PETICIÓN HTTP	16
5.2. INTERFACES ASOCIADAS AL MENSAJE DE PETICIÓN	17
5.3. OBTENCIÓN DE LA CABECERA DEL MENSAJE DE PETICIÓN	18
6. MENSAJE DE RESPUESTA DEL SERVIDOR	20
6.1. CABECERAS DE RESPUESTA HTTP	20
6.2. INTERFACES ASOCIADAS AL MENSAJE DE RESPUESTA	21
6.3. ESTADO PERSISTENTE DEL SERVLET Y PÁGINAS DE RECARGA AUTOMÁTICA ..	21
7. CÓDIGOS DE ESTADO HTTP	27
7.1. DESCRIPCIÓN DE LOS CÓDIGOS DE ESTADO HTTP	27
7.2. REDIRECCIONES Y ERRORES	31
8. MANEJO DE COOKIES.....	35
8.1. DETECCIÓN DE VISITANTES DE PÁGINAS WEB.....	35
8.2. COOKIES DE SESIÓN Y COOKIES PERSISTENTES.....	36
8.3. MODIFICACIÓN DE COOKIES	39
9. RASTREO DE SESIONES.....	41
9.1. RASTREO DE SESIONES CON SERVLETS	41
9.2. SESIONES DE NAVEGADOR Y SESIONES DE SERVIDOR	43
9.3. CIFRADO DE LAS URL ENVIADAS AL CLIENTE	43
9.4. CREACIÓN DE UNA LISTA DE DATOS DEL USUARIO.....	44
9.5. TIENDA VIRTUAL CON CARRO DE COMPRA.....	46
9.5.1. <i>Generación del Catálogo de Artículos de Venta</i>	46
9.5.2. <i>Generación del Pedido del Cliente</i>	49

1. INTRODUCCIÓN

Los **servlets** de Java son programas del lado servidor (se ejecutan dentro de un servidor web) que manejan las peticiones de los clientes y que devuelven una respuesta personalizada o dinámica para cada solicitud. La respuesta dinámica se puede basar en:

- Las entradas del usuario (búsquedas, compras online, transacciones online) con datos recuperados de bases de datos o de otras aplicaciones.
- Datos sensibles a fechas y horas (como noticias y precios).

Los *servlets* de Java normalmente se ejecutan sobre **HTTP (Hypertext Transfer Protocol)**, que es un protocolo de petición-respuesta asimétrico. El cliente envía un mensaje de petición al servidor y el servidor devuelve un mensaje de respuesta al cliente:



Los *servlets* de Java son la tecnología base para la programación web en entorno servidor con Java. Se deben entender bien a fondo su funcionamiento antes de iniciarse en otras tecnologías Java como JSP (*Java Server Pages*) y JSF (*Java Server Faces*).

2. CICLO DE VIDA

El **ciclo de vida** de un *servlet* se define como el proceso completo desde su creación hasta su destrucción, pasando por los siguientes pasos:

- 1) El *servlet* es inicializado mediante la invocación al método *init()*.
- 2) El *servlet* invoca al método *service()* para procesar la petición de un cliente.
- 3) El *servlet* es terminado mediante la invocación al método *destroy()*.
- 4) El recolector de basura de la JVM (*Java Virtual Machine*) libera los recursos usados por el *servlet*.

El método **init()** sólo se llama una vez, cuando el *servlet* se crea. No es invocado para las peticiones de los clientes. Por tanto, se usa para inicializaciones de datos que se tengan que realizar una única vez y que se necesiten durante el ciclo de vida del *servlet*.

Normalmente, el *servlet* se crea cuando un usuario invoca su URL correspondiente por primera vez, pero también se puede especificar que el *servlet* se cargue cuando el servidor se arranque la primera vez.

Cuando un usuario invoca un *servlet*, se crea una única instancia suya y cada petición de usuario se gestiona con un nuevo *thread* que llama al método apropiado *doGet()* o *doPost()*.

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

El método **service()** es el método principal para realizar tareas. El contenedor de *servlets* (servidor web) llama a este método para manejar la petición procedente de un cliente (navegador) y para devolverle una respuesta formateada al cliente.

Cada vez que el servidor recibe una petición de un *servlet*, el servidor crea un nuevo *thread* e invoca al método *service()*, que comprobará el tipo de petición HTTP (GET, POST, PUT, DELETE) y llamará al método apropiado (*doGet()*, *doPost()*, *doPut()*, *doDelete()*). Así, sólo será necesario sobrescribir estos métodos dependiendo del tipo de petición que se haya recibido del cliente.

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

El método **doGet()** se utiliza para procesar las peticiones GET procedentes de peticiones normales de URL o de formularios HTML que no tienen asociado un método.

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

El método **doPost()** se utiliza para procesar las peticiones POST procedentes de formularios HTML que explícitamente indican el método POST.

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

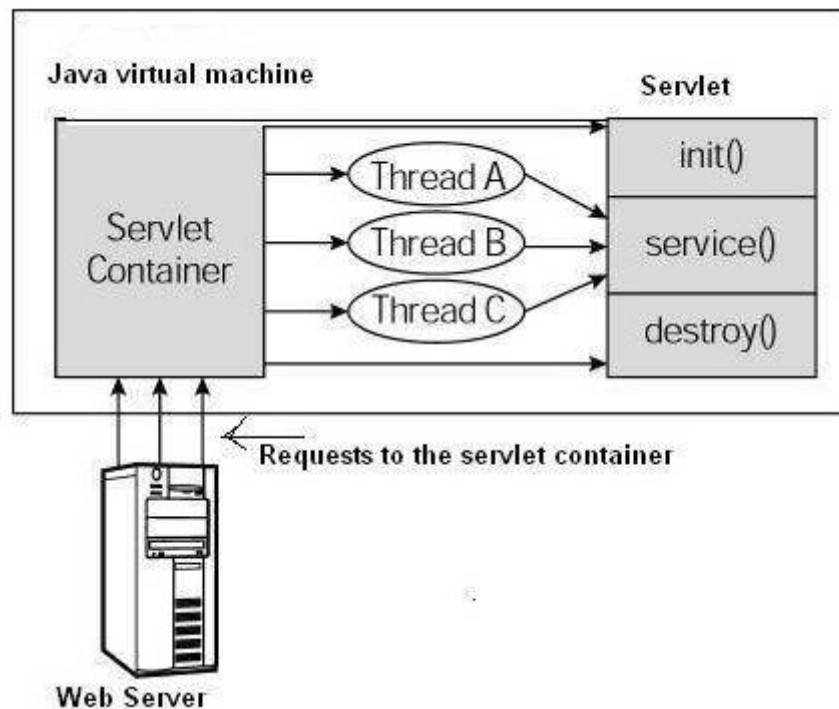
El método **destroy()** es invocado sólo una vez al final del ciclo de vida de un *servlet*. Este método le proporciona al *servlet* oportunidad de cerrar conexiones de bases de datos, detener hilos de ejecución en segundo plano (*background*), escribir listas de *cookies* o contadores de visitas a disco, y realizar otras actividades de limpieza.

Después de llamar al método *destroy()*, el objeto *servlet* asociado se marca para la recolección de basura.

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

Un escenario típico del ciclo de vida de un *servlet* puede ser el siguiente:

- 1) Primero, la petición HTTP procedente del cliente se delega al contenedor de *servlets*.
- 2) El contenedor de *servlets* carga el *servlet* asociado antes de invocar al método *service()*.
- 3) Entonces, el contenedor de *servlets* puede manejar múltiples peticiones mediante varios *threads*. Cada *thread* o hilo de ejecución ejecuta el método *service()* de una única instancia del *servlet*.

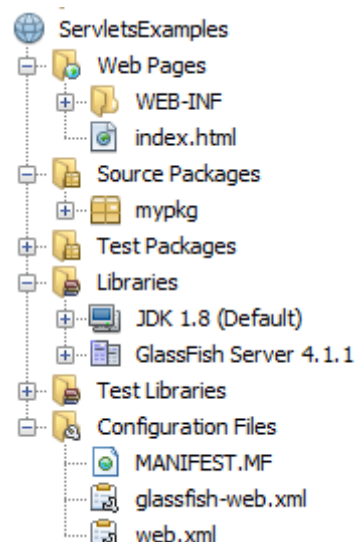


3. VISUALIZACIÓN DEL TEXTO “HOLA MUNDO”

Una **aplicación web** (a veces denominada *webapp*) está compuesta por un conjunto de recursos, tales como archivos HTML, CSS, JavaScript, imágenes, programas y bibliotecas.

Una aplicación web Java tiene una jerarquía estándar de directorios para almacenar los diversos tipos de recursos. El IDE de *NetBeans* transforma esta vista de archivos en una vista de proyectos con las siguientes carpetas fundamentales:

- **Web Pages.** Incluye el contenido estático de la aplicación web, como páginas HTML estáticas, páginas JSP, hojas de estilo CSS, imágenes, etc.
- **Source Packages.** Incluye el código fuente Java estructurado en paquetes.
- **Libraries.** Incluye las librerías de terceras partes (del JDK, del servidor usado, *GlassFish* o *Apache Tomcat*) que se emplean en la aplicación web.
- **Configuration Files.** Incluye los archivos de configuración de la aplicación web, como varios descriptores de despliegue de la aplicación: *web.xml* es el estándar de Java EE (independiente del servidor) y *glassfish-web.xml* es específico para el servidor de aplicaciones *GlassFish*.



Los *servlets* son programas Java que se guardan dentro de un paquete Java, que se compilan en ficheros `.class` y que se ejecutan dentro de un servidor HTTP. El usuario puede invocar un *servlet* concreto escribiendo una URL específica en el navegador web (cliente HTTP).

El siguiente *servlet* **HelloServlet.java** muestra en una página HTML estática el texto clásico “Hola, Mundo”, junto con información del mensaje de petición del cliente y un número aleatorio generado entre 0 y 1.

```
package mypkg;

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

public class HelloServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        // Set the response message's MIME type
        response.setContentType("text/html;charset=UTF-8");
        // Allocate a output writer to write the response message into the network socket
        PrintWriter out = response.getWriter();
        // Write the response message, in an HTML page
        try {
            out.println("<!DOCTYPE html>");
            out.println("<html><head>");
            out.println("<meta http-equiv='Content-Type' content='text/html; charset=UTF-8'>");
            out.println("<title>Hello, World</title></head>");
            out.println("<body>");
            out.println("<h1>Hello, World!</h1>"); // says Hello
            // Echo client's request information
            out.println("<p>Request URI: " + request.getRequestURI() + "</p>");
            out.println("<p>Protocol: " + request.getProtocol() + "</p>");
            out.println("<p>Path Info: " + request.getPathInfo() + "</p>");
            out.println("<p>Remote Address: " + request.getRemoteAddr() + "</p>");
            // Generate a random number upon each request
            out.println("<p>Random Number: <strong>" + Math.random() + "</strong></p>");
            out.println("</body>");
            out.println("</html>");
        }
        finally {
            // Always close the output writer
            out.close();
        }
    }
}
```

El *servlet* no se codifica desde cero, sino que se crea mediante herencia desde la clase **javax.servlet.http.HttpServlet**.

El *servlet* se invoca como respuesta a una petición URL de un cliente. Normalmente, el cliente realiza una petición HTTP, el servidor redirige el mensaje al *servlet* para su procesamiento y el *servlet* devuelve un mensaje de respuesta al cliente.

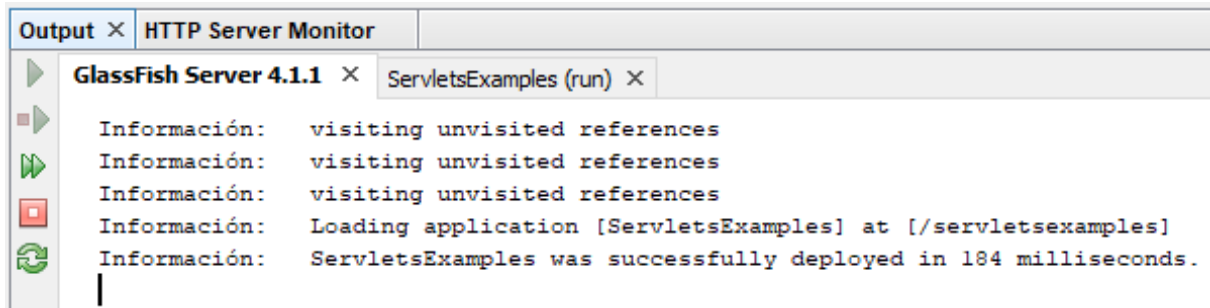
Cualquier petición HTTP puede utilizar los modos GET o POST, que se procesan con los métodos **doGet()** o **doPost()** del *servlet*, respectivamente.

En este ejemplo, se ha sobrescrito el método **doGet()**, que se ejecutará como respuesta a una petición HTTP de tipo GET, realizada por un usuario en una URL. El método **doGet()** tiene dos argumentos:

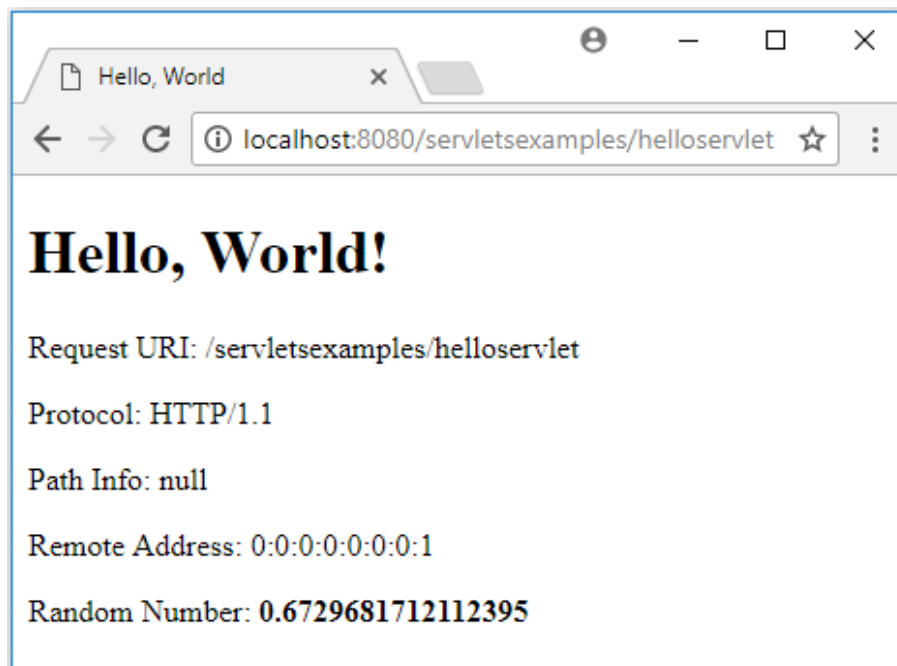
- Un objeto **HttpServletRequest**, que se utiliza para obtener las cabeceras de la petición HTTP y datos de formularios.
- Un objeto **HttpServletResponse**, que se utiliza para establecer las cabeceras de la respuesta HTTP y el cuerpo del mensaje de respuesta.

Antes de visualizar la página HTML, se debe establecer el tipo MIME del mensaje de respuesta a `text/html`, para que el cliente pueda visualizar correctamente los datos recibidos. Otros tipos MIME son `text/plain`, `image/jpeg`, `video/mpeg` y `application/xml`.

Antes de ejecutar el *servlet*, hay que iniciar el servidor de aplicaciones (*GlassFish* o *Apache Tomcat*) y asegurarse de que el contexto web del proyecto se ha desplegado correctamente. Para ello, se debe mirar los mensajes de la consola del servidor:



A continuación, para ejecutar el *servlet* en el servidor y obtener una respuesta, hay que utilizar un navegador web (Firefox, Chrome, etc.) y navegar a la URL del *servlet*:



4. PROCESAMIENTO DE DATOS DE UN FORMULARIO HTML

Una de las principales motivaciones para generar páginas web dinámicas es que el resultado devuelto por el servidor pueda estar basado en los datos proporcionados por el usuario.

4.1. CONSTRUCCIÓN DE UN FORMULARIO HTML

HTML proporciona una etiqueta `<form>...</form>`, que se puede usar para construir un **formulario de entrada de datos** del usuario con diversos elementos (campos de texto, menús desplegables, opción múltiple, opción única y botones). Esto permite a los usuarios interactuar con el servidor web para enviarle datos.

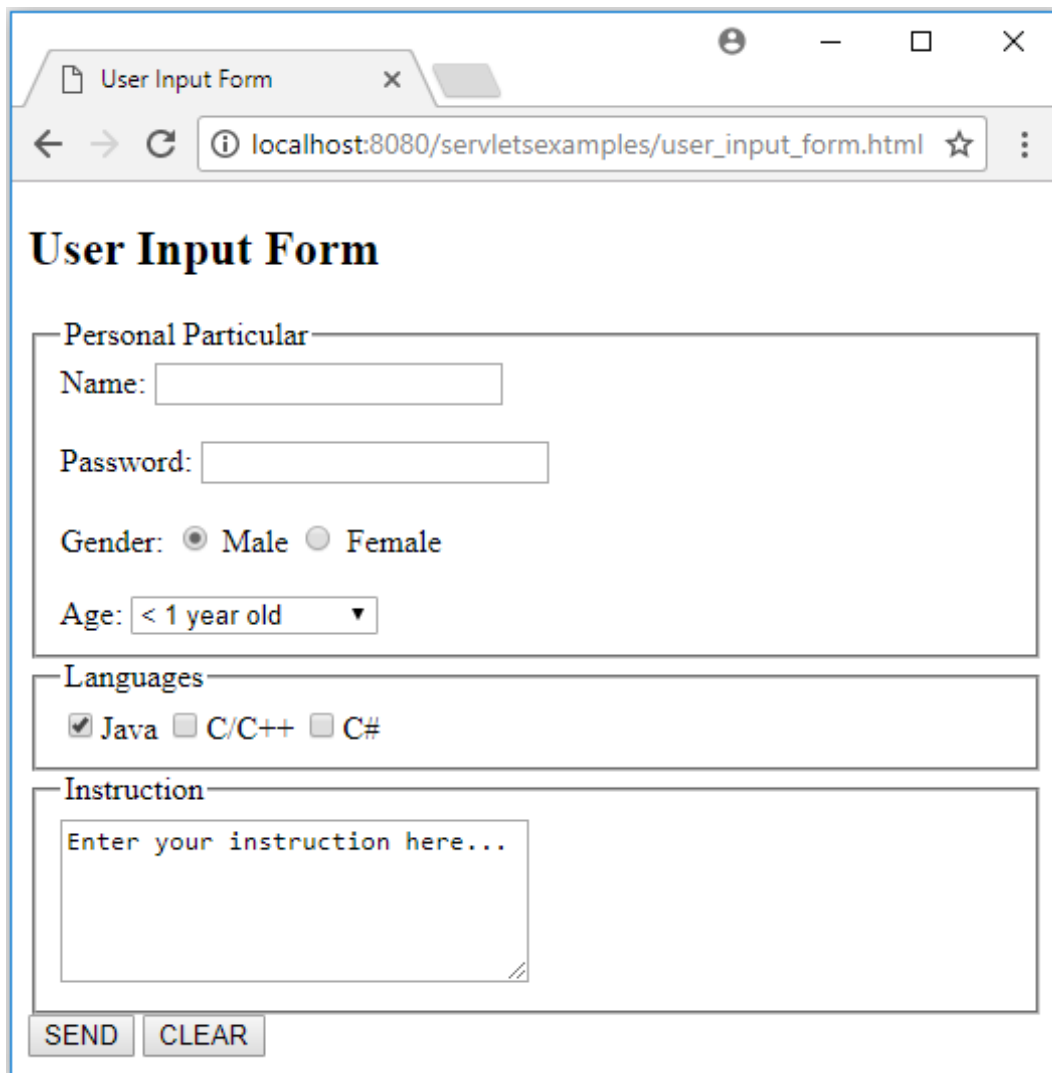
El siguiente formulario HTML se puede usar para recoger datos del usuario:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv='Content-Type' content='text/html; charset=UTF-8'>
    <title>User Input Form</title>
  </head>
  <body>
    <h2>User Input Form</h2>
    <form method="get" action="echo">
      <fieldset>
        <legend>Personal Particular</legend>
        Name: <input type="text" name="username" /><br /><br />
        Password: <input type="password" name="password" /><br /><br />
        Gender: <input type="radio" name="gender" value="m" checked />
        Male <input type="radio" name="gender" value="f" />
        Female<br /><br />
        Age: <select name = "age">
          <option value="1">&lt; 1 year old</option>
          <option value="99">1 to 99 years old</option>
          <option value="100">&gt; 99 years old</option>
        </select>
      </fieldset>
      <fieldset>
        <legend>Languages</legend>
        <input type="checkbox" name="language" value="java" checked />Java
        <input type="checkbox" name="language" value="c" />C/C++
        <input type="checkbox" name="language" value="cs" />C#
      </fieldset>
      <fieldset>
        <legend>Instruction</legend>
        <textarea rows="5" cols="30" name="instruction">Enter your instruction here...
      </fieldset>
      <input type="hidden" name="secret" value="888" />
      <input type="submit" value="SEND" />
      <input type="reset" value="CLEAR" />
    </form>
  </body>
</html>
```

Cada elemento de entrada tiene un atributo `name` y un atributo opcional `value`. Si el usuario introduce un dato en un elemento del formulario, el par `name=value` correspondiente se enviará al servidor para ser procesado.

Este formulario se puede almacenar en el proyecto de una aplicación web como una página HTML denominada **user_input_form.html** y será accesible desde cualquier navegador web mediante la siguiente ruta URL:

`http://localhost:8080/servletsexamples/user_input_form.html`



Por ejemplo, si se introducen algunos datos y se pulsa el botón **SEND**, se obtendrá el error 404 (*page not found*), porque aún falta escribir un *servlet* que procese los datos enviados. Pero la URL destino puede quedar así:

```
http://localhost:8080/servletsexamples/echo?
username=John&password=1234&gender=m&age=99&
language=java&language=cs&...
```

La URL `http://localhost:8080/servletsexamples/echo` se obtiene a partir del atributo `action="echo"` de la etiqueta `<form>`.

El carácter `?` separa la URL de la llamada **cadena de la consulta** (*query string*), es decir, los parámetros de la petición que van seguidos). Esta cadena de la consulta está compuesta por las parejas `name=value` de los elementos de entrada seleccionados por el usuario, separados por un carácter `&`.

Los caracteres especiales no están permitidos en una URL, por lo que tienen que ser codificados. El espacio en blanco se codifica con un `+` o `%20`. Otros caracteres se codifican como `%xx`, siendo `xx` su código ASCII en hexadecimal.

HTTP proporciona dos métodos de petición: **GET** y **POST**. En las peticiones GET, los parámetros de la consulta se añaden al final de la URL. En las peticiones POST, la cadena de la consulta se envía en el cuerpo del mensaje de petición. A menudo, se prefiere usar el método POST, ya que los usuarios no ven cosas extrañas en la URL y se puede enviar una cantidad ilimitada de datos. La cantidad de datos que puede ser enviada con el método GET viene dada por la longitud máxima de la URL (que es variable y depende del navegador).

4.2. LECTURA DE LOS DATOS DE UN FORMULARIO HTML

El formulario que se ha escrito envía sus datos a un programa del lado servidor que tiene la URL relativa `echo` (especificada en el atributo `action="echo"` de la etiqueta `<form>`). El siguiente *servlet* **EchoServlet.java** está asociado a esta URL relativa `"echo"`, procesa los datos del formulario y simplemente los devuelve sin alterar al cliente web (navegador).

```
package mypkg;

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

public class EchoServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        // Set the response message's MIME type
        response.setContentType("text/html; charset=UTF-8");
        // Allocate a output writer to write the response message into the network socket
        PrintWriter out = response.getWriter();
        // Write the response message, in an HTML page
        try {
            out.println("<!DOCTYPE html>");
            out.println("<html><head>");
            out.println("<meta http-equiv='Content-Type' content='text/html; charset=UTF-8'>");
            out.println("<title>Echo Servlet</title></head>");
            out.println("<body><h2>You have enter</h2>");

            // Retrieve the value of the query parameter "username" (from text field)
            String username = request.getParameter("username");
            // Get null if the parameter is missing from query string.
            // Get empty string or string of white spaces if user did not fill in
            if (username == null || (username = htmlFilter(username.trim())).length() == 0) {
                out.println("<p>Name: MISSING</p>");
            } else {
                out.println("<p>Name: " + username + "</p>");
            }

            // Retrieve the value of the query parameter "password" (from password field)
            String password = request.getParameter("password");
            if (password == null || (password = htmlFilter(password.trim())).length() == 0) {
                out.println("<p>Password: MISSING</p>");
            } else {
                out.println("<p>Password: " + password + "</p>");
            }
        }
    }
}
```

```
// Retrieve the value of the query parameter "gender" (from radio button)
String gender = request.getParameter("gender");
// Get null if the parameter is missing from query string.
if (gender == null) {
    out.println("<p>Gender: MISSING</p>");
} else if (gender.equals("m")) {
    out.println("<p>Gender: male</p>");
} else {
    out.println("<p>Gender: female</p>");
}

// Retrieve the value of the query parameter "age" (from pull-down menu)
String age = request.getParameter("age");
if (age == null) {
    out.println("<p>Age: MISSING</p>");
} else if (age.equals("1")) {
    out.println("<p>Age: &lt; 1 year old</p>");
} else if (age.equals("99")) {
    out.println("<p>Age: 1 to 99 years old</p>");
} else {
    out.println("<p>Age: &gt; 99 years old</p>");
}

// Retrieve the value of the query parameter "language" (from checkboxes).
// Multiple entries possible.
// Use getParameterValues() which returns an array of String.
String[] languages = request.getParameterValues("language");
// Get null if the parameter is missing from query string.
if (languages == null || languages.length == 0) {
    out.println("<p>Languages: NONE</p>");
} else {
    out.println("<p>Languages: ");
    for (String language : languages) {
        if (language.equals("c")) {
            out.println("C/C++ ");
        } else if (language.equals("cs")) {
            out.println("C# ");
        } else if (language.equals("java")) {
            out.println("Java ");
        }
    }
    out.println("</p>");
}

// Retrieve the value of the query parameter "instruction" (from text area)
String instruction = request.getParameter("instruction");
// Get null if the parameter is missing from query string.
if (instruction == null
    || (instruction = htmlFilter(instruction.trim())).length() == 0
    || instruction.equals("Enter your instruction here...")) {
    out.println("<p>Instruction: NONE</p>");
} else {
    out.println("<p>Instruction: " + instruction + "</p>");
}
```

```

        // Retrieve the value of the query parameter "secret" (from hidden field)
        String secret = request.getParameter("secret");
        out.println("<p>Secret: " + secret + "</p>");

        // Get all the names of request parameters
        Enumeration names = request.getParameterNames();
        out.println("<p>Request Parameter Names are: ");
        if (names.hasMoreElements()) {
            out.print(htmlFilter(names.nextElement().toString()));
        }
        do {
            out.print(", " + htmlFilter(names.nextElement().toString()));
        } while (names.hasMoreElements());
        out.println("</p>");

        // Hyperlink "BACK" to input page
        out.println("<a href='user_input_form.html'>BACK</a>");
        out.println("</body></html>");
    }
    finally {
        // Always close the output writer
        out.close();
    }
}

// Redirect POST request to GET request.
@Override
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    doGet(request, response);
}

// Filter the string for special HTML characters to prevent command injection attack
private static String htmlFilter(String message) {
    if (message == null) return null;
    int len = message.length();
    StringBuffer result = new StringBuffer(len + 20);
    char aChar;
    for (int i = 0; i < len; ++i) {
        aChar = message.charAt(i);
        switch (aChar) {
            case '<': result.append("&lt;"); break;
            case '>': result.append("&gt;"); break;
            case '&': result.append("&amp;"); break;
            case '"': result.append("&quot;"); break;
            default: result.append(aChar);
        }
    }
    return (result.toString());
}

} // End of EchoServlet

```

La cadena de la consulta contiene parejas `name=value` que se pueden recuperar del mensaje de la petición del cliente (almacenados en el parámetro `request` del método `doGet()`), usando los siguientes métodos:

```

request.getParameter("paramName")
    // Returns the parameter value in a String.
    // Returns null if parameter name does not exist.
    // Returns the first parameter value for a multi-value parameter.

request.getParameterValues("paramName")
    // Return all the parameter values in a String[].
    // Return null if the parameter name does not exist.

request.getParameterNames()
    // Return all the parameter names in a java.util.Enumeration, possibly empty.

```

Los caracteres HTML especiales (<, >, &, ") se deben reemplazar con secuencias de escape en la cadena de entrada, antes de visualizarlos en la página HTML devuelta al cliente. Este paso es necesario para evitar **ataques de inyección de comandos** (el usuario introduce un script en un campo de texto) y lo realiza el método estático `htmlFilter()`.

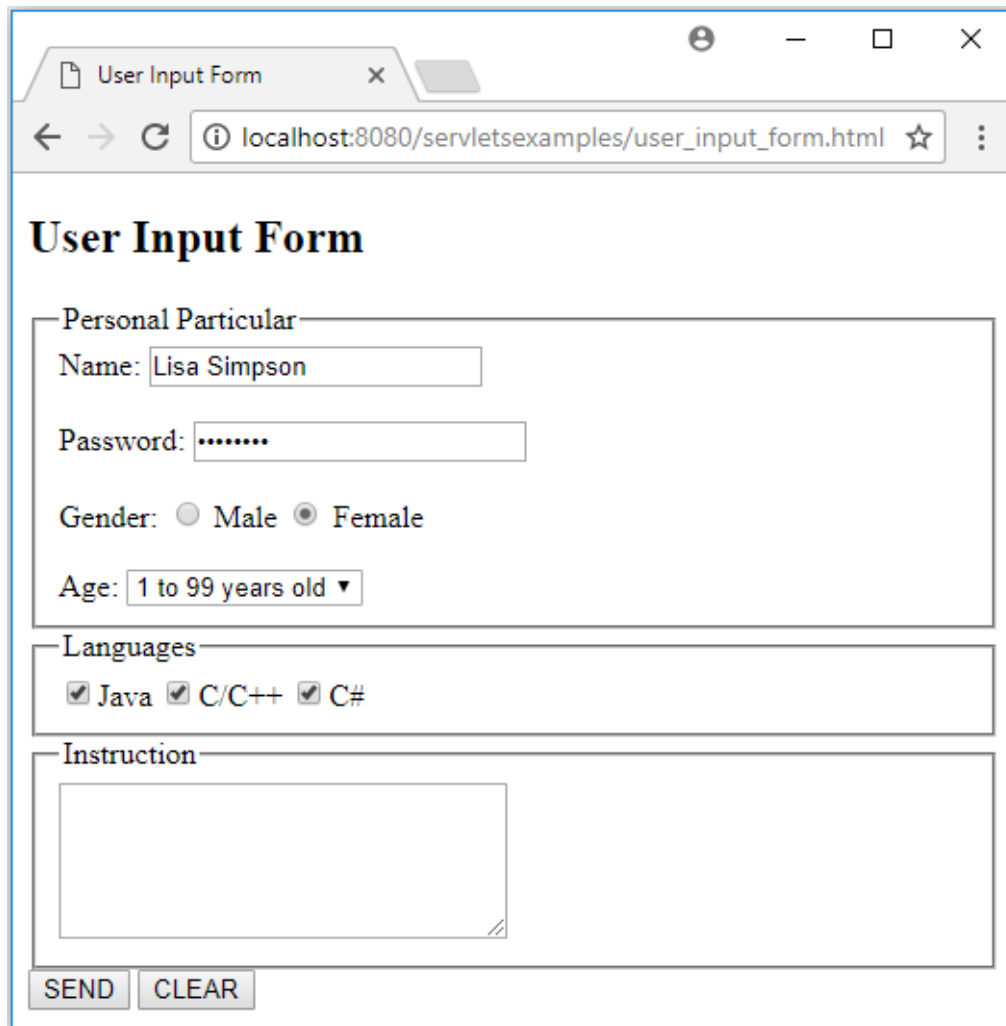
Dado que el atributo `action` de la etiqueta `<form>` de formulario está referida a la URL relativa "echo", el `servlet` `EchoServlet` debe estar mapeado adecuadamente. Para ello, hay que modificar el descriptor de despliegue **web.xml** de la aplicación web:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/
<servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>mypkg.HelloServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>EchoServlet</servlet-name>
    <servlet-class>mypkg.EchoServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/helloservlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>EchoServlet</servlet-name>
    <url-pattern>/echo</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
</web-app>

```

Una vez realizado este cambio, se puede iniciar el servidor de aplicaciones (*GlassFish* o *Apache Tomcat*), ir a la URL del formulario en un navegador web y rellenarlo con datos y pulsar el botón **SEND**:



User Input Form

Personal Particular

Name:

Password:

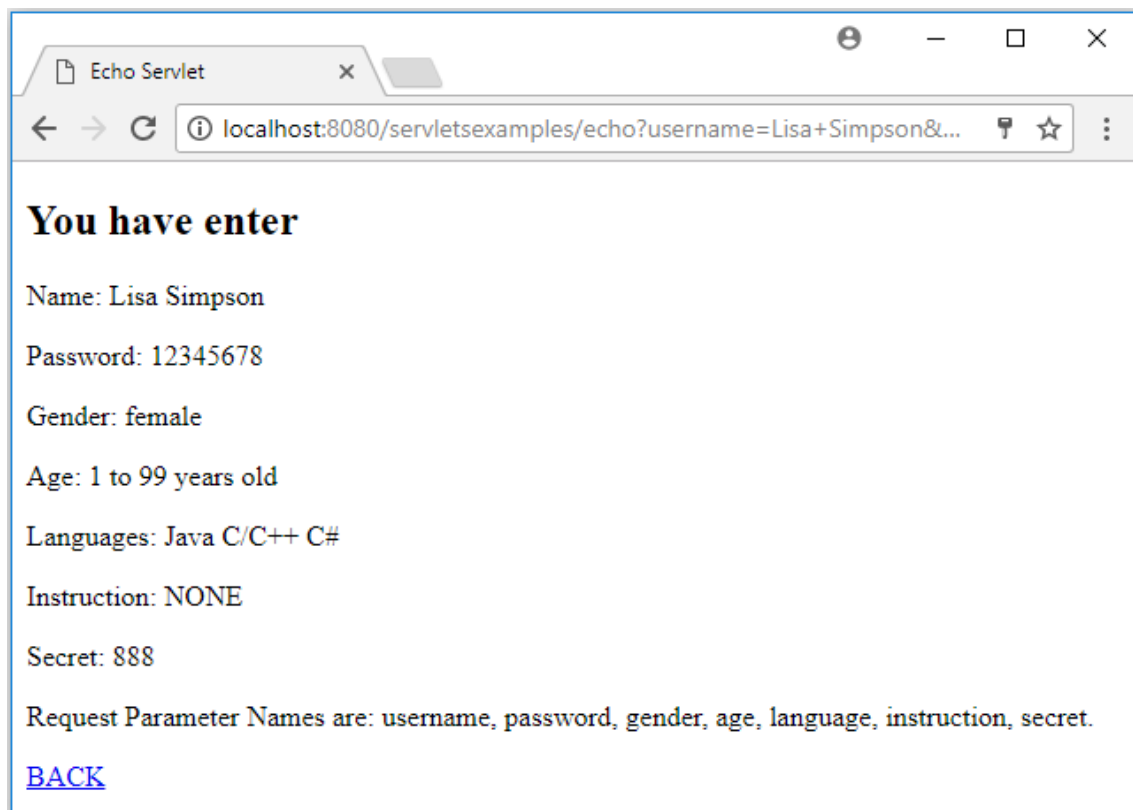
Gender: ☐ Male ☒ Female

Age:

Languages

☒ Java ☒ C/C++ ☒ C#

Instruction



You have enter

Name: Lisa Simpson

Password: 12345678

Gender: female

Age: 1 to 99 years old

Languages: Java C/C++ C#

Instruction: NONE

Secret: 888

Request Parameter Names are: username, password, gender, age, language, instruction, secret.

[BACK](#)

Para probar el método de petición POST, se tiene que modificar la etiqueta `<form>` del formulario `user_input_form.html`:

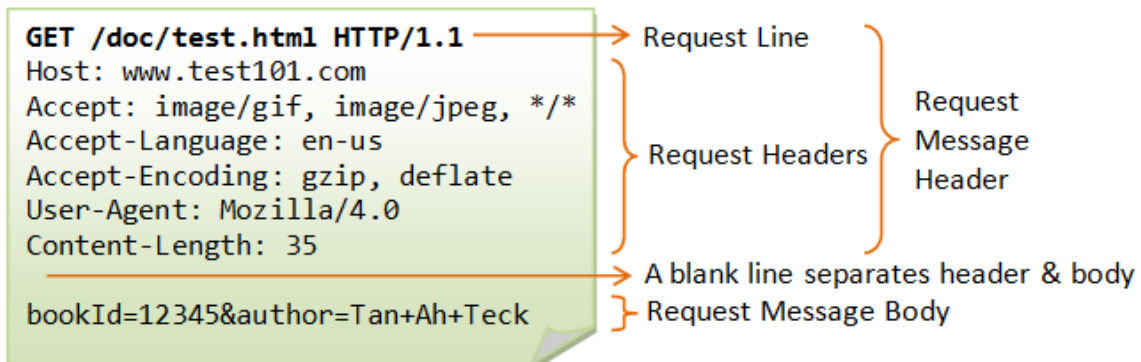
```
<body>
  <h2>User Input Form</h2>
  <form method="post" action="echo">
    <fieldset>
      <legend>Personal Particular</legend>
      Name: <input type="text" name="username" /><br /><br />
      Password: <input type="password" name="password" /><br /><br />
    </fieldset>
  </form>
</body>
```

5. MENSAJE DE PETICIÓN DEL CLIENTE

HTTP es un protocolo de petición y respuesta. El cliente le envía un **mensaje de petición** al servidor y después el servidor le devuelve un **mensaje de respuesta** al cliente. Los mensajes de petición y respuesta son mensajes de texto que están formados por dos partes:

- **Cabecera.** Contiene información sobre el mensaje. Los datos de la cabecera están organizados en parejas nombre-valor.
- **Cuerpo.** Es el contenido del mensaje.

La siguiente figura muestra un ejemplo de un **mensaje de petición HTTP**:



5.1. CABECERAS DE PETICIÓN HTTP

Las principales **cabeceras de petición HTTP 1.1** son las siguientes:

- **Accept.** Especifica los tipos MIME que el navegador o el cliente pueden manejar. Un *servlet* que puede devolver un recurso en más de un formato examinará esta cabecera para decidir qué formato usar.
- **Accept-Charset.** Especifica los conjuntos de caracteres que el navegador soporta para mostrar la información. Por ejemplo, ISO-8859-1.
- **Accept-Encoding.** Especifica los tipos de codificaciones que el navegador puede manejar. Los valores *gzip* y *compress* son las dos opciones más usadas.
- **Accept-Language.** Especifica los idiomas preferidos del cliente, en caso de que el *servlet* pueda producir resultados en más de un idioma.
- **Autorization.** Se utiliza cuando para identificarse cuando los clientes acceden a páginas web protegidas por contraseña.
- **Cache-Control.** El cliente puede utilizar esta cabecera para establecer la cantidad de opciones de la forma en que los servidores *proxy* deberán almacenar temporalmente las páginas.
- **Connection.** Indica si el cliente puede manejar conexiones HTTP persistentes. Tales conexiones persistentes permiten al cliente u otro navegador obtener varios archivos con una sola conexión *socket*. El valor *keep-alive* indica que se deben usar conexiones persistentes.
- **Content-Length.** Sólo se aplica a peticiones POST y devuelve el tamaño en bytes de los datos POST.

- **Content-Type.** Se utiliza generalmente en respuestas del servidor y también puede ser parte de las peticiones del cliente cuando adjunta un documento como datos POST o cuando se realizan peticiones PUT.
- **Cookie.** Se utiliza para devolver las *cookies* a los servidores que previamente las hayan enviado al navegador.
- **Host.** Indica la máquina anfitrión (*host*) y el puerto del servidor tal como se muestra en la URL original.
- **If-Modified-Since.** Indica que el cliente desea la página sólo si ha sido modificada desde una fecha indicada. Permite que el navegador web almacene temporalmente los documentos y los vuelva a cargar a través de la red sólo si han cambiado. Se usa para peticiones GET.
- **If-Unmodified-Since.** Indica que la operación deberá salir adelante sólo si el documento es anterior a la fecha establecida. Se utiliza para peticiones PUT.
- **Referer.** Indica la URL de la página web a la que se hace referencia. Sirve para rastrear de dónde provienen las peticiones.
- **Upgrade.** Permite al navegador web o a otro cliente especificar un protocolo de comunicación preferido sobre HTTP.
- **User-Agent.** Identifica al navegador web o a otro cliente que hace la petición y puede ser utilizado para devolver contenido distinto a diferentes tipos de navegadores.

5.2. INTERFACES ASOCIADAS AL MENSAJE DE PETICIÓN

Los métodos *doGet()* y *doPost()* tienen un objeto **HttpServletRequest** como primer parámetro, que proporciona diversos métodos para obtener la información de la cabecera del mensaje de petición:

- Métodos propios:
 - *getHeader(name)*, *getHeaders(name)*, *getHeaderNames()*.
 - *getMethod()*, *getQueryString()*.
 - *getCookies()*, *getAuthType()*, *getSession()*, *getRemoteUser()*.
 - *getContextPath()*, *getServletPath()*, *getRequestURI()*, *getRequestURL()*.
- Métodos heredados de **ServletRequest**:
 - *getProtocol()*, *getScheme()*.
 - *getCharacterEncoding()*, *getContentLength()*, *getContentType()*.
 - *getServerName()*, *getServerPort()*.
 - *getParameter()*, *getParameterNames()*, *getParameterValues(name)*.
 - *getAttribute(name)*, *getAttributeNames()*.

Para más información, consultar la documentación de las interfaces:

<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html>

<http://docs.oracle.com/javaee/6/api/javax/servlet/ServletRequest.html>

5.3. OBTENCIÓN DE LA CABECERA DEL MENSAJE DE PETICIÓN

El siguiente *serv/et* **RequestHeaderServlet.java** recupera y muestra en una página HTML estática los datos de la cabecera del mensaje de petición del cliente:

```
package mypkg;

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

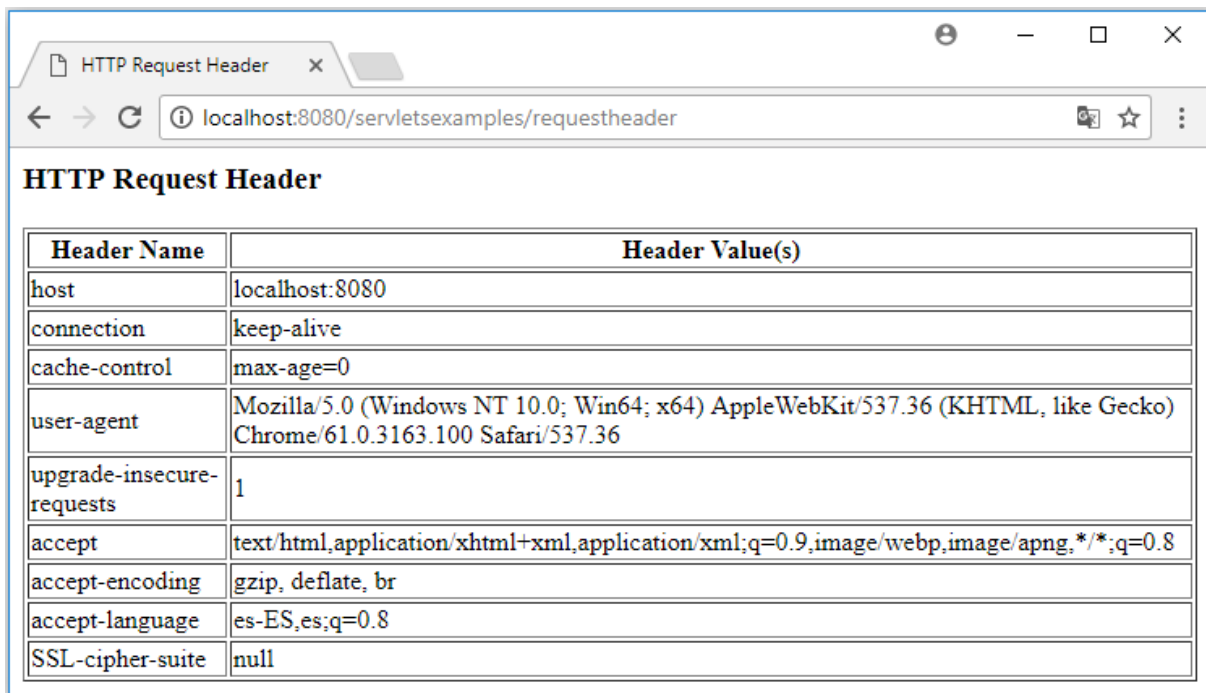
public class RequestHeaderServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        // Set the response message's MIME type
        response.setContentType("text/html;charset=UTF-8");
        // Allocate a output writer to write the response message into the network socket
        PrintWriter out = response.getWriter();
        // Write the response message, in an HTML page
        try {
            // HTML 5
            out.println("<!DOCTYPE html");
            out.println("<html><head>");
            out.println("<meta http-equiv='Content-Type' content='text/html; charset=UTF-8'>");
            String title = "HTTP Request Header";
            out.println("<head><title>" + title + "</title></head>");
            out.println("<body>");
            out.println("<h3>" + title + "</h3>");
            // Tabulate the request header information
            out.println("<table border=1>");
            out.println("<th>Header Name<th>Header Value(s)");
            Enumeration headerNames = request.getHeaderNames();
            while (headerNames.hasMoreElements()) {
                String paramName = (String) headerNames.nextElement();
                String paramValue = request.getHeader(paramName);
                out.println("<tr><td>" + paramName + "</td>");
                out.println("<td>" + paramValue + "</td></tr>");
            }
            // SSL (HTTPS) Cipher suites
            String cipherSuite = (String) request.getAttribute("javax.servlet.request.cipher_suite");
            out.println("<tr><td>SSL-cipher-suite</td>");
            if (cipherSuite != null) {
                out.println("<td>" + cipherSuite + "</td></tr>");
            }
            else {
                out.println("<td>null</td></tr>");
            }
            out.println("</table></body></html>");
        }
        finally {
            // Always close the output writer
            out.close();
        }
    }
}
```

```
// Do the same thing for GET and POST requests
@Override
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
    doGet(request, response);
}

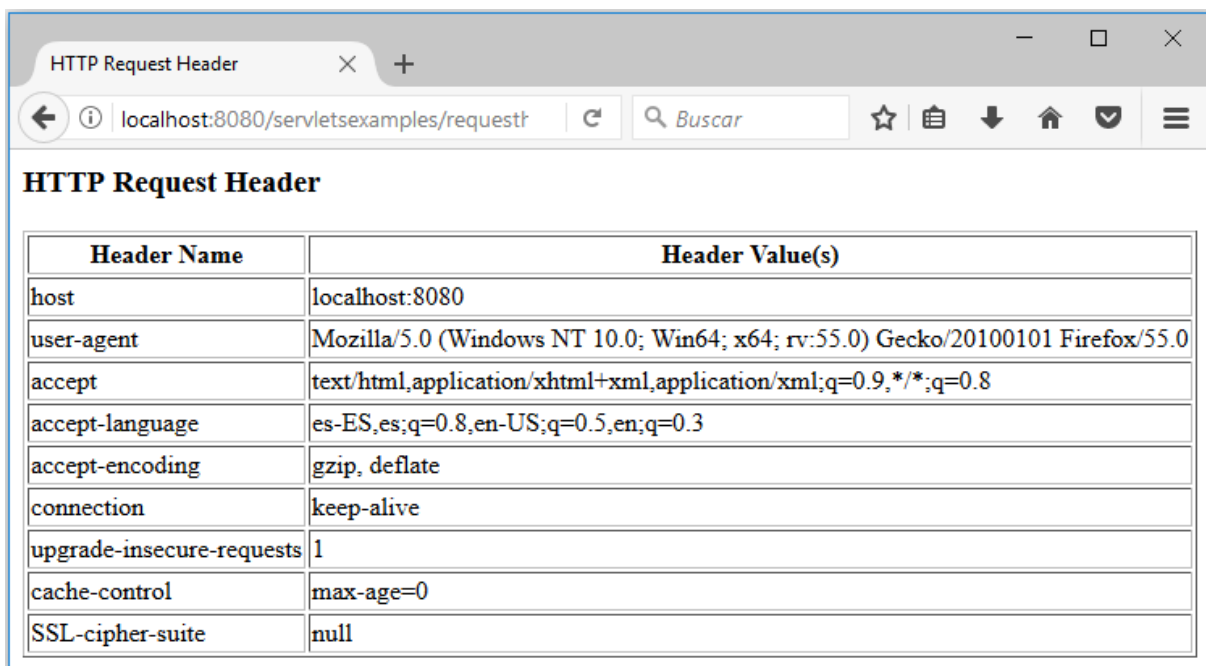
} // End of RequestHeaderServlet
```

Si la petición se realiza desde el navegador Chrome, este *servlet* devuelve los siguientes datos de la cabecera de petición del cliente:



Header Name	Header Value(s)
host	localhost:8080
connection	keep-alive
cache-control	max-age=0
user-agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
upgrade-insecure-requests	1
accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
accept-encoding	gzip, deflate, br
accept-language	es-ES,es;q=0.8
SSL-cipher-suite	null

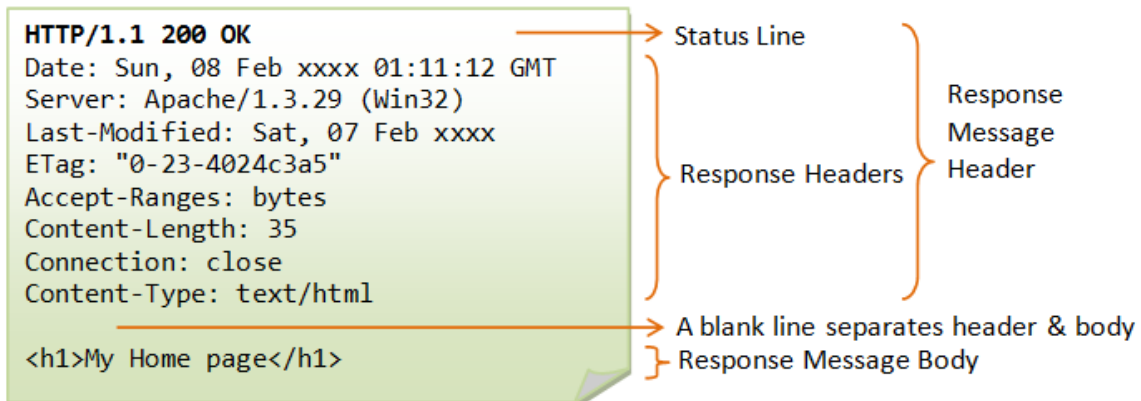
Si la petición se realiza desde el navegador Firefox, este *servlet* devuelve los siguientes datos de la cabecera de petición del cliente:



Header Name	Header Value(s)
host	localhost:8080
user-agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:55.0) Gecko/20100101 Firefox/55.0
accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-language	es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
accept-encoding	gzip, deflate
connection	keep-alive
upgrade-insecure-requests	1
cache-control	max-age=0
SSL-cipher-suite	null

6. MENSAJE DE RESPUESTA DEL SERVIDOR

La siguiente figura muestra un ejemplo de un **mensaje de respuesta HTTP**:



6.1. CABECERAS DE RESPUESTA HTTP

Las principales **cabeceras de respuesta HTTP 1.1** son las siguientes:

- **Allow.** Especifica los métodos de petición (GET, POST, PUT, DELETE) que el servidor soporta.
- **Cache-Control.** Especifica las circunstancias bajo las cuales el documento de respuesta se puede cachear con seguridad. Puede tener los siguientes valores: *public* (el documento se puede cachear), *private* (el documento es para un único usuario y solo se puede almacenar en caches privadas no compartidas) o *no-cache* (el documento no se debe cachear).
- **Connection.** Indica al navegador si tiene que usar una conexión HTTP persistente o no. El valor *close* le indica al navegador que no la utilice, mientras que el valor *keep-alive* significa lo contrario.
- **Content-Disposition.** Permite que el navegador le dé al usuario la opción de guardar la respuesta a disco como un fichero con un nombre dado.
- **Content-Encoding.** Especifica la forma en la que la página se ha codificado para la transmisión.
- **Content-Language.** Especifica el idioma en el que el documento está escrito.
- **Content-Length.** Indica el número de bytes del mensaje de respuesta. Esta información sólo se necesita si el navegador utiliza una conexión HTTP persistente (*keep-alive*).
- **Content-Type.** Indica el tipo MIME (Extensiones Multipropósito de Correo de Internet) del documento de respuesta. El tipo MIME predeterminado de los *servlets* es *text/plain*, pero por lo general, los *servlets* establecen específicamente *text/html*.
- **Expires.** Especifica el momento temporal en el que el contenido se debería considerar obsoleto, caducado o desactualizado para que no se cachee más.
- **Last-Modified.** Indica cuándo fue modificado el documento por última vez. El cliente puede entonces cachear el documento y proporcionar una fecha en la cabecera de petición *If-Modified-Since* de las sucesivas solicitudes.
- **Location.** Esta cabecera se debería incluir en todas las respuestas del servidor que tengan un código de estado 3XX. Sirve para indicar la dirección del documento al navegador. Automáticamente, el navegador se conecta a esta dirección y recupera el nuevo documento.

- **Refresh.** Especifica el intervalo de tiempo con el que navegador debería solicitar la página actualizada. Se puede indicar una tasa de refresco, medida en número de segundos, que tiene que transcurrir para que la página se refresque.
- **Retry-After.** Se utiliza junto con el mensaje de respuesta 503 (servicio no disponible) para informar al cliente de cuánto tiempo debe esperar para repetir su solicitud.
- **Set-Cookie.** Especifica una cookie asociada con la página que se sirve.

6.2. INTERFACES ASOCIADAS AL MENSAJE DE RESPUESTA

Los métodos *doGet()* y *doPost()* tienen un objeto **HttpServletResponse** como segundo parámetro, que proporciona diversos métodos para generar la información de la cabecera del mensaje de respuesta:

1) Métodos propios:

- *getHeader(name)*, *setHeader(name,value)*, *addHeader(name,value)*, *getHeaders(name)*, *getHeaderNames()*, *addCookie(cookie)*.
- *encodeURL(url)*, *encodeRedirectURL(url)*, *sendRedirect(location)*.
- *getStatus()*, *setStatus(statuscode)*, *sendError(statuscode, message)*.

2) Métodos heredados de **ServletResponse**:

- *getBufferSize()*, *setBufferSize(size)*, *resetBuffer()*, *flushBuffer()*.
- *getCharacterEncoding()*, *setCharacterEncoding(charset)*.
- *getContentType()*, *setContentType(type)*, *setContentLength(length)*.
- *getWriter()*, *getOutputStream()*.

Para más información, consultar la documentación de las interfaces:

<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html>

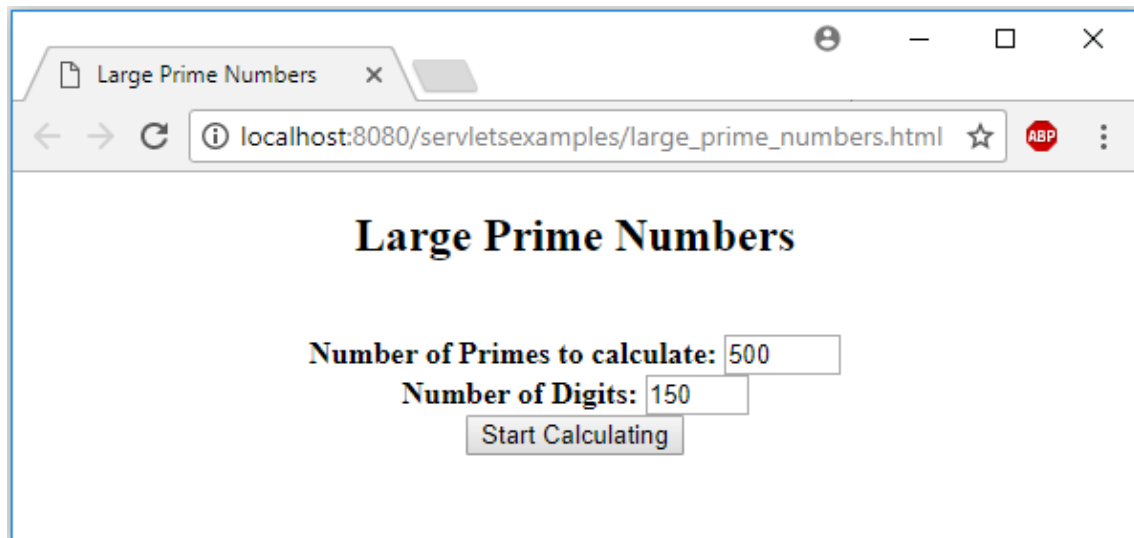
<http://docs.oracle.com/javaee/6/api/javax/servlet/ServletResponse.html>

6.3. ESTADO PERSISTENTE DEL SERVLET Y PÁGINAS DE RECARGA AUTOMÁTICA

El siguiente caso práctico permite al usuario solicitar una lista de **números primos grandes**, elegidos al azar. Para números muy largos, este cálculo puede tardar cierto tiempo, por lo que el *servlet* inmediatamente devuelve unos resultados iniciales aunque continúe calculando mediante un *thread* de baja prioridad, de modo que no reduzca el rendimiento del servidor. Si los cálculos no se han completado, el *servlet* le indica al navegador web que solicite una nueva página dentro de unos segundos enviándole una cabecera *refresh*.

El formulario **large_prime_numbers.html** es la página web que actúa como interfaz del *servlet* y que pide al usuario que especifique dos parámetros:

- El número de primos a calcular (por defecto, 500 números primos).
- El número de dígitos de los números (por defecto, números de unos 150 dígitos).



Su código HTML es el siguiente:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Large Prime Numbers</title>
  </head>
  <body>
    <center>
      <h2>Large Prime Numbers</h2>
      <br>
      <form action="primenumber">
        <b>Number of Primes to calculate:</b>
        <input type="text" name="NumPrimes" value=500 size=4><br>
        <b>Number of Digits:</b>
        <input type="text" name="NumDigits" value=150 size=3><br>
        <input type="submit" value="Start Calculating">
      </form>
    </center>
  </body>
</html>
```

Para desarrollar el *servlet* *PrimeNumberServlet* con la funcionalidad requerida, se necesitan varias clases que incluyen código auxiliar: *Primes.java*, *PrimeList.java* y *ServletUtilities.java*.

La clase **Primes.java** contiene los algoritmos de bajo nivel que se usan para elegir un número aleatorio de una longitud establecida, y luego, encontrar un número primo en o por encima de ese valor:

- Utiliza métodos integrados en la clase *BigInteger*.
- Dado que el algoritmo que determina si un número es primo es uno probabilístico, existe la posibilidad de errores.
- Asume que la probabilidad de devolver erróneamente un número como primo es $1 / 2^{100}$.

La clase **PrimeList.java** crea un *thread* o hilo de ejecución en segundo plano que se utiliza para generar una lista de números primos grandes:

- Genera una lista de números primos grandes a partir de los parámetros establecidos (número de primos a calcular y número de dígitos de los números).
- Normalmente, utiliza un *thread* en segundo plano con prioridad baja para no ralentizar al servidor.
- Proporciona algunos métodos de acceso a prueba de *threads*, dado que la lista de números primos puede ser compartida por varios *threads* de clientes y el acceso a su contenido se debe realizar en exclusión mutua.

La clase **ServletUtilities.java** contiene funciones estáticas que el *servlet* *PrimeNumberServlet* necesita para:

- Leer un parámetro como una cadena de caracteres (si el formato de número entero es erróneo o si no hay parámetro, se proporciona un valor por defecto) y convertirlo a un dato entero (*int*).
- Leer un parámetro como una cadena de caracteres (si el formato de número real es erróneo o si no hay parámetro, se proporciona un valor por defecto) y convertirlo a un dato real (*double*).
- Reemplazar los caracteres especiales (que tienen un significado HTML) de una cadena de texto con sus entidades de caracteres HTML correspondientes.

El siguiente *servlet* **PrimeNumberServlet.java** procesa una petición de un cliente para generar *N* números primos, cada uno con al menos *M* dígitos. Realiza los cálculos en un *thread* en segundo plano con prioridad baja y sólo devuelve los resultados obtenidos hasta ese momento. Si los resultados son parciales (no están terminados), envía una cabecera *refresh* para ordenarle al navegador web que solicite nuevos resultados dentro de 5 segundos.

Este *servlet* también gestiona una colección que contiene las listas de números primos calculadas anteriormente, de forma que, si un nuevo cliente solicita una lista de números primos con unos parámetros *N* y *M* iguales a los de una lista generada recientemente, se le pueda devolver de forma inmediata una lista con resultados parciales o totales.

```
package prime;

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

// Servlet that processes a request to generate n prime numbers, each with at least m digits.
// It performs the calculations in a low-priority background thread, returning only the results
// it has found so far. If these results are not complete, it sends a Refresh header instructing
// the browser to ask for new results a little while later.
// It also maintains a list of a small number of previously calculated prime lists to return
// immediately to anyone who supplies the same n and m as a recently completed computation.
public class PrimeNumberServlet extends HttpServlet {

    private ArrayList primeListCollection = new ArrayList();
    private int maxPrimeLists = 30;
```



```

@Override
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    int numPrimes = ServletUtilities.getIntParameter(request, "NumPrimes", 500);
    int numDigits = ServletUtilities.getIntParameter(request, "NumDigits", 150);
    PrimeList primeList = findPrimeList(primeListCollection, numPrimes, numDigits);
    if (primeList == null) {
        primeList = new PrimeList(numPrimes, numDigits, true);
        // Multiple servlet request threads share the instance variables (fields) of
        // PrimeNumbers. So synchronize all access to servlet fields.
        synchronized (primeListCollection) {
            if (primeListCollection.size() >= maxPrimeLists) {
                primeListCollection.remove(0);
            }
            primeListCollection.add(primeList);
        }
    }
    ArrayList currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
        response.setIntHeader("Refresh", 5);
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Some " + numDigits + "-Digit Prime Numbers";
    out.println(ServletUtilities.headWithTitle(title) + "<body>\n" +
        "<h2 align=center>" + title + "</h2>\n" +
        "<h3>Primes found with " + numDigits + " digits: " + numCurrentPrimes + "</h3>");
    if (isLastResult)
        out.println("<b>Search of Prime Numbers completed.</b>");
    else
        out.println("<b>Still looking for " + numPrimesRemaining + " Prime Numbers.</b>");
    out.println("<ol>");
    for (int i = 0; i < numCurrentPrimes; i++) {
        out.println("  <li>" + currentPrimes.get(i));
    }
    out.println("</ol>");
    out.println("</body></html>");
}

// See if there is an existing ongoing or completed calculation with the same number of
// primes and number of digits per prime. If so, return those results instead of starting
// a new background thread.
// Keep this list small so that the web server doesn't use too much memory.
// Synchronize access to the list since there may be multiple simultaneous requests.
private PrimeList findPrimeList(ArrayList primeListCollection, int numPrimes, int numDigits) {
    synchronized (primeListCollection) {
        for(int i = 0; i < primeListCollection.size(); i++) {
            PrimeList primes = (PrimeList) primeListCollection.get(i);
            if ((numPrimes == primes.numPrimes()) && (numDigits == primes.numDigits()))
                return (primes);
        }
        return (null);
    }
}

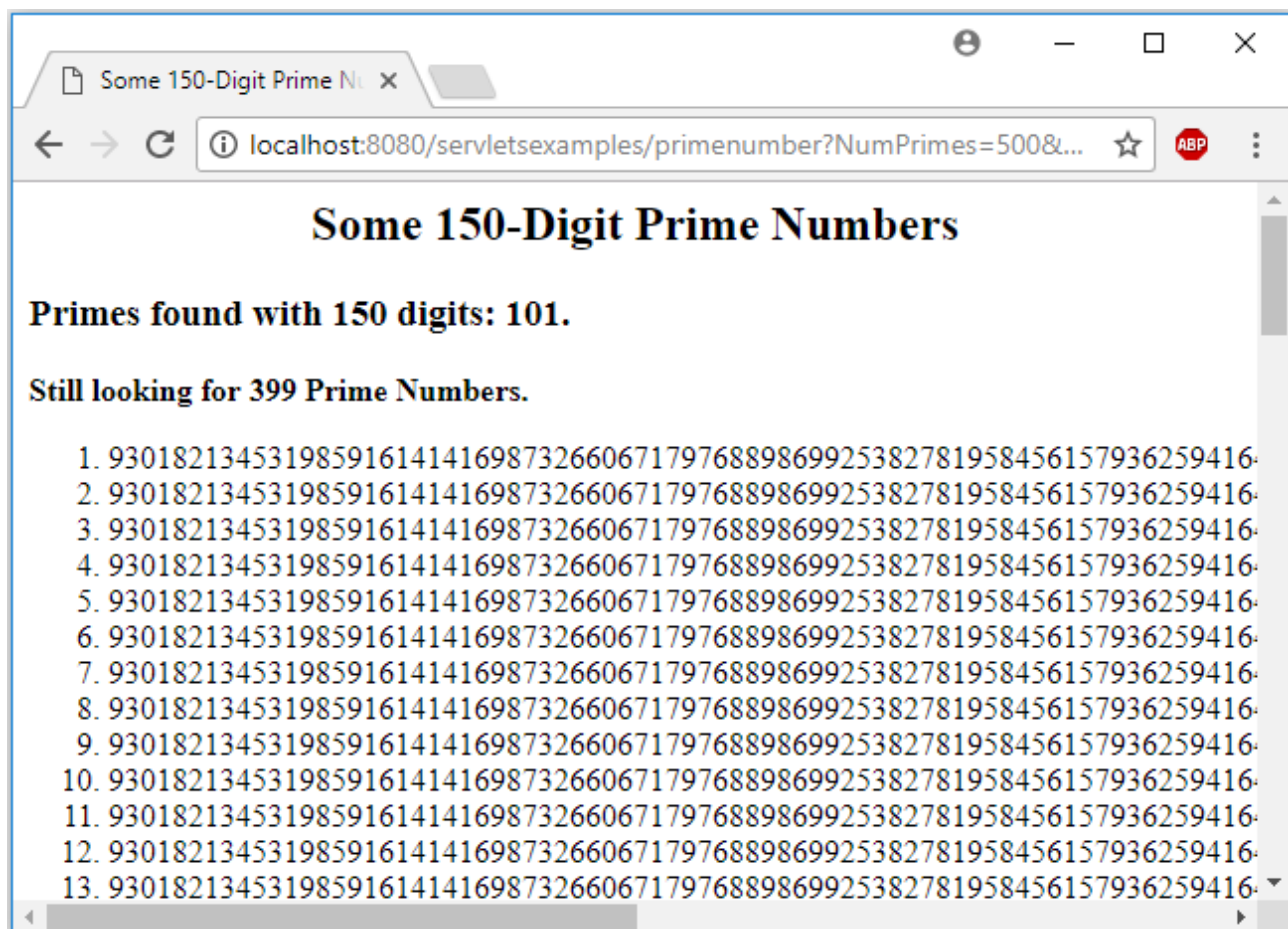
} // End of PrimeNumberServlet

```

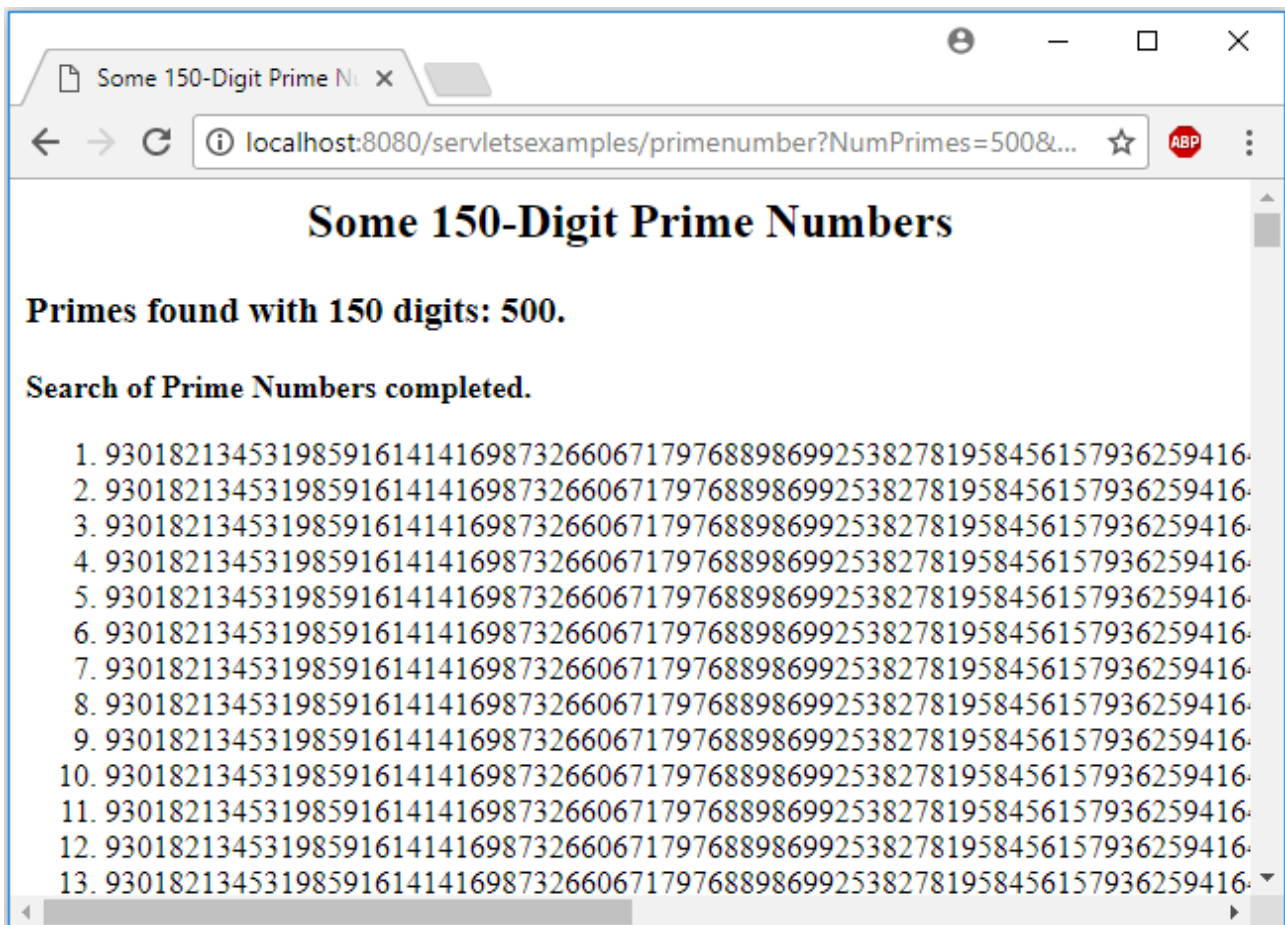

Este *servlet* sobrescribe el método *doGet()*, que realiza lo siguiente:

- Recibe una petición GET con dos parámetros procedentes del usuario y enviados al *servlet* mediante un formulario HTML.
- Estos parámetros (*numPrimes* y *numDigits*) se convierten en enteros usando una función estática auxiliar que llama a *Integer.parseInt()*.
- A continuación, se invoca el método privado *findPrimeList()* con estos parámetros. Este método busca una lista de números primos (que puede contener resultados parciales o finales) con estos parámetros específicos en la colección de listas de primos del *servlet*.
- Si se ha encontrado una lista, se utiliza. En caso contrario, se crea una nueva *PrimeList* y se almacena en la colección de listas de números primos del *servlet*.
- Después, se comprueba si el *thread* asociado a la lista *PrimeList* ha terminado de calcular todos sus números primos. Si no es así, se envía al cliente una cabecera *Refresh* para indicarle que regrese en 5 segundos para obtener resultados actualizados.
- De cualquier modo, el servidor devuelve al cliente una respuesta que informa del progreso del cálculo y que incluye una lista de los números primos encontrados hasta ese momento.

La siguiente captura muestra el **resultado intermedio** de una petición al *servlet* *PrimeNumberServlet*. Este resultado se obtiene cuando el navegador se refresque de forma automática o cuando otro cliente solicite una petición con los mismos parámetros que los de una petición en curso o reciente. De cualquier forma, el navegador automáticamente volverá a cargar la página para obtener los resultados actualizados.



La siguiente captura muestra el **resultado final** de una petición al *servlet* *PrimeNumberServlet*. Este resultado se obtiene cuando el navegador se refresque de forma automática o cuando otro cliente solicite una petición con los mismos parámetros que los de una petición en curso o reciente. En este punto, el navegador dejará de actualizar la página.



Este ejemplo muestra dos facetas valiosas de los *servlets*:

- 1) **El *servlet* puede administrar varias conexiones simultáneas.** Cada conexión de un cliente será manejada por un *thread* o hilo de ejecución creado explícitamente para ello. De esta forma, mientras un *thread* finaliza un cálculo en segundo plano para un cliente, otro cliente puede conectarse y ver los resultados parciales.
- 2) **El *servlet* conserva el estado entre peticiones.** Sólo se genera una instancia del *servlet* y cada petición de un cliente dará como resultado un nuevo *thread* que ejecute el método *service()* del *servlet*.
 - Los datos compartidos tendrán que ser colocados en una instancia de variable normal (campo) del *servlet*.
 - El *servlet* podrá acceder a los cálculos adecuados que estén en curso, cuando el navegador web recargue la página y obtenga la lista de los resultados solicitados recientemente.
 - El *servlet* podrá devolver estos resultados inmediatamente, si una nueva petición especifica los mismos parámetros que otra petición anterior.

7. CÓDIGOS DE ESTADO HTTP

HTTP ha pasado por múltiples versiones del protocolo, muchas de las cuales son compatibles con las anteriores. La versión más usada actualmente es la 1.1 (1999), aunque la versión 2.0 (2015) tiene características que mejoran las carencias existentes en anteriores versiones.

El formato de los mensajes de petición y respuesta HTTP tiene la siguiente estructura:

- Una línea de estado inicial.
- Cero o más líneas de cabeceras.
- Una línea en blanco.
- Un cuerpo de mensaje opcional como un fichero, datos de consulta o salidas de consulta.

7.1. DESCRIPCIÓN DE LOS CÓDIGOS DE ESTADO HTTP

A continuación, se muestran varias tablas con los **códigos de estado HTTP 1.1** y los mensajes asociados que el servidor web podría devolver al cliente.

Los **códigos de estado comprendidos entre 100 y 199** tienen propósitos informativos. Indican que la petición ha sido recibida y se está procesando. El cliente deberá responder con alguna otra acción.

CÓDIGO	MENSAJE	DESCRIPCIÓN
100	<i>Continue</i>	El cliente está preguntando si puede enviar un documento en una petición reiterada. El servidor sólo ha recibido una parte de la petición, pero mientras no haya sido rechazada, el cliente debería continuar con la petición.
101	<i>Switching Protocol</i>	El servidor ha acatado la cabecera <i>upgrade</i> y ha cambiado a un protocolo diferente.

Los **códigos de estado comprendidos entre 200 y 299** significan que la petición realizada por el cliente ha tenido éxito y se ha procesado correctamente.

CÓDIGO	MENSAJE	DESCRIPCIÓN
200	<i>OK</i>	La petición ha ido bien.
201	<i>Created</i>	La petición está completada y se ha creado un nuevo recurso. La cabecera <i>location</i> devuelve su URL.
202	<i>Accepted</i>	Se ha aceptado la petición para procesar, pero su procesamiento no ha acabado.
203	<i>Non-authoritative Information</i>	La página se ha generado y devuelto de forma normal, pero parte de las cabeceras de respuesta podrían ser incorrectas dado que se está usando una copia de la página.
204	<i>No Content</i>	El navegador web deberá continuar mostrando la página anterior, dado que no está disponible una nueva.

205	<i>Reset Content</i>	No hay una página nueva, pero el navegador web deberá refrescar la vista de la página.
206	<i>Partial Content</i>	El servidor ha procesado una petición parcial que incluye una cabecera <i>range</i> .

Los **códigos de estado comprendidos entre 300 y 399** se utilizan para archivos que han sido movidos (redirección) y generalmente incluyen una cabecera *location* que indica la nueva dirección. Indican que el cliente necesita realizar más acciones para finalizar la petición.

CÓDIGO	MENSAJE	DESCRIPCIÓN
300	<i>Multiple Choices</i>	La página solicitada se puede encontrar en varios lugares, que serán listados en la respuesta. Si el servidor tiene una opción preferente, se deberá indicar en la cabecera <i>location</i> .
301	<i>Moved Permanently</i>	La página solicitada ha sido movida de forma permanente a una nueva URL, que se indica en la cabecera <i>location</i> .
302	<i>Found</i>	La página solicitada ha sido movida de forma temporal a una nueva URL, que se indica en la cabecera <i>location</i> .
303	<i>See Other</i>	La página solicitada se puede encontrar en otra URL distinta. Si la petición original fue POST, la nueva página (indicada en la cabecera <i>location</i>), se deberá obtener con GET.
304	<i>Not Modified</i>	La página solicitada que el navegador tiene almacenada en la cache está actualizada y es la que el cliente deberá utilizar.
305	<i>Use Proxy</i>	La página solicitada se deberá obtener mediante el proxy indicado en la cabecera <i>location</i> .
306	<i>Unused</i>	Este código se usaba en anteriores versiones. Ya no se utiliza, pero el código se ha reservado.
307	<i>Temporary Redirect</i>	La página solicitada ha sido movida de forma temporal a una nueva URL, que se indica en la cabecera <i>location</i> . Si la petición original fue POST, la nueva página se deberá obtener con GET.

Los **códigos de estado comprendidos entre 400 y 499** indican un error en el cliente. Indican que ha habido un error en el procesamiento de la petición a causa de que el cliente ha hecho algo mal.

CÓDIGO	MENSAJE	DESCRIPCIÓN
400	<i>Bad Request</i>	El servidor no ha comprendido la petición debido a una sintaxis incorrecta.
401	<i>Unauthorized</i>	La página solicitada necesita un nombre de usuario y una contraseña. Estas credenciales se deberán indicar

		en la cabecera <i>authorization</i> .
402	<i>Payment Required</i>	Este código aún no se puede usar.
403	<i>Forbidden</i>	El servidor ha rehusado el acceso a la página solicitada, sin importar la autorización. Esto se debe a permisos erróneos de archivos o directorios en el servidor.
404	<i>Not Found</i>	El servidor no ha podido encontrar la página solicitada en la dirección indicada.
405	<i>Method Not Allowed</i>	El método especificado (GET, POST, PUT, DELETE) en la petición no está permitido para esta página en particular.
406	<i>Not Acceptable</i>	La página solicitada tiene un tipo MIME que no es compatible con los tipos indicados en la cabecera <i>accept</i> .
407	<i>Proxy Authentication Required</i>	Antes de que la petición sea servida, el cliente se debe autenticar con un servidor proxy. El servidor devolverá la cabecera <i>proxy-authenticate</i> al cliente, dando lugar a una reconexión del navegador web con la cabecera <i>proxy-authorization</i> .
408	<i>Request Timeout</i>	La petición ha tardado más tiempo en llegar al servidor de lo que el servidor estaba preparado para esperar.
409	<i>Conflict</i>	La petición no ha podido ser completada por causa de un conflicto. Ocurre cuando en una petición PUT se intenta enviar una versión incorrecta de un archivo.
410	<i>Gone</i>	La página solicitada ya no está disponible y no se conoce una dirección de reenvío.
411	<i>Length Required</i>	El servidor no ha podido procesar la petición (POST con documento adjunto), porque se necesita la cabecera <i>content-length</i> .
412	<i>Precondition Failed</i>	Cierta precondition establecida en la petición ha sido evaluada a falso por el servidor.
413	<i>Request Entity Too Large</i>	El servidor no aceptará la petición, porque el documento solicitado es mayor de lo que quiere manejar el servidor en ese momento. Si después el servidor piensa que lo puede procesar, deberá incluir una cabecera <i>retry-after</i> .
414	<i>Request-URI Too Long</i>	El servidor no aceptará la petición, porque la URI (parte de la URL que va después del host y del puerto) es demasiado larga. Ocurre cuando se convierte una petición POST en una petición GET con una cadena de consulta larga.

415	<i>Unsupported Media Type</i>	El servidor no aceptará la petición, porque la petición contiene un documento adjunto con un tipo de medio que el servidor no sabe manejar.
416	<i>Requested Range Not Satisfiable</i>	El cliente ha incluido en la petición una cabecera <i>range</i> que no se satisface.
417	<i>Expectation Failed</i>	El cliente ha enviado una petición con una cabecera <i>expect</i> para preguntar si puede enviar un documento adjunto en una petición recurrente. Si el servidor no puede procesar este documento, utiliza este código para indicar que no acepta el documento.

Los **códigos de estado comprendidos entre 500 y 599** indican un error en el servidor. Indican que ha habido un error en el procesamiento de la petición a causa de un fallo en el servidor.

CÓDIGO	MENSAJE	DESCRIPCIÓN
500	<i>Internal Server Error</i>	La petición no se ha completado. El servidor ha encontrado una condición inesperada. Se devuelve como resultado de programas CGI o <i>servlets</i> que han tenido un error o han retornado cabeceras con formatos inadecuados.
501	<i>Not Implemented</i>	La petición no se ha completado. El servidor no es compatible con la funcionalidad requerida para satisfacer la petición.
502	<i>Bad Gateway</i>	La petición no se ha completado. El servidor ha recibido una respuesta inválida del servidor ascendente. Este código lo devuelven servidores que funcionan como <i>proxies</i> o puertas de enlace.
503	<i>Service Unavailable</i>	La petición no se ha completado. El servidor no puede responder porque está temporalmente sobrecargado, caído o en mantenimiento.
504	<i>Gateway Timeout</i>	El servidor no ha obtenido una respuesta a tiempo de otro servidor: la puerta de enlace ha expirado. Este código lo devuelven servidores que funcionan como <i>proxies</i> o puertas de enlace.
505	<i>HTTP Version Not Supported</i>	El servidor no es compatible con la versión del protocolo HTTP.

7.2. REDIRECCIONES Y ERRORES

El siguiente caso práctico utiliza dos de los códigos de estado más comunes (además del 200 OK):

- El **código de estado 302 (Found)** se establece mediante el método `sendRedirect` de la interfaz `HttpServletResponse`. Este método genera una cabecera `location` que devuelve la URL de la nueva página.
- El **código de estado 404 (Not Found)** se establece mediante el método `sendError` de la interfaz `HttpServletResponse`. Este método envía además un pequeño mensaje que de forma automática se formatea dentro de una página HTML.

El `servlet FormBuilderServlet.java` construye un formulario HTML que permite al usuario introducir una cadena de búsqueda y elegir uno de los motores de búsqueda disponibles (*Google, Bing, Yahoo, Baidu, Ask, Aol, DuckDuckGo, WolframAlpha* o *Yandex*) para su uso.

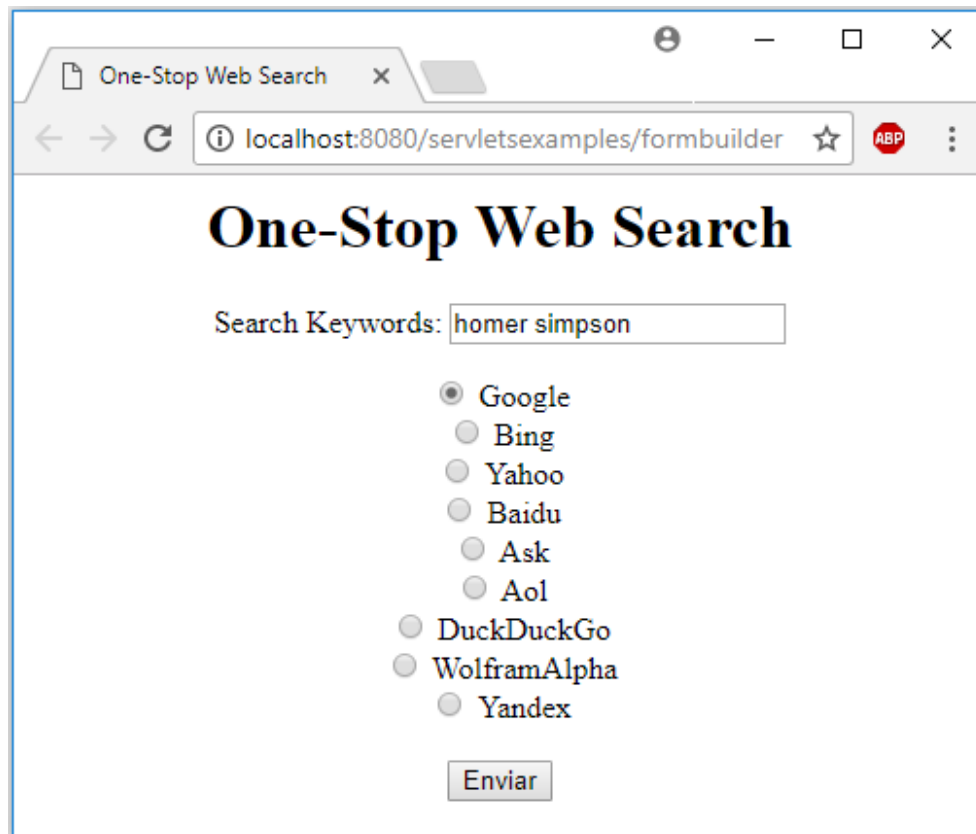
```
package search;

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

// Servlet that builds the HTML form that gathers input for the search engine servlet.
// This servlet first displays a textfield for the search query, then looks up the search engine
// names known to SearchUtilities and displays a list of radio buttons, one for each search engine.
public class FormBuilderServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "One-Stop Web Search";
        String actionURL = "/servletsexamples/searchengine";
        String docType = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " + "Transitional//EN">\n";
        out.println (docType +
            "<html>\n" + "<head><title>" + title + "</title></head>\n" +
            "<body>\n" + "<center>\n" +
            "<h1>" + title + "</h1>\n" +
            "<form action=\"" + actionURL + "\">\n" +
            "  Search Keywords: \n" +
            "    <input type=\"text\" name=\"SearchString\"><p>\n");
        SearchSpec[] specs = SearchUtilities.getCommonSpecs();
        for (int i = 0; i < specs.length; i++) {
            String searchEngineName = specs[i].getName();
            out.println("<input type=\"radio\" " +
                "name=\"SearchEngine\" " + "value=\"" + searchEngineName + "\">\n");
            out.println(searchEngineName + "<br>\n");
        }
        out.println("<br> <input type=\"submit\">\n" +
            "</form>\n" + "</center></body></html>");
    }

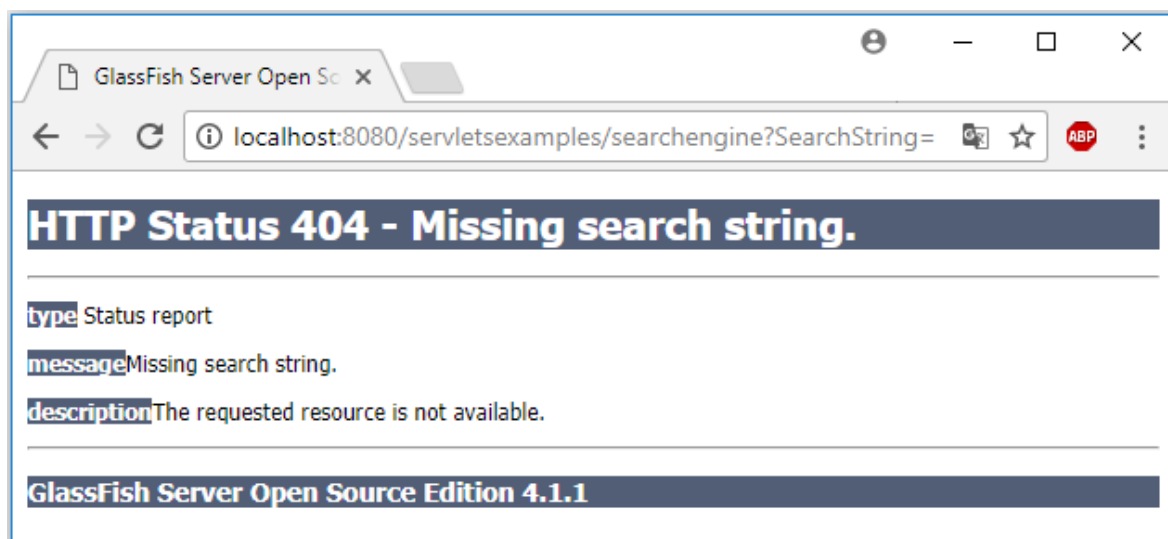
} // End of FormBuilderServlet
```



The screenshot shows a web browser window with the title 'One-Stop Web Search'. The address bar shows 'localhost:8080/servletexamples/formbuilder'. The main content area has the heading 'One-Stop Web Search' and a search form. The search keywords are 'homer simpson'. Below the search keywords, there is a list of search engines with radio buttons: Google (selected), Bing, Yahoo, Baidu, Ask, Aol, DuckDuckGo, WolframAlpha, and Yandex. At the bottom of the form is an 'Enviar' button.

Cuando el cliente envía los datos del usuario (cadena de búsqueda y motor de búsqueda) con este formulario generado, el servidor invoca el `servlet SearchEngineServlet.java`, que recibe estos parámetros y realiza lo siguiente:

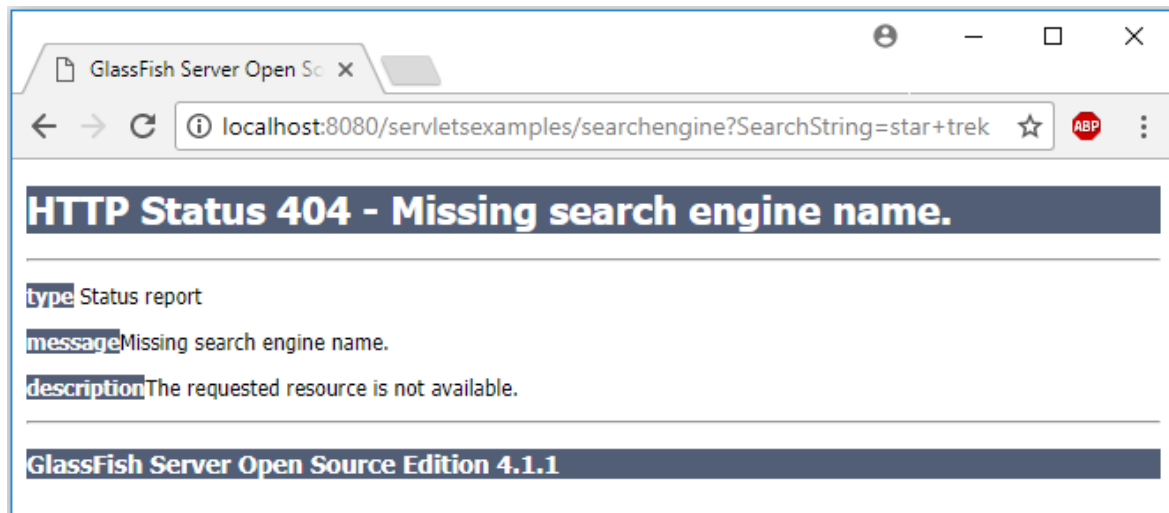
- Si los parámetros son válidos, se genera una URL que contiene el motor de búsqueda elegido y la cadena de búsqueda indicada y el navegador web se redirige hacia esa URL.
- Si el usuario no elige un motor de búsqueda o no indica una cadena a buscar, se muestra una página de error con un mensaje informativo.



The screenshot shows a web browser window with the title 'GlassFish Server Open Source Edition 4.1.1'. The address bar shows 'localhost:8080/servletexamples/searchengine?SearchString='. The main content area displays an HTTP 404 error message: 'HTTP Status 404 - Missing search string.' Below the error message, there is a table with the following information:

type	message	description
Status report	Missing search string.	The requested resource is not available.

At the bottom of the page, there is a footer that reads 'GlassFish Server Open Source Edition 4.1.1'.



```
package search;

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

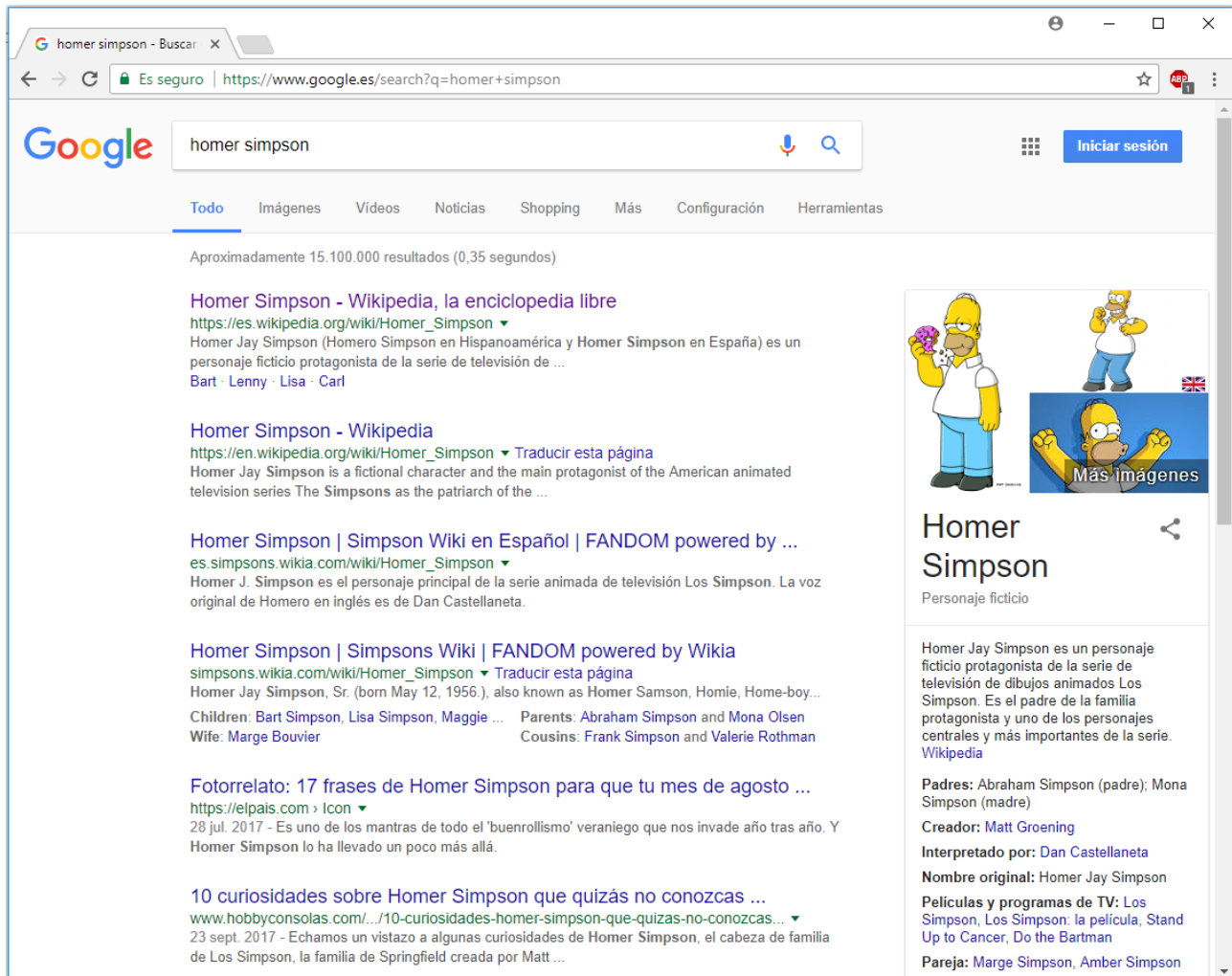
// Servlet that takes a search string and a search engine name and sends the query to
// that search engine.
// Illustrates manipulating the response status code. It sends a 302 response (via sendRedirect)
// if it gets a known search engine, and sends a 404 response (via sendError) otherwise.
public class SearchEngineServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String searchString = request.getParameter("SearchString");
        if ((searchString == null) || (searchString.length() == 0)) {
            reportProblem(response, "Missing search string.");
            return;
        }
        // The URLEncoder changes spaces to "+" signs and other non-alphanumeric characters to
        // "%XY", where XY is the hex value of the ASCII (or ISO Latin-1) character.
        // Browsers always URL-encode form values, and the getParameter method decodes automatica-
        // lly. But since we're just passing this on to another server, we need to re-encode it
        // to avoid characters that are illegal in URLs.
        // Also note that JDK 1.4 introduced a 2-argument version of URLEncoder.encode and deprecate
        // the 1-arg version. However, since version 2.3 of the servlet spec mandates only the
        // Java 2 Platform (JDK 1.2 or later), we stick with the 1-arg version for portability.
        searchString = URLEncoder.encode(searchString);
        String searchEngineName = request.getParameter("SearchEngine");
        if ((searchEngineName == null) || (searchEngineName.length() == 0)) {
            reportProblem(response, "Missing search engine name.");
            return;
        }
        String searchURL = SearchUtilities.makeURL(searchEngineName, searchString);
        if (searchURL != null) {
            response.sendRedirect(searchURL);
        }
        else {
            reportProblem(response, "Unrecognized search engine.");
        }
    }
}
```

```
private void reportProblem(HttpServletResponse response, String message)
throws IOException {
    response.sendError(response.SC_NOT_FOUND, message);
}

} // End of SearchEngineServlet
```

La siguiente captura muestra un ejemplo de ejecución de esta aplicación web con el motor de búsqueda Google y la cadena a buscar "homer simpson":



8. MANEJO DE COOKIES

Las **cookies** son pequeños ficheros de texto que un servidor web envía a un navegador web y que el propio navegador le devuelve posteriormente cuando se visita el mismo sitio o dominio. Al permitir que el servidor lea la información que el cliente envió anteriormente, el sitio web puede dar al usuario diversas facilidades como presentar la página web de la forma en que el usuario lo haya indicado antes, o dejar que los usuarios identificables entren sin tener que escribir una contraseña.

Para identificar a los usuarios de un sitio web, hay 3 pasos que suceden:

- 1) El *servlet* que se ejecuta en el servidor envía un conjunto de cookies al navegador web. Por ejemplo, nombre o número identificativo.
- 2) El navegador web almacena esta información en la máquina del cliente para un uso posterior.
- 3) Cada vez que el navegador web solicita una petición al servidor web, el cliente envía las cookies al servidor y el servidor utiliza esta información para identificar al usuario.

8.1. DETECCIÓN DE VISITANTES DE PÁGINAS WEB

El siguiente *servlet* **RepeatVisitorServlet.java** se puede utilizar para diferenciar entre los visitantes que acceden por primera vez a una página web y los visitantes que ya lo han hecho antes. Para ello, sólo se necesita almacenar en el cliente una *cookie*.

```
package cookies2;

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

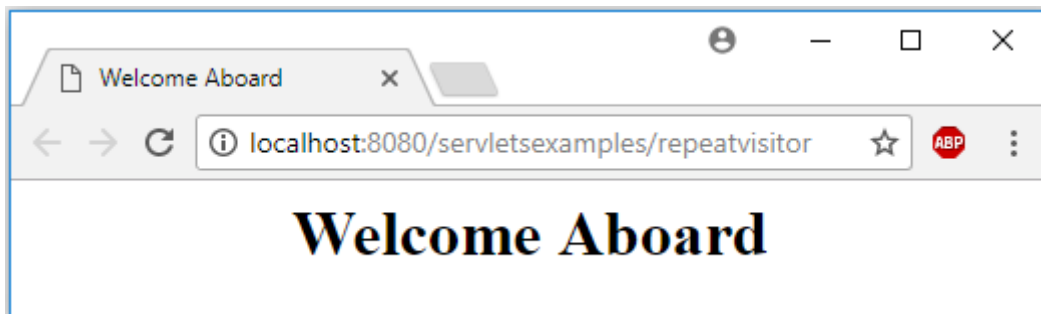
// Servlet that says "Welcome aboard" to first-time visitors and "Welcome back" to repeat visitors.
public class RepeatVisitorServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        boolean newbie = true;
        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for (int i = 0; i < cookies.length; i++) {
                Cookie c = cookies[i];
                if ((c.getName().equals("repeatVisitor")) && (c.getValue().equals("yes"))) {
                    newbie = false;
                    break;
                }
            }
        }
        String title;
        if (newbie) {
            Cookie returnVisitorCookie = new Cookie("repeatVisitor", "yes");
            returnVisitorCookie.setMaxAge(60*60*24*365); // 1 year
            response.addCookie(returnVisitorCookie);
            title = "Welcome Aboard";
        }
        else {
            title = "Welcome Back";
        }
    }
}
```

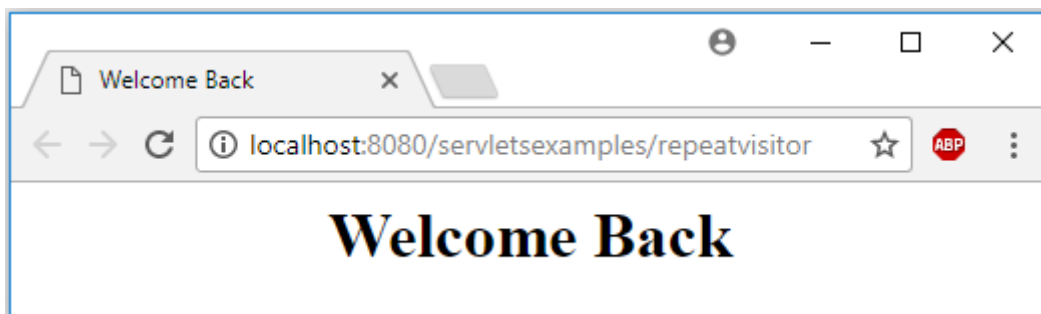
```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String docType = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " + "Transitional//EN">\n";
out.println(docType +
    "<html>\n" +
    "<head><title>" + title + "</title></head>\n" +
    "<body>\n" +
    "<h1 align=\"center\">" + title + "</h1>\n" +
    "</body></html>");
}

} // End of RepeatVisitorServlet
```

La primera visita de un cliente a la página web procesada por este *servlet* producirá esta salida:



Las siguientes visitas de un cliente a la página web procesada por este *servlet* producirán esta salida:



8.2. COOKIES DE SESIÓN Y COOKIES PERSISTENTES

Las *cookies* se comportan de manera diferente según si tienen establecido un tiempo de vida máximo o si no lo tienen indicado.

El siguiente *servlet* **CookieTestServlet.java** realiza dos tareas:

- 1) El *servlet* crea seis *cookies*:
 - Tres de ellas no tienen establecido un tiempo de vida máximo (por defecto, tienen un valor negativo de -1), lo que significa que únicamente serán válidas para la sesión actual del navegador. Estas son **cookies de sesión**.
 - Las otras tres utilizan el método `setMaxAge()` para indicarle al navegador web que almacene estas *cookies* en el disco y que se mantengan persistentes durante los siguientes 5 minutos, independientemente de si el usuario reinicia el navegador o reinicia el ordenador para abrir una nueva sesión del navegador. Estas son **cookies persistentes**.
- 2) El *servlet* utiliza el método `getCookies()` del objeto *request* para buscar todas las *cookies* y visualizar sus nombres y sus valores en una tabla HTML.

```

package cookies2;

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

// Creates a table of the cookies associated with the current page. Also sets 6 cookies:
//   3 that apply only to the current session (regardless of how long that session lasts)
//   and 3 that persist for an hour (regardless of whether the browser is restarted).
public class CookieTestServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        for (int i = 0; i < 3; i++) {
            // Default maxAge is -1, indicating cookie
            // applies only to current browsing session.
            Cookie cookie = new Cookie("Session-Cookie-" + i, "Cookie-Value-S" + i);
            response.addCookie(cookie);
            cookie = new Cookie("Persistent-Cookie-" + i, "Cookie-Value-P" + i);
            // Cookie is valid for 5 minutes, regardless of whether
            // user quits browser, reboots computer, or whatever.
            cookie.setMaxAge(300);
            response.addCookie(cookie);
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " + "Transitional//EN">\n";
        String title = "Active Cookies";
        out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body>\n" +
            "<h1 align=\"center\">" + title + "</h1>\n" +
            "<table border=1 align=\"center\">\n" +
            "<tr>\n" + "    <th>Cookie Name\n" + "    <th>Cookie Value");
        Cookie[] cookies = request.getCookies();
        if (cookies == null) {
            out.println("<tr><th colspan=2>No cookies");
        }
        else {
            for (Cookie cookie: cookies) {
                out.println("<tr>\n" +
                    "    <td>" + cookie.getName() + "\n" + "    <td>" + cookie.getValue());
            }
        }
        out.println("</table></body></html>");
    }

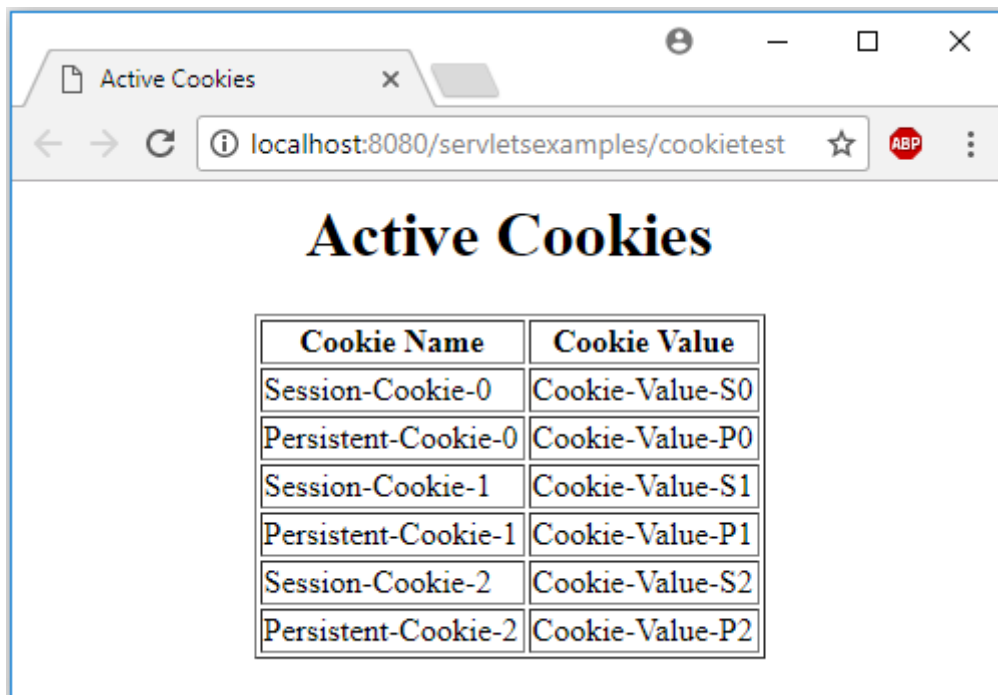
} // End of CookieTestServlet

```

La siguiente captura muestra el navegador del cliente, tras el procesamiento inicial del *servlet* *CookieTestServlet*. Este mismo resultado se produce también cuando se visita el *servlet*, se cierra el navegador, se espera 5 minutos y se vuelve a visitar el *servlet*.



La siguiente captura muestra el navegador del cliente cuando se vuelve a visitar el *servlet* *CookieTestServlet* dentro de los siguientes 5 minutos tras la primera visita, en la misma sesión del navegador (el navegador ha permanecido abierto entre la visita anterior y la visita indicada aquí).



La siguiente captura muestra el navegador del cliente cuando se vuelve a visitar el *servlet* *CookieTestServlet* dentro de los siguientes 5 minutos tras la primera visita, en una sesión diferente del navegador (se ha reiniciado el navegador entre la visita inicial y la visita indicada aquí).



8.3. MODIFICACIÓN DE COOKIES

Para modificar el valor de una cookie previa, se incluye en la cabecera de respuesta una *cookie* con el mismo nombre y un valor diferente.

Para indicar al navegador web que elimine una cookie, se usa el método *setMaxAge()* con un valor de tiempo de vida máximo de 0.

El siguiente *servlet* **ClientAccessCountServlet.java** sigue el rastro de las veces que cada cliente web ha visitado la página web. Para ello, el *servlet* obtiene del mensaje de petición (*request*) una cookie cuyo nombre es `accessCount` y cuyo valor muestra la cuenta actual, y añade al mensaje de respuesta (*response*) una *cookie* con el mismo nombre y con el valor actualizado.


```

package cookies2;

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

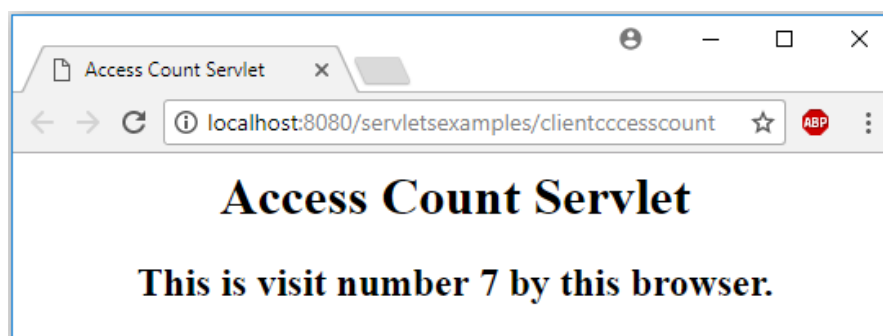
// Servlet that prints per-client access counts.
public class ClientAccessCountServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String countString = CookieUtilities.getCookieValue(request, "accessCount", "1");
        int count = 1;
        try {
            count = Integer.parseInt(countString);
        }
        catch (NumberFormatException nfe) { }
        LongLivedCookie c = new LongLivedCookie("accessCount", String.valueOf(count+1));
        response.addCookie(c);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Access Count Servlet";
        String docType = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " + "Transitional//EN">\n";
        out.println(docType +
            "<html>\n" + "<head><title>" + title + "</title></head>\n" +
            "<body>\n" + "<center>\n" +
            "<h1>" + title + "</h1>\n" +
            "<h2>This is visit number " + count + " by this browser.</h2>\n" +
            "</center></body></html>");
    }

} // End of ClientAccessCountServlet

```

Cada usuario sólo verá sus propios contadores de acceso. Los navegadores web mantienen las cookies de forma separada. En este caso, el mismo usuario observa dos contadores de acceso a la página web independientes en dos navegadores web (Edge y Chrome).



9. RASTREO DE SESIONES

HTTP es un protocolo sin estado: cada vez que un cliente obtiene una página web, el cliente abre una conexión por separado al servidor y el servidor no conserva ninguna información contextual respecto a la petición del cliente. Incluso si el servidor permite el uso de conexiones HTTP persistentes y mantiene un *socket* abierto para diversas peticiones del cliente, no hay un apoyo integrado para conservar información del contexto del cliente.

Hay 3 soluciones típicas para este problema:

- 1) **Cookies.** El servidor web puede asignar un identificador de sesión único a cada cliente web en una *cookie*, para que las peticiones realizadas por ese cliente se puedan reconocer en el servidor mediante la *cookie* recibida. En este caso el servidor tiene que:
 - Obtener la cookie que contiene el identificador de sesión.
 - Establecer un plazo de caducidad adecuado para la cookie.
 - Asociar los datos del cliente en cada la petición.
 - Generar identificadores de sesión únicos.

Sin embargo, muchos navegadores no soportan *cookies* o algunos usuarios las inhabilitan. Además, sería necesario contar con una API de nivel superior que trate estos detalles.

- 2) **Campos Ocultos de un Formulario.** El servidor web puede enviar un formulario HTML con un campo oculto que indica un identificador de sesión único:

```
<input type="hidden" name="session_id" value="123456">
```

Este elemento significa que, cuando se envíe el formulario, el nombre y el valor especificados se incluyen de forma automática en los datos GET o POST. Este campo oculto se puede utilizar para almacenar información respecto a la sesión, pero sólo funciona si todas las páginas se generan de forma dinámica.

- 3) **Reescritura de la URL.** El cliente agrega algunos datos al final de la URL que identifican a la sesión, y el servidor asocia ese identificador de sesión con los datos de la sesión que tiene almacenados. Por ejemplo:

```
http://host/ruta/archivo.html;idsesion=123456
```

Esta solución funciona cuando el navegador web no permite el uso de *cookies* o cuando el usuario las haya inhabilitado. Pero el servidor tiene que realizar mucho procesamiento directo y tedioso (generar cada URL dinámicamente para asignarle un identificador de sesión, incluso en el caso de una página HTML estática).

9.1. RASTREO DE SESIONES CON SERVLETS

Los *servlets* proporcionan la interfaz **HttpSession** para el rastreo de sesiones. Esta interfaz de alto nivel está basada en el manejo de cookies o la reescritura de la URL. Con ella, el programador del *servlet* no tiene que manipular explícitamente cookies o la información añadida a la URL, y dispone de métodos adecuados para almacenar objetos arbitrarios asociados con cada sesión.

El uso de sesiones implica realizar 4 pasos:

- 1) **Acceder al objeto de sesión asociado con la petición actual.** Los objetos de sesión son tablas hash que pueden almacenar objetos arbitrarios del usuario. Por debajo, el sistema obtiene un identificador de usuario de una cookie o de datos adjuntos a la URL, y lo utiliza como clave para buscar un objeto de sesión.

```
HttpSession session = request.getSession(true);
```

Si no se encuentra un identificador de sesión, el sistema crea una sesión nueva y vacía. Además, se puede usar el método `isNew()` para saber si la sesión ya existía o es nueva.

Si el navegador web soporta cookies (situación por defecto), el sistema también crea una cookie saliente con el nombre "JSESSIONID" y con un valor único que representa el identificador de sesión.

Por otro lado, si solamente se desea mostrar la información de una sesión, no se requiere crear una nueva sesión si no existe, por lo que la sesión se procesa de la siguiente forma:

```
HttpSession session = request.getSession(false);
if (session == null) {
    printMessage("Session is null.");
}
else {
    printContent(session);
}
```

- 2) **Buscar la información asociada a la sesión.** Los objetos `HttpSession` residen en el servidor y se asocian de forma automática con el cliente usando mecanismos de cookies o de reescritura de URL. Estos objetos de sesión tienen una estructura de datos incorporada (tabla *hash*) que permite almacenar cualquier cantidad de claves y sus valores correspondientes.

```
HttpSession session = request.getSession();
SomeClass value =
    (SomeClass) session.getAttribute("someIdentifier");
if (value == null) {
    value = new SomeClass(...);
    session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

Para obtener todos los nombres de los atributos almacenados en una sesión, se puede utilizar el método `getAttributeNames()`, que devuelve un objeto *Enumeration*.

- 3) **Almacenar información en la sesión.** Como se ha visto previamente, la información asociada a una sesión se lee con el método `getAttribute()`.

Para especificar información en una sesión, se utiliza el método `setAttribute()`, indicando clave y valor. Este método reemplaza cualquier valor anterior. Si se desea que el valor realice efectos laterales cuando se almacene en una sesión, el objeto asociado a la sesión tendrá que implementar la interfaz `HttpSessionBindingListener`. Así, cuando se ejecute `setAttribute()` en este objeto, se llamará después al método `valueBound()`.

Para eliminar un valor sin proporcionar un reemplazo, se utiliza el método `removeAttribute()`. Este método disparará la ejecución del método `valueUnbound()` de cualquier objeto que implemente la interfaz `HttpSessionBindingListener`.

- 4) **Descartar los datos de la sesión.** Cuando se ha terminado de procesar los datos de la sesión de usuario, hay varias formas de finalizar la sesión:

- *Eliminar sólo los datos que el servlet ha creado.* Se puede llamar al método `removeAttribute()` para descartar los valores asociados a las claves especificadas.
- *Borrar la sesión entera (en la aplicación web actual).* Se puede llamar al método `invalidate()` para descartar una sesión entera. Esto provocará la pérdida de todos los datos de esa sesión del usuario, no sólo los datos de la sesión que el *servlet* ha creado.

- *Desconectar al usuario y borrar todas sus sesiones.* Se puede llamar al método `logout()` para desconectar al cliente del servidor web e invalidar todas las sesiones (como máximo una por aplicación web) asociadas con dicho usuario.

Para más información, consultar la documentación de las interfaces:

<http://docs.oracle.com/javaee/6/api/index.html?javax/servlet/http/HttpSession.html>
<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSessionBindingListener.html>

9.2. SESIONES DE NAVEGADOR Y SESIONES DE SERVIDOR

Por defecto, el rastreo de sesiones se basa en *cookies* que se almacenan en la memoria del navegador, no se guardan en disco. Así, a no ser que el *servlet* lea la cookie `"JSESSIONID"`, establezca un tiempo de vida máximo y la devuelva en el mensaje de respuesta, cerrar el navegador web provocará que la **sesión de navegador** desaparezca: el cliente no podrá acceder a ella.

Sin embargo, en este caso el servidor no sabe que el navegador web ha sido cerrado por el usuario, por lo que tiene que mantener la **sesión de servidor** en memoria hasta que el tiempo máximo de inactividad haya transcurrido.

Automáticamente, una sesión se vuelve **inactiva** cuando el intervalo de tiempo entre los accesos del cliente supera el intervalo de tiempo especificado por el método `getMaxInactiveInterval()`. Cuando esto sucede, los objetos almacenados en el objeto `HttpSession` se eliminan, y si estos objetos implementan la interfaz `HttpSessionBindingListener`, se les notifica de forma automática.

Cuando se invoca el método `invalidate()` de un objeto *sesión* o cuando se invoca el método `logout()` de un objeto *request*, el servidor no espera a que la sesión expire, sino que inmediatamente elimina todos los objetos de la sesión y destruye el objeto sesión.

9.3. CIFRADO DE LAS URL ENVIADAS AL CLIENTE

Por defecto, el contenedor de *servlets* utiliza *cookies* como mecanismo subyacente para el rastreo de sesiones. Si se configura el servidor para usar la reescritura de URL en el rastreo de sesiones, se tiene que añadir explícitamente los datos de la sesión a cada URL. Por tanto, la mayoría o todas las páginas web se tendrán que generar de forma dinámica. No se pueden tener páginas HTML estáticas que contengan enlaces a páginas dinámicas.

Hay dos situaciones en las que se pueden utilizar URL que enlazan a una página web:

- 1) **La URL está embebida en la página web que el *servlet* genera.** Esta URL se debe pasar al método `encodeURL()` del objeto `HttpServletResponse`. Este método determina si la reescritura de URL se usa actualmente y añade la información de la sesión si es necesario. En caso contrario, se devuelve la URL sin modificar.

```
String originalURL = "..."; // URL relativa o absoluta
String encodedURL = response.encodeURL(originalURL);
out.println("<a href=\"" + encodedURL + "\">...</a>");
```

- 2) **La URL se encuentra en la cabecera de respuesta *location*.** Esta URL se debe pasar al método `sendRedirect()` del objeto `HttpServletResponse`, pero antes hay que determinar si se debe adjuntar la información de la sesión con el método `encodeRedirectURL()`.

```
String originalURL = "..."; // URL relativa
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

9.4. CREACIÓN DE UNA LISTA DE DATOS DEL USUARIO

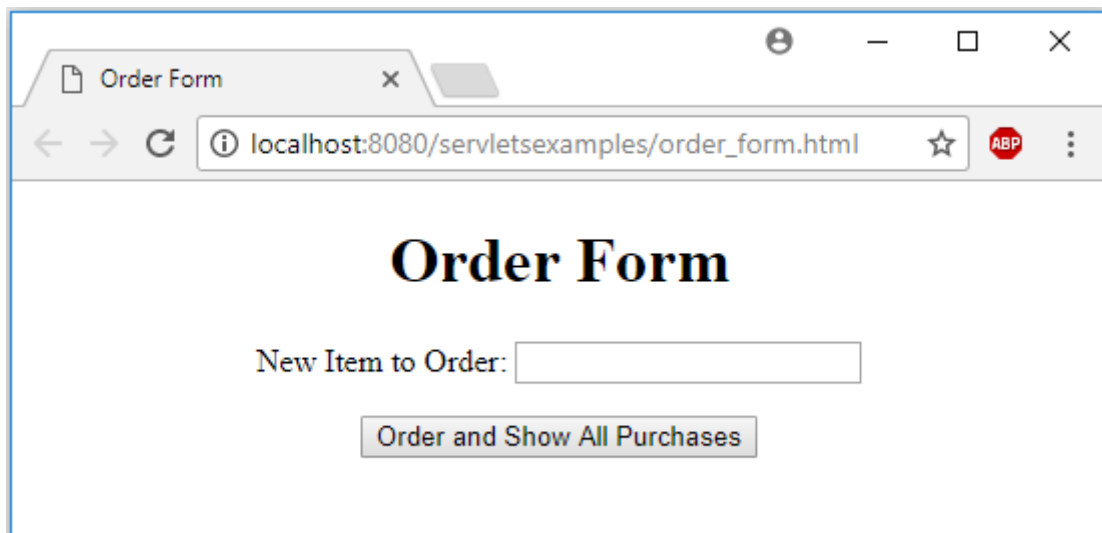
Se puede utilizar **estructuras de datos mutables**, como vectores, listas o mapas, que tengan variables instancias en forma de campos que se pueden escribir. En este caso, no hace falta llamar al método `setAttribute()`, excepto cuando el objeto se crea en memoria por primera vez.

```
HttpSession session = request.getSession();
MutableClass value =
    (MutableClass) session.getAttribute("someIdentifier");
if (value == null) {
    value = new MutableClass(...);
    session.setAttribute("someIdentifier", value);
}
value.updateInternalState(...);
doSomethingWith(value);
```

Se suelen utilizar estructuras de datos mutables para mantener un conjunto de datos asociado con el usuario.

El siguiente formulario **order_form.html** recoge un valor de un ítem solicitado por el usuario y se lo envía al servidor.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Order Form</title>
  </head>
  <body>
    <center>
      <h1>Order Form</h1>
      <form action="showitemsservlet">
        New Item to Order:
        <input type="text" name="newItem" value="Yacht"><P>
        <input type="submit" value="Order and Show All Purchases">
      </form>
    </center>
  </body>
</html>
```



El siguiente *servlet* **ShowItemsServlet.java** utiliza un *ArrayList* para acumular cada ítem y muestra esta lista de ítems solicitados por el cliente en el navegador web.

```
package session;

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

@SuppressWarnings("unchecked")
public class ShowItemsServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession();
        ArrayList<String> previousItems =
            (ArrayList<String>) session.getAttribute("previousItems");
        if (previousItems == null) {
            previousItems = new ArrayList<String>();
            session.setAttribute("previousItems", previousItems);
        }
        String newItem = request.getParameter("newItem");
        if ((newItem != null) && (!newItem.trim().equals(""))) {
            previousItems.add(newItem);
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Items Purchased";
        String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" + \"Transitional//EN\">\n";
        out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body>\n" +
            "<h1>" + title + "</h1>");
        if (previousItems.size() == 0) {
            out.println("<i>No items</i>");
        }
        else {
            out.println("<ul>");
            for (String item: previousItems) {
                out.println("  <li>" + item);
            }
            out.println("</ul>");
        }
        out.println("</body></html>");
    }

} // End of ShowItemsServlet
```

9.5. TIENDA VIRTUAL CON CARRO DE COMPRA

El siguiente caso práctico muestra el proceso de construcción de una tienda virtual (online) que utiliza rastreo de sesiones. Para ello, se ha dividido el desarrollo de la aplicación en 2 partes:

- 1) Generación de la página web que muestra al cliente los artículos a la venta. Incluye las clases que representan los artículos individuales y el catálogo de artículos a la venta.
- 2) Generación de la página web que muestra al cliente el pedido realizado. Incluye las clases que representan el carro de compra y el pedido del cliente.

9.5.1. Generación del Catálogo de Artículos de Venta

El siguiente *servlet* **CatalogPage.java** es una clase abstracta que se usa como punto de partida para futuros *servlets* que visualicen artículos determinados a la venta. Este *servlet* obtiene los identificadores de los artículos a la venta, los busca en el catálogo y utiliza sus descripciones y precios encontrados para visualizar la página web del catálogo de artículos en el cliente.

```
package session2;

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

// Base class for pages showing catalog entries.
// Servlets that extend this class must specify the catalog entries that they are selling and
// the page title before the servlet is ever accessed. This is done by putting calls to
// setItems and setTitle in init.
public abstract class CatalogPage extends HttpServlet {

    private CatalogItem[] items;
    private String[] itemIDs;
    private String title;

    // Given an array of item IDs, look them up in the Catalog and put their corresponding
    // CatalogItem entry into the items array.
    // The CatalogItem contains a short description, a long description, and a price, using
    // the item ID as the unique key.
    // Servlets that extend CatalogPage must call this method (usually from init) before the
    // servlet is accessed.
    protected void setItems(String[] itemIDs) {
        this.itemIDs = itemIDs;
        items = new CatalogItem[itemIDs.length];
        for (int i = 0; i < items.length; i++) {
            items[i] = Catalog.getItem(itemIDs[i]);
        }
    }

    // Sets the page title, which is displayed in an H1 heading in resultant page.
    // Servlets that extend CatalogPage must call this method (usually from init) before the
    // servlet is accessed.
    protected void setTitle(String title) {
        this.title = title;
    }

    // First display title, then, for each catalog item, put its short description in a
    // level-two (H2) heading with the price in parentheses and long description below.
    // Below each entry, put an order button that submits info to the OrderPage servlet for
    // the associated catalog entry.
    // To see the HTML that results from this method, do "View Source" on KidsBooksPage or
    // TechBooksPage, two concrete classes that extend this abstract class.
```

```

@Override
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    if (items == null) {
        response.sendError(response.SC_NOT_FOUND, "Missing Items.");
        return;
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " + "Transitional//EN">\n";
    out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body>\n" +
        "<h1 align='center'>" + title + "</h1>");
    CatalogItem item;
    for (int i = 0; i < items.length; i++) {
        out.println("<hr>");
        item = items[i];
        // Show error message if subclass lists item ID that's not in the catalog.
        if (item == null) {
            out.println("<font color='red'>" + "Unknown item ID " + itemIDs[i] + "</font>");
        }
        else {
            out.println();
            String formURL = "orderpage";
            // Pass URLs that reference own site through encodeURL.
            formURL = response.encodeURL(formURL);
            out.println
                ("<form action='" + formURL + "'>\n" +
                 "<input type='hidden' name='itemID' value='" + item.getItemID() + "'>\n" +
                 "<h2>" + item.getShortDescription() + " ($" + item.getCost() + "</h2>\n" +
                 item.getLongDescription() + "\n" +
                 "<p>\n<center>\n" +
                 "<input type='submit' value='Add to Shopping Cart'>\n" +
                 "</center>\n<p>\n</form>");
        }
    }
    out.println("<hr>\n</body></html>");
}

} // End of CatalogPage

```

Por ejemplo, a partir de este *servlet* base *CatalogPage.java*, el *servlet* **KidsBooksPage.java** visualiza el catálogo de libros infantiles disponibles a la venta en el navegador web.

```

package session2;

// A specialization of the CatalogPage servlet that displays a page selling three famous
// kids-book series.
// Orders are sent to the OrderPage servlet.
public class KidsBooksPage extends CatalogPage {

    public void init() {
        String[] ids = { "lewis001", "alexander001", "rowling001" };
        setItems(ids);
        setTitle("All-Time Best Children's Fantasy Books");
    }

}

} // End of KidsBooksPage

```


All-Time Best Children's F x

localhost:8080/servletexamples/kidsbookspage

☆

ABP

All-Time Best Children's Fantasy Books

The Chronicles of Narnia by C.S. Lewis (\$49.9)

The classic children's adventure pitting Aslan the Great Lion and his followers against the White Witch and the forces of evil. Dragons, magicians, quests, and talking animals wound around a deep spiritual allegory. Series includes *The Magician's Nephew*, *The Lion, the Witch and the Wardrobe*, *The Horse and His Boy*, *Prince Caspian*, *The Voyage of the Dawn Treader*, *The Silver Chair*, and *The Last Battle*.

Add to Shopping Cart

The Prydain Series by Lloyd Alexander (\$29.9)

Humble pig-keeper Taran joins mighty Lord Gwydion in his battle against Arawn the Lord of Annuvn. Joined by his loyal friends the beautiful princess Eilonwy, wannabe bard Fflewddur Fflam, and furry half-man Gurgi, Taran discovers courage, nobility, and other values along the way. Series includes *The Book of Three*, *The Black Cauldron*, *The Castle of Llyr*, *Taran Wanderer*, and *The High King*.

Add to Shopping Cart

The Harry Potter Series by J.K. Rowling (\$69.9)

The seven popular stories about wizard-in-training Harry Potter topped both the adult and children's best-seller lists. Series includes *Harry Potter and the Sorcerer's Stone*, *Harry Potter and the Chamber of Secrets*, *Harry Potter and the Prisoner of Azkaban*, *Harry Potter and the Goblet of Fire*, and *Harry Potter and the Order of the Phoenix*. *Harry Potter and the Half-Blood Prince*, and *Harry Potter and the Deathly Hallows*.

Add to Shopping Cart

9.5.2. Generación del Pedido del Cliente

El siguiente *servlet* **OrderPage.java** gestiona los pedidos realizados en las diferentes páginas de catálogos. Utiliza el rastreo de sesiones para asociar un carro de compra a cada usuario.

```
package session2;

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

// Shows all items currently in ShoppingCart.
// Clients have their own session that keeps track of which ShoppingCart is theirs. If this is
// their first visit to the order page, a new shopping cart is created.
// Usually, people come to this page by way of a page showing catalog entries, so this page adds
// an additional item to the shopping cart. But users can also bookmark this page, access it from
// their history list, or be sent back to it by clicking on the "Update Order" button after
// changing the number of items ordered.
public class OrderPage extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession();
        ShoppingCart cart;
        synchronized(session) {
            cart = (ShoppingCart) session.getAttribute("shoppingCart");
            // New visitors get a fresh shopping cart.
            // Previous visitors keep using their existing cart.
            if (cart == null) {
                cart = new ShoppingCart();
                session.setAttribute("shoppingCart", cart);
            }
            String itemID = request.getParameter("itemID");
            if (itemID != null) {
                String numItemsString = request.getParameter("numItems");
                if (numItemsString == null) {
                    // If request specified an ID but no number, then customers came here
                    // via an "Add Item to Cart" button on a catalog page.
                    cart.addItem(itemID);
                }
                else {
                    // If request specified an ID and number, then customers came here
                    // via an "Update Order" button after changing the number of items in order.
                    // Note that specifying a number of 0 results in item being deleted from cart.
                    int numItems;
                    try {
                        numItems = Integer.parseInt(numItemsString);
                    }
                    catch (NumberFormatException nfe) {
                        numItems = 1;
                    }
                    cart.setNumOrdered(itemID, numItems);
                }
            }
        }
    }
}
```

Dado que cada usuario tiene una sesión separada, es improbable que múltiples hilos de ejecución o *threads* accedan al mismo carro de compra simultáneamente. Sin embargo, por ejemplo, esto podría ocurrir si un usuario tuviera varias ventanas del navegador abiertas y enviara actualizaciones desde más de una en un breve intervalo de tiempo.

Así que, por seguridad, el código sincroniza el acceso basado en el objeto `session`. Esta sincronización evita que otros *threads* utilicen la misma sesión para acceder a los datos de forma concurrente, mientras que permite que varias peticiones simultáneas de diferentes usuarios se procesen.

```
// Whether or not the customer changed the order, show order status.
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "Status of Your Order";
String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" + \"Transitional//EN\">\n";
out.println(docType +
    "<html>\n" + "<head><title>" + title + "</title></head>\n" +
    "<body>\n" + "<h1 align=\"center\">" + title + "</h1>");

synchronized(session) {
    List itemsOrdered = cart.getItemsOrdered();
    if (itemsOrdered.size() == 0) {
        out.println("<h2><i>No items in your cart...</i></h2>");
    }
    else {
        // If there is at least one item in cart, show table of items ordered.
        out.println("<table border=1 align=\"center\">\n" + "<tr>\n" +
            "  <th>Item ID<th>Description\n" + "  <th>Unit Cost<th>Number<th>Total Cost");
        ItemOrder order;
        // Rounds to two decimal places, inserts dollar sign (or other currency symbol),
        // as appropriate in current Locale.
        NumberFormat formatter = NumberFormat.getCurrencyInstance();
        // For each entry in shopping cart, make table row showing ID, description,
        // per-item cost, number ordered, and total cost.
        // Put number ordered in textfield that user can change, with "Update Order" button
        // next to it, which resubmits to this same page but specifying a different number
        // of items.

        for (int i = 0; i < itemsOrdered.size(); i++) {
            order = (ItemOrder) itemsOrdered.get(i);
            out.println
                ("<tr>\n" +
                 "  <td>" + order.getItemID() + "\n" +
                 "  <td>" + order.getShortDescription() + "\n" +
                 "  <td>" + formatter.format(order.getUnitCost()) + "\n" +
                 "  <td>" + "<form>\n" + // Submit to current URL
                 "<input type=\"hidden\" name=\"itemID\" value=\"" +
                 order.getItemID() + "\">\n" +
                 "<input type=\"text\" name=\"numItems\" \n size=3 value=\"" +
                 order.getNumItems() + "\">\n" +
                 "<small>\n" +
                 "<input type=\"submit\" value=\"Update Order\">\n" +
                 "</small>\n" + "</form>\n" +
                 "  <td>" + formatter.format(order.getTotalCost()));
        }
        String checkoutURL = response.encodeURL("checkout.html");
        // "Proceed to Checkout" button below table
        out.println
            ("</table>\n" +
             "<form action=\"" + checkoutURL + "\">\n" + "<big><center>\n" +
             "<input type=\"submit\" value=\"Proceed to Checkout\">\n" +
             "</center></big></form>");
    }
    out.println("</body></html>");
}

} // End of OrderPage
```

La siguiente captura del navegador muestra la página web del pedido del cliente, tras haber añadido un artículo del catálogo al carro de compra.

Item ID	Description	Unit Cost	Number	Total Cost
rowling001	<i>The Harry Potter Series</i> by J.K. Rowling	69,90 ?	<input type="text" value="1"/> <input type="button" value="Update Order"/>	69,90 ?

La siguiente captura del navegador muestra la página web del pedido del cliente, tras haber añadido varios artículos del catálogo al carro de compra y haber modificado la cantidad de los artículos del pedido.

Item ID	Description	Unit Cost	Number	Total Cost
rowling001	<i>The Harry Potter Series</i> by J.K. Rowling	69,90 ?	<input type="text" value="2"/> <input type="button" value="Update Order"/>	139,80 ?
lewis001	<i>The Chronicles of Narnia</i> by C.S. Lewis	49,90 ?	<input type="text" value="2"/> <input type="button" value="Update Order"/>	99,80 ?