



# Java™

Desarrollo de Aplicaciones Web con JEE

## **PARTE I**

- **SERVLETS**

## 2. Servlets

Los servlets permiten **seguir la trayectoria de un cliente**, es decir, obtener y mantener una determinada información acerca del cliente. De esta forma se puede **tener identificado a un cliente durante un determinado tiempo**. Esto es muy importante si se quiere disponer de aplicaciones que impliquen la ejecución de varios servlets o la ejecución repetida de un mismo servlet. Un claro ejemplo de aplicación de esta técnica es el de los **comercios vía Internet** que permiten llevar un carrito de la compra en el que se van guardando aquellos productos solicitados por el cliente. El cliente puede ir navegando por las distintas secciones del comercio virtual, es decir **realizando distintas conexiones HTTP y ejecutando diversos servlets**, y a pesar de ello no se pierde la información contenida en el carrito de la compra y se sabe en todo momento que es un mismo cliente quien está haciendo esas conexiones diferentes.

El mantener información sobre un cliente a lo largo de un proceso que implica múltiples conexiones se puede realizar de formas distintas:

- ✓ Paso de parámetros en formulario (Request)
- ✓ Mediante cookies
- ✓ Mediante seguimiento de sesiones (Session Tracking)
- ✓ Mediante la reescritura de URLs

En definitiva, los Servlets **son pequeños programas Java que se ejecutan en un servidor de aplicaciones o en un contenedor de Servlets y que se comunican con el cliente mediante el protocolo HTTP<sup>1</sup>**. Estos programas suelen ser intermediarios entre el cliente (la mayoría de las veces un navegador web) y los datos (normalmente una base de datos).

El funcionamiento básico de un Servlet es el siguiente:

- ✓ El cliente envía una petición al Servlet mediante HTTP
- ✓ El Servlet procesa los argumentos de la petición
- ✓ El Servlet realiza la tarea solicitada
- ✓ El Servlet devuelve una respuesta al cliente en HTTP

El trabajo con Servlets es muy cómodo, ya que **es el servidor de aplicaciones el encargado de crear distintos hilos de ejecución para un mismo Servlet si se realizan varias peticiones de forma**

---

<sup>1</sup> [HTTP](#)

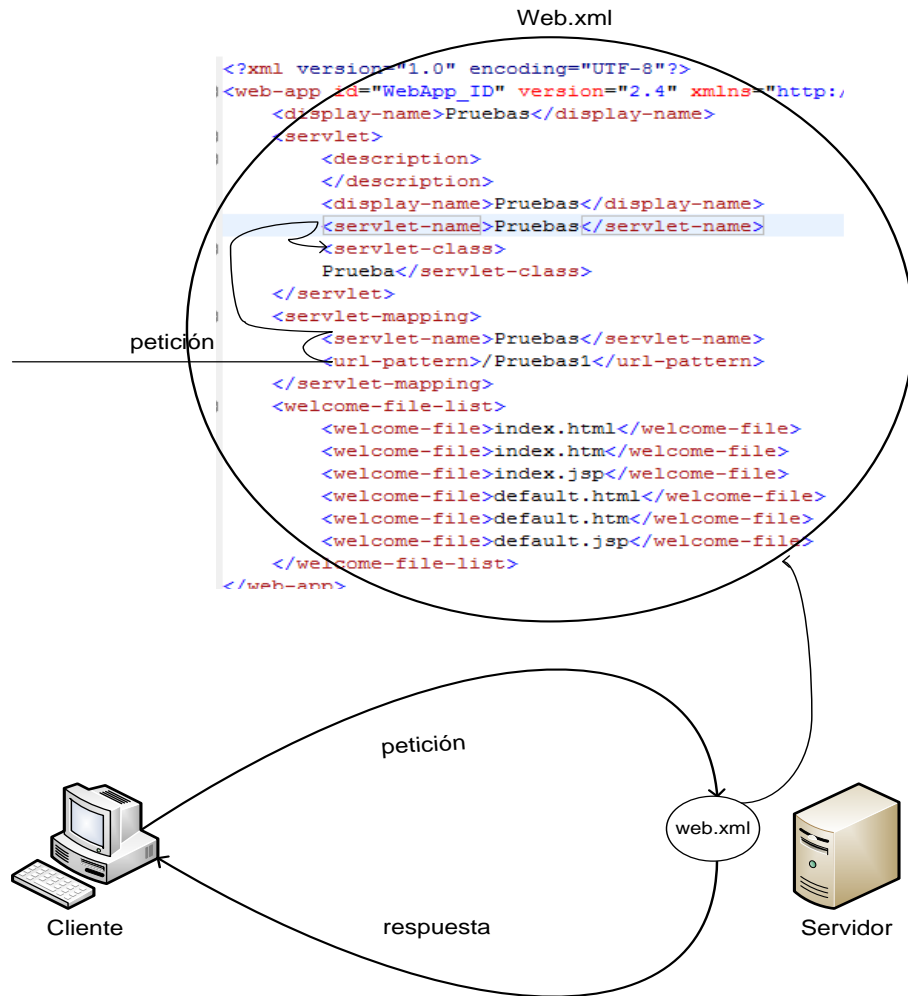
**simultánea.** También el servidor de aplicaciones es el encargado de crear un entorno seguro en el que se ejecutarán los Servlets.

## 2.1 Estructura básica de una aplicación utilizando servlets

La estructura en directorios y ficheros depende del servidor de aplicaciones seleccionado, en nuestro caso Tomcat.

Lo primero es localizar el directorio en el que deben instalarse las aplicaciones. En el caso de Tomcat deben instalarse en la carpeta **webapps**. Dentro de dicha carpeta debemos crear una carpeta para cada aplicación web que se pretenda instalar sobre Tomcat. Dentro de esta carpeta se copiarán todos los archivos html, gif, css, etc., que formen las páginas web estáticas de la aplicación, respetando la estructura de directorios que más nos convenga para su mantenimiento, es decir, no es obligatorio copiar todos los archivos en el raíz de la carpeta de la aplicación.

En esa carpeta raíz de la aplicación debemos crear una carpeta que debe llamarse **WEB-INF**. Esta carpeta contendrá un archivo **web.xml**, **el descriptor de despliegue de nuestra aplicación, es un archivo escrito en XML que describe diversas características de la aplicación web**. Contiene entradas, como parámetros de inicialización, servlets y otros componentes. Podemos ignorar este archivo si utilizamos **anotaciones** en nuestros servlets. Esto es una de las nuevas características a partir de la plataforma Java EE 6 y de la especificación 3 de los servlets. Para poder utilizar las anotaciones debemos de utilizar un contenedor servlet que implemente la especificación indicada, sino es así debemos de utilizar el descriptor de despliegue. Para poder utilizar las anotaciones debemos de importar el paquete annotations y utilizar la anotación **@WebServlet("URL-Pattern")**.



También contiene una carpeta **lib** para las librerías .jar necesarias para ejecución de la aplicación. Dentro de la carpeta **build** se encuentra una carpeta **classes** con los archivos .class correspondientes a la compilación de los Servlets.

También puede incluirse una carpeta **src** con el código fuente de los Servlets. Cuando desarrollamos servlets no es buena idea colocar todos en el mismo directorio dentro de la jerarquía de directorios de una aplicación Web. Es una buena práctica empaquetar los servlets en **packages** para evitar conflictos y poder gestionarlos más fácilmente.

Para empaquetar un servlet en una aplicación Web desde Eclipse basta con pinchar con el botón derecho del ratón sobre la carpeta **Java Resources->src** y elegir la opción **New -> Package**. Después situamos todas las clases de nuestros ejemplos en este paquete. El IDE Eclipse ya se encarga a la hora de

compilar y desplegar de crear el directorio que representa ese paquete tanto en el código fuente como en la carpeta **classes**.

Para ilustrar todo esto vamos a implementar el programa HolaMundo de diferentes formas. Para ello crearemos un proyecto nuevo llamado HolaMundo.

### 2.1.1 Ejemplo1

Vamos a ver cómo crear nuestro primer servlet llamado Ejemplo1. En este caso el servlet generará como salida texto plano con la cadena Hola Mundo.

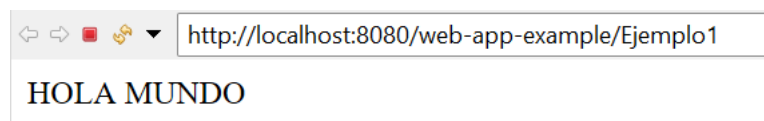
```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * Servlet implementation class Ejemplo1
 */
public class Ejemplo1 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo1() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
        response.getWriter().append("HOLA MUNDO");
    }
}
```

A continuación, se arranca Tomcat y desde el browser se lanza la app y aparecería en la pantalla lo siguiente la llamada directa al servlet:



### 2.1.2 Ejemplo2

Un servlet que genera código HTML Para que el servlet genere HTML tendremos que decirle al navegador que le vamos a enviar texto HTML.

```
package servlets;

import jakarta.servlet.ServletException;

/**
 * Servlet implementation class Ejemplo2
 */
@WebServlet("/Ejemplo2")
public class Ejemplo2 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo2() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)
     */
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
        response.setContentType("text/html");
        response.getWriter().append("<html>")
            .append("<body>")
            .append("<h1>EJEMPLO2</h1>")
            .append("<br>HOLA MUNDO")
            .append("</body>")
            .append("</html>");
        response.getWriter().close();
    }
}
```

A continuación, se arranca Tomcat y aparecería en la pantalla:



## 2.2 Ciclo de vida de un servlet

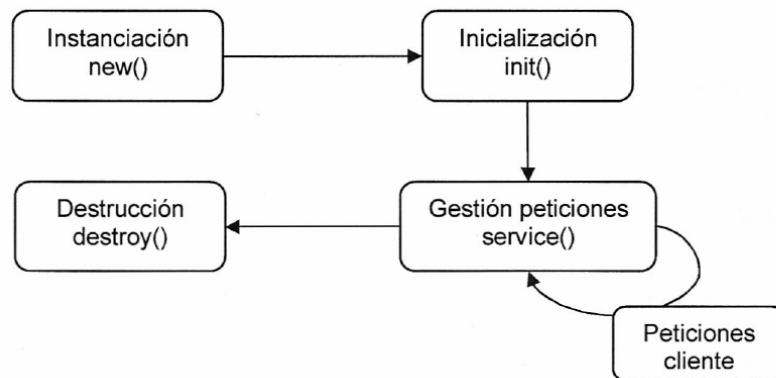
Un servlet es un objeto que se ejecuta en un contenedor web, fue diseñado para servir páginas dinámicas web. **Un servlet se compila una única vez y se aloja en el servidor a la espera de que lleguen peticiones desde los navegadores clientes.** Solamente se inicializa una vez, y posteriormente por cada llamada que se realice al servidor de aplicaciones, éste ejecutará el método que corresponda.

El ciclo de vida de un servlet tiene tres fases fundamentales:

- ✓ **Instanciación e inicialización:** se crea el servlet en memoria y se llama al método `ini()` para inicializar su funcionamiento.
- ✓ **Gestión de peticiones:** para atender cada petición, **el servidor de aplicaciones o contenedor de servlets crea un hilo de ejecución del servlet, lo que permite atender varias peticiones de**

**forma simultánea**, compartir datos de forma automática, etc. Como desventaja, es importante tener en cuenta los **posibles problemas de concurrencia** que pueden derivarse de esto.

- ✓ **Destrucción:** si durante un tiempo (configurable) el servlet no recibe peticiones, es descargado de memoria.



## 2.3 Estructura básica de un servlet

### 2.3.1 Ejemplo 3

```
import jakarta.servlet.ServletConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

public class Ejemplo3 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo3() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see Servlet#init(ServletConfig)
     */
    public void init(ServletConfig config) throws ServletException {
        // TODO Auto-generated method stub
    }

    /**
     * @see Servlet#destroy()
     */
    public void destroy() {
        // TODO Auto-generated method stub
    }

    /**
     * @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)
     */
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }
}
```



Para describir la estructura básica de un servlet vamos a analizar uno de los ejemplos descritos anteriormente:

```
import jakarta.servlet.ServletConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
```

Vemos que utiliza las clases correspondientes a servlets (`jakarta.servlet.*`), las clases correspondientes a servlets que utilizan el protocolo HTTP (`jakarta.servlet.http.*`). En Tomcat podemos encontrar estas clases en la biblioteca de clases **Servlet-api.jar** dentro de la carpeta lib. También utiliza clases de entrada/salida (`java.io.*` - io viene input/output) para poder escribir en pantalla.

Lo segundo que vemos es el nombre de la clase (`class Ejemplo3`) que es pública (`public`), en el sentido de que cualquiera puede usarla sin ningún tipo de restricción y que hereda (`extends`) de la clase `HttpServlet`.

```
public class Ejemplo3 extends HttpServlet {
```

Algo que debemos saber es que toda clase, para que se considere un servlet, debe implementar el interfaz **`jakarta.servlet.Servlet`**. Para conseguirlo lo más sencillo es hacer que nuestra clase herede o bien de la clase **`jakarta.servlet.GenericServlet`** o **`jakarta.servlet.http.HttpServlet`**. Con la primera obtendremos un servlet independiente del protocolo, mientras que con la segunda tendremos un servlet HTTP. Sólo vamos a ver servlets que funcionen con el protocolo HTTP así que, por tanto, siempre heredarán de **`HttpServlet`**. Resumiendo, sólo cambiará el nombre de la clase para cada servlet que hagamos.

El siguiente trozo de código que aparece es la redefinición del método **`init`**. El servidor invoca a este método cuando **se crea el servlet** y en este método podemos hacer todas las **operaciones de inicialización** que queramos. Como en este servlet no nos hace falta inicialización ninguna, lo único que hacemos es llamar al método `init` por defecto (al de la superclase).

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);
}
```

Podemos observar que el método `init` es público, no devuelve ningún tipo (`void`), que puede lanzar una excepción (`ServletException`) y que tiene un parámetro (`ServletConfig config`). De estos dos últimos aspectos (excepción y parámetro) no nos tenemos que preocupar pues es el servidor quien ejecuta el

método `init`. En el peor de los casos, tendríamos que lanzar la excepción (si sabemos hacerlo), si por algún motivo el método `init` que nosotros implementemos falle (por ejemplo, que no se pueda conectar a la base de datos y evitamos mostrar un mensaje de error).

Lo siguiente que hacemos redefinir el método **`service`**, cuando el servidor web recibe una petición para un servlet llama al método **`public void service(HttpServletRequest req, HttpServletResponse res)`** con dos parámetros: el primero, de la clase **`HttpServletRequest`**, representa la petición del cliente y el segundo, de la clase **`HttpServletResponse`**, representa la respuesta del servidor (del servlet, más concretamente). **Este método será el que se ejecute cada vez que se realice una petición al servlet.** Es importante recordar al implementar este método que pueden ejecutarse de forma concurrente varias instancias del mismo servlet.

```
protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
    // TODO Auto-generated method stub  
    response.setContentType("text/html");  
    response.getWriter().append("<HTML>");  
    response.getWriter().append("<BODY>");  
    response.getWriter().append("<h1>EJEMPLO2</h1>");  
    response.getWriter().append("<br>HOLA MUNDO");  
    response.getWriter().append("</BODY>");  
    response.getWriter().append("</HTML>");  
    response.getWriter().close();  
}
```

Como en este primer ejemplo no necesitamos ninguna información del cliente, no usaremos para nada el parámetro **`HttpServletRequest`**, más adelante veremos cómo hacerlo. De la clase **`HttpServletResponse`** usamos dos métodos:

- ✓ **`setContentType(String str)`** para establecer el tipo de respuesta que vamos a dar. Para indicar que se trata de una página web, como haremos siempre, usamos el tipo `"text/html"`.

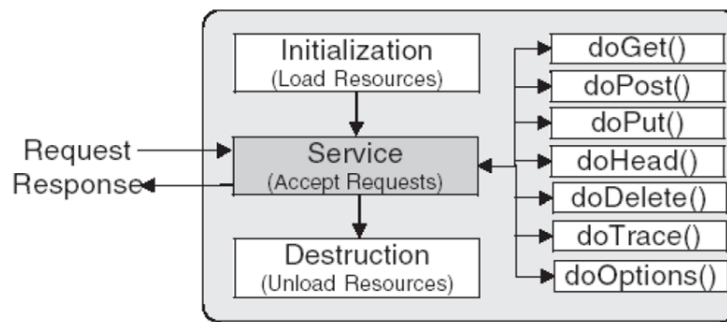
**`res.setContentType("text/html");`**

- ✓ **`PrintWriter getWriter(void)`** con el que obtendremos una clase `PrintWriter` en donde iremos escribiendo los datos que queremos que el cliente reciba.

**`PrintWriter out = res.getWriter();`**

Una vez que hemos establecido el tipo de respuesta (`text/html`) y tenemos el flujo de salida (variable `out`) sólo nos queda utilizar el método `println` de la clase `PrintWriter` para ir escribiendo en dicho flujo de salida la página HTML que queremos que visualice el cliente.

Por último se puede sobrescribir el método **`destroy()`** para realizar las acciones que se deseen realizar sólo una vez al destruir el servlet.



## 2.4 El HttpServlet

HTTP define las sentencias y protocolos por los que deben comunicarse los clientes, servidores, proxies, etc., de Internet.

Una orden o **transacción HTTP** está formada por un **encabezado** seguido de algunos datos. La cabecera especificará datos como el tipo de transacción, código de estado, etc. Algunas de las transacciones son:

- ✓ **GET:** petición de la representación de un recurso especificado. **Los parámetros necesarios se transmiten con la URL** del recurso al que quiere accederse.
- ✓ **POST:** igual que el anterior pero **los parámetros se incluyen en el cuerpo de la petición** y no en los argumentos. Es más seguro que el anterior, sobre todo si se cifra la comunicación.
- ✓ **PUT:** **sube un recurso al servidor.**
- ✓ **DELETE:** **borra el recurso especificado.**
- ✓ **OPTIONS:** **indica las operaciones que soporta un recurso situado en el servidor.**

El servidor responde con un código numérico y una breve descripción del mismo. Algunos de los códigos de respuesta del servidor son:

- ✓ **200:** petición realizada
- ✓ **400:** petición incorrecta
- ✓ **404:** recurso no encontrado
- ✓ **500:** error interno

El paquete `jakarta.servlet.http` ofrece la clase `HttpServlet` que implementa las características básicas de los servlets que trabajan con el protocolo HTTP. Algunos de los métodos más importantes son:

- ✓ **doGet():** **permite tratar una petición GET del protocolo HTTP.** Normalmente, para servlets HTTP, se sobrescribe este método o el método `doPostO` en lugar del método `service`).
- ✓ **doPost():** **permite tratar una petición POST del protocolo HTTP.** Normalmente, para servlets HTTP, se sobrescribe este método o el método `doGetO` en lugar del método `service`).

- ✓ **doPut():** permite tratar una petición PUT del protocolo HTTP
- ✓ **doDelete():** permite tratar una petición DELETE del protocolo HTTP
- ✓ **doOptions():** permite tratar una petición OPTIONS del protocolo HTTP

El HttpServlet extiende la clase GenericServlet, de la cual hereda algunos métodos como:

- ✓ **init():** tanto en su versión sin argumentos como en la versión que recibe argumentos de inicialización. Método que se ejecuta al inicio del ciclo de vida de un servlet.
- ✓ **destroy():** método que se ejecuta al final del ciclo de vida del servlet.
- ✓ **log():** métodos para la escritura en los ficheros de log de los servlets.
- ✓ **getServletContext():** devuelve el contexto en el que se ejecuta el servlet.
- ✓ **getServletInfo():** devuelve la información básica del servlet.
- ✓ **getInitParameter():** devuelve el valor de un parámetro de inicialización concreto.

## 2.5 Obtención de argumentos

Tanto si las peticiones son por GET (debemos implementar el método doGet()) como si son por POST (debemos implementar el método doPost()) podemos acceder a los argumentos mediante el objeto **HttpServletRequest** que se encuentra como parámetro en los dos métodos. Para el acceso a los argumentos podemos utilizar el método **getParameter()** pasando como argumento el nombre del parámetro.

### 2.5.1 Ejemplo4

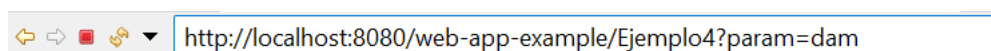
Por ejemplo, un servlet que reciba como argumento un nombre mediante GET:

```
public class Ejemplo4 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo4() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // TODO Auto-generated method stub
        response.setContentType("text/plain");
        response.getWriter().append("Recogida del parámetro:" + request.getParameter("param"));
        response.getWriter().close();
    }
}
```

La URL para llamar al servlet sería la siguiente:



Recogida del parámetro:dam

Como podemos ver en la URL, los argumentos van incluidos en la URL. En el caso de utilizar POST los datos van incluidos en la petición. Para realizar la petición debemos crear una página web con un formulario indicando que la acción a realizar será llamar al servlet y que el método utilizado será POST.

### 2.5.2 Ejemplo5

El código de un servlet que reciba los parámetros del formulario puede ser este:

```
public class Ejemplo5 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo5() {
        super();
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        response.getWriter().append("Nombre:" + request.getParameter("nombre") + "<br>");
        response.getWriter().append("Apellidos:" + request.getParameter("apellidos"));
        response.getWriter().close();
    }
}
```

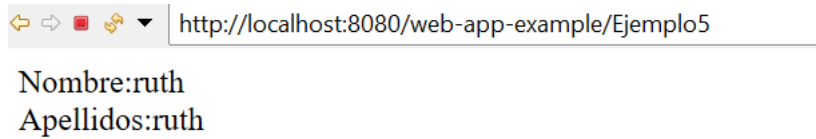
Por ejemplo, la página html “ejemplo5.html” podría ser esta:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>EJEMPLO5</title>
</head>
<body>
    <form action="Ejemplo5" method="post">
        <fieldset>
            <legend>Datos personales</legend>
            <label for="nombre">Nombre</label>
            <input type="text" name="nombre" size="25" maxlength="25"><br />
            <label for="apellidos">Apellidos</label>
            <input type="text" name="apellidos" size="25" maxlength="25"><br />
        </fieldset>
        <input type="submit" value="Enviar" />
        <input type="reset" value="Borrar" />
    </form>
</body>
</html>
```

Este código genera algo como esto:

Datos personales	
Nombre	<input type="text" value="ruth"/>
Apellidos	<input type="text" value="ruth"/>
<input type="button" value="Enviar"/>	<input type="button" value="Borrar"/>

Si pulsamos el botón enviar del formulario con los cuadros de texto rellenos, obtendremos algo como esto:



A screenshot of a web browser window. The address bar shows 'http://localhost:8080/web-app-example/Ejemplo5'. Below the address bar, the text 'Nombre:ruth' and 'Apellidos:ruth' is displayed, indicating the result of a form submission.

Los parámetros que llegan con GET y POST los tiene el objeto request y se pueden acceder con varios métodos:

- ✓ **public Enumeration getParameterNames():** Devuelve un Enumeration con los nombres de los parámetros, si no los hubiera, el Enumeration estará vacío. Un objeto Enumeration se usa como un Iterator:

```
Enumeration e = request.getParameterNames();
while (e.hasMoreElements()) {
    out.println(e.nextElement());
}
```

- ✓ **public String getParameter(String name) :** Valor del parámetro como un String, null si no existiera ese parámetro
- ✓ **public String[] getParameterValues(name):** Si un parámetro puede tener varios valores utilizar este método.

Lo habitual cuando hay errores en los formularios es volver a mostrarlos de nuevo:

- ✓ Mostrando los valores correctos en sus campos (el usuario no tiene que volver a introducirlos)
- ✓ Marcar los campos que no se han rellenado
- ✓ Si se han rellenado mal, añadir un comentario para indicar al usuario qué tiene que hacer
- ✓ Con JSP por ejemplo, es más fácil la gestión de formularios
- ✓ También es conveniente usar JavaScript para hacer una primera comprobación de los campos del formulario

Los parámetros de la cabecera de un mensaje HTTP se pueden obtener con los siguientes métodos:

- ✓ **String getHeader(String nombre):** Devuelve el valor de la cabecera indicada, null si no hay una cabecera con ese nombre
- ✓ **Enumeration<String> getHeaders(String nombre):** Algunas cabeceras, como Accept-Language, pueden tener una lista de valores

- ✓ **Enumeration<String> getHeaderNames():** Los nombres de todas las cabeceras presentes
- ✓ **String getMethod():** Indica el método HTTP: GET, POST, o PUT
- ✓ **String getQueryString():** La query string que hay en el URL, o null si no hay

## 2.5.2 Ejemplo6

```
public class Ejemplo6 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo6() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "MOSTRAR CABECERAS REQUEST";
        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"
            + "<B>Request Method: </B>" + request.getMethod() + "<BR>\n" + "<B>Request URI: </B>"
            + request.getRequestURI() + "<BR>\n" + "<B>Request Protocol: </B>" + request.getProtocol()
            + "<BR><BR>\n" + "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" + "<TR>\n"
            + "<TH>Header Name<TH>Header Value");
        Enumeration<String> headerNames = request.getHeaderNames();
        while (headerNames.hasMoreElements()) {
            String headerName = headerNames.nextElement();
            out.println("<TR><TD>" + headerName);
            out.println("<TD>" + request.getHeader(headerName));
        }
    }
}
```

## MOSTRAR CABECERAS REQUEST

Request Method: GET  
 Request URI: /web-app-example/Ejemplo6  
 Request Protocol: HTTP/1.1

Header Name	Header Value
accept	image/gif, image/jpeg, image/pjpeg, application/x-ms-application, application/xhtml+xml, application/x-ms-xbap, */*
accept-language	es-ES,es;q=0.5
cache-control	no-cache
ua-cpu	AMD64
accept-encoding	gzip, deflate
user-agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko
host	localhost:8080
connection	Keep-Alive