-Tema 5
Introducción a la plataforma JavaEE. Servlets

¿Qué es un servidor de aplicaciones?

Un **servidor web** (Apache) almacena y entrega contenido o servicios a usuarios en respuesta a peticiones, normalmente http.

Un **servidor de aplicaciones** es un software que, además de funcionar como servidor web, proporciona una serie de servicios a aplicaciones que se ejecutan en su interior. Por ejemplo: pools de conexiones con BBDDs, gestión de transacciones, clustering, reparto de la carga de la aplicación web entre varios servidores, etc.
El término "servidor de aplicaciones" usualmente hace referencia a un servidor de aplicaciones Java EE.
Algunos servidores de aplicación Java EE son: WebLogic, Jboss, GlassFish
Mucha gente confunde Tomcat como un servidor de aplicaciones; sin embargo, es solamente un contenedor de servlets.

Para una app web Java, el servidor de aplicaciones será quien:

- Recibe la petición http.
- Analiza dicha petición y almacena su información en forma de objetos Java.
- Busca si existe algún programa registrado para gestionar la petición (un servlet) e invoca dicho programa, pasándole como parámetros objetos Java con toda la información de la petición.
- Empleando el API (clases de Java) adecuado, nuestro programa podrá examinar la petición, procesarla, y generar la respuesta

¿Qué son los Servlets?

Los Servlets son objetos Java (de clases que implementan la interface Servlet) que se ejecutan en un servidor de aplicaciones Java EE y permanecen a la espera de peticiones web de clientes, normalmente peticiones http, y les dan una respuesta.
Actúan como una capa intermediaría entre la petición web y las aplicaciones, BBDDS o recursos del servidor Web.
Los Servlets no se encuentran limitados a un protocolo de comunicaciones específico, pero en la práctica se utilizan únicamente con el protocolo HTTP.

Lugar de los Servlets en la jerarquía de clases Java

Un servlet, al ser una clase de Java, incorpora todas las ventajas del lenguaje Java en cuanto a portabilidad y seguridad. Los 2 paquetes principales de la API Java con clases para programación de servlets son:

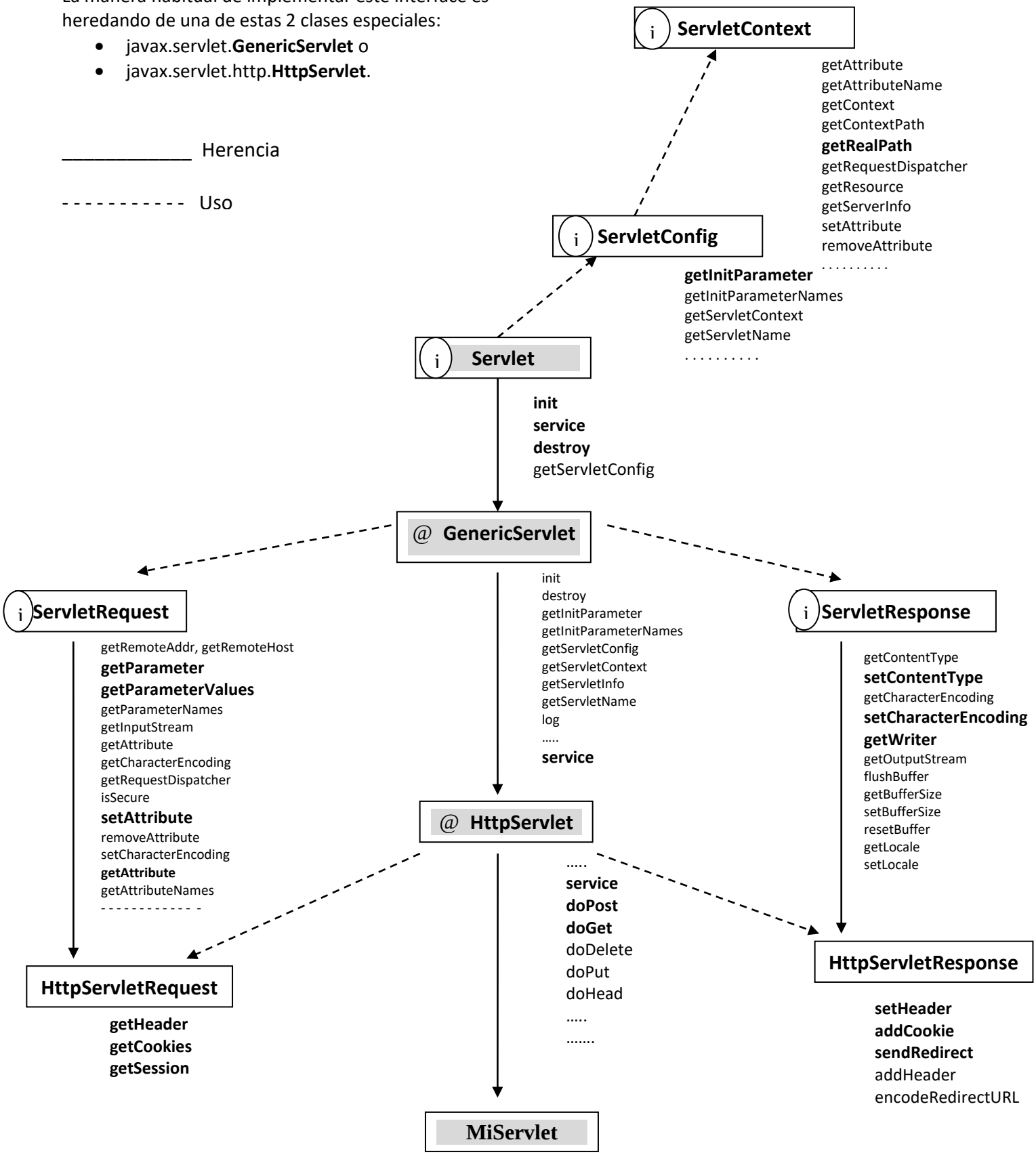
- **javax.servlet** → Contiene clases e interfaces para los servlets genéricos, independientes del protocolo
- **javax.servlet.http** →Clases que extienden las del paquete anterior, añadiendo funcionalidad específica para el protocolo http

Los servlets son clases que deben implementar la interface javax.servlet.**Servlet**
La manera habitual de implementar este interface es heredando de una de estas 2 clases especiales:

- javax.servlet.**GenericServlet** o
- javax.servlet.http.**HttpServlet**.

_____ Herencia

- - - - - - - - - - - Uso



Características de los servlets

- Al estar escritos en Java, son portables: el servidor en el que corren puede estar escrito en cualquier lenguaje de programación, y tener cualquier arquitectura

- Los servlets pueden llamar a otros servlets, e incluso a métodos concretos de otros servlets. Así, se puede distribuir del trabajo a realizar. P.e, se podría tener un servlet encargado de la interacción con los clientes y que llamara a otro servlet que se encargara de la comunicación con una base de datos.

- Los servlets pueden obtener fácilmente información acerca del cliente (la enviada por el protocolo), tal como su dirección, puerto que se utiliza en la llamada, método (GET, POST, etc)

- Permiten la utilización de cookies y sesiones, para guardar información específica acerca de un cliente

- Pueden actuar como enlace entre el cliente y una o más bases de datos

- Permiten la generación dinámica de código html

Tareas de un servlet HTTP

Este es la secuencia de pasos más habitual que realiza un servlet http:

- Leer los datos enviados por el usuario: estos datos se mandan frecuentemente desde formularios HTML. Aunque esta información también puede venir de otras herramientas HTTP.

- Buscar otra información asociada a la petición: características del navegador, cookies, nombre de la máquina del cliente, id de la sesión si existe, etc.

- Procesar la información para obtener los resultados: este proceso puede requerir acceder a una base de datos utilizando JDBC o generar la respuesta de manera directa.

- Formatear los resultados en un flujo de salida, normalmente una página HTML.

- Asignar los parámetros apropiados de la respuesta HTTP: indicar al navegador el tipo de documento que se le envía (por ejemplo HTML), asignar valores a las cookies, etc.

- Enviar el documento al cliente: en formato de texto (HTML), pero también en formato binario (imágenes), en formato comprimido (fichero ZIP), etc

Principales clases/métodos dentro de la API de Servlets:

Interface **Servlet**

Es la interface principal. Todos los servlets que creemos deben implementarla, directamente o más comúnmente, extendiendo de una clase que lo implemente como HttpServlet. Métodos:

- init (ServletConfig config): Método invocado por el servidor cuando se instancia del servlet. Puede ser sobreescrito para realizar tareas que deban realiza 1 vez al comienzo de la vida del servlet, por ejemplo, poner un contador de visitas a 0

- getServletConfig (): Retorna la configuración dada para la inicialización del servlet

- service (ServletRequest request, ServletResponse response): Invocado por el servidor cada vez que se recibe una petición de un cliente. En su implementación para HTTP verifica el tipo de solicitud GET, POST, etc. y la redirige a los métodos doPost, doGet. Es el punto de entrada de las peticiones realizadas a un Servlet. No es necesario ni recomendable sobrescribir este método.

- destroy(): Este método es llamado por el servidor antes de que el servlet sea destruido.

Clase abstracta **GenericServlet**: **public abstract class GenericServlet** implements **Servlet, ServletConfig**

Usada para programar servlets genéricos, independientes del protocolo
Implementa el método service para establecer el comportamiento mínimo de todos los servlets.

Clase abstracta **HttpServlet**: **public abstract class HttpServlet extends GenericServlet**:

Utilizada para programar servlets que respondan a peticiones http. Todo Servlet http debe heredar de ella y sobrescribir alguno de sus métodos. Sus métodos más comunes son:

- doGet (HttpServletRequest request, HttpServletResponse response)

- doPost (HttpServletRequest request, HttpServletResponse response):

  Invocados cuando se solicita el servlet por GET o por POST, respectivamente
  doGet y doPost reciben 2 parámetros, de tipo HttpServletRequest y HttpServletResponse.
  Veamos de dónde proceden esas clases, así como su propósito.

Interface **ServletRequest:** Se utiliza para representar la petición del cliente, independiente del protocolo, p.e:

- getInputStream(): Recupera el cuerpo de la petición en binario (para peticiones POST)

Clase **HttpServletRequest:** Implementa la interface ServletRequest, añadiendo métodos específicos para acceder al contenido de una petición HTTP

Al recibir una petición, el contenedor de Servlets creará un objeto HttpServletRequest que la represente y se lo pasará como parámetro a los distintos métodos de servicio de un servlet: doGet(), doPost() , etc

Un objeto de tipo HttpServletRequest representa la petición del servlet que hace el cliente (navegador). Tiene métodos para obtener la información que envía el usuario en la petición: encabezados de petición del protocolo HTTP, cookies enviadas, datos de un formulario, datos de sesión, atributos…

Algunos métodos:

- String getHeader(String name)**:** Devuelve el contenido de la cabecera http cuyo nombre se le pasa
- Cookie[] getCookies(): Devuelve 1 array con las cookies que el cliente envía al servlet
- HttpSession getSession(): Devuelve la sesión en la que se encuentra el cliente. Si la petición no tiene una sesión asociada, crea una.
- HttpSession getSession(boolean create):Similar al anterior, pero sólo crea la sesión si se le pasa true
- String getParameter(String name): Devuelve el valor del parámetro cuyo nombre se le pasa como argumento, o null
- String[] getParameterValues(String name): Devuelve un array de Strings con todos los valores asociados con un determinado parámetro, o null (Para recibir campos "multiselect")
- Object getAttribute(String name): devuelve el valor del atributo cuyo nombre se le pasa, o null.
- void setAttribute(String name, Object object): asigna a la petición un atributo con el nombre indicado
- String getContextPath(): Devuelve el fragmento de la URL que indica el contexto de la petición. En un mismo servidor de aplicaciones puede haber desplegadas varias aplicaciones Java EE. El servidor se encarga de aislar las unas de las otras, de modo que un error en una de ellas no afecte a las demás. Cada una de estas aplicaciones tiene su propio contexto, y ese contexto es el primer fragmento de la URL que va justo después del nombre del servidor.
- String getMethod(): Devuelve el método http empleado en la petición; por ejemplo GET, POST, o PUT.

Interface **ServletResponse**: Representa la respuesta que el servlet dará al cliente, independiente del protocolo

- setContentType(String type): Permite definir el tipo de respuesta que se enviará al cliente: **text/html**, si va a ser una página web, **application/*pdf, image/jpeg***, etc
- getWriter(): Devuelve un objeto PrintWriter asociado con la respuesta que será enviada al cliente. Crearemos la página web que se va a enviar al usuario escribiendo HTML en este flujo
- getOutputStream(): Devuelve un objeto ServletOutputStream. Se usa para construir Servlets que devuelvan respuestas binarias (por ejemplo, una imagen) y no sólo texto.

Clase **HttpServletResponse:** Implementa la interface ServletResponse, añadiendo métodos para acceder y configurar una respuesta HTTP

- void addCookie(Cookie cookie): Añade una cookie a la respuesta
- void addHeader(String name, String value): Añade a la respuesta una nueva cabecera
- void sendRedirect(String location): Envía un mensaje al cliente para que éste redireccione la respuesta a la dirección señalada. Es importante notar que genera una nueva petición desde el cliente
- String encodeRedirectURL(String url): Prepara una URL para ser usada por el método sendRedirect

Interface **ServletConfig**: Para que el contenedor del servlet, pase al servlet información de configuración: parámetros de inicialización, contexto del servlet, etc

- getInitParameter(String name): Devuelve el valor de 1 de parámetro inicialización del servlet
- getInitParameterNames(): Devuelve los nombres de todos los parámetros de inicialización del servlet
- getServletContext(): Devuelve 1 objeto ServletContext con la información referente al servidor

Interface **ServletContext**: Se refiere al contexto del servlet o dicho de otra manera, a la "Aplicación web". Se trata de aquella información que es común a todos los Servlets desplegados en la aplicación actual.

Existe un único contexto por cada aplicación web y por cada instancia de la máquina virtual Java. Por eso, el contexto es un mecanismo para comunicar información entre todos los Servlet que corran en una misma aplicación web, siempre y cuando esa aplicación web corra en un mismo servidor (si tenemos varios servidores, y por tanto varias máquinas virtuales, el contexto de la aplicación no es un sitio adecuado para guardar información a la que queramos acceder globalmente; en estos escenarios suele recurrirse a una base de datos).

- getAttribute(String) : Devuelve el valor de un atributo del contexto
- setAttribute(String, Object): guarda en el contexto el objeto que se le pasa como 2º parámetro, con el nombre pasado como 1º parámetro
- removeAttribute(String) :elimina el atributo de contexto indicado
- getContextPath(): devuelve la ruta del contexto de la aplicación web
- getRequestDispatcher(String): devuelve un objeto RequestDispatcher que suele emplearse para redirigir la petición a otro recurso. Ejemplo de redirección a "recurso":
  request. getRequestDispatcher("recurso").forward(request,response);
- getResource(String):devuelve una URL que se corresponde con la ruta del recurso que se le pasa como parámetro. La ruta del recurso debe comenzar como un "/" y se interpreta como una ruta relativa a la raíz del contexto de la aplicación, o relativa al directorio /META-INF/resources de un archivo jar que esté contenido en /WEB-INF/lib
- getResourceAsStream(String): Similar al anterior pero devuelve el recurso como un flujo de entrada
- getRealPath(String):Devuelve la URL absoluta para nuestra URL relativa

Funcionamiento de un servlet (NUESTRO PRIMER SERVLET)

Los Servlets son programas Java (clases Java compiladas -.class-) que se ejecutan dentro de un servidor de aplicaciones Java EE.
A la parte web de un servidor de aplicaciones Java EE se le suele llama **contenedor de Servlets**.
Los servlets deben estar **registrados** para poder comenzar a atender peticiones de clientes.
(Más adelante veremos cómo se registran los servlets → esto corresponde al despliegue de la aplicación web)

Cuando al contenedor le llega una petición http, comprueba si hay algún Servlet registrado para responder a dicha petición; en caso afirmativo, lo llamará

Para que una clase pueda ser trata como un Servlet, debe heredar de la clase **HttpServlet.** Se trata de una clase abstracta que representa a los Servlets HTTP y de la que deben heredar los servlets que programemos.

Esta clase ofrece, entre otros, los métodos **doGet()** y **doPost()** que podrá sobrescribir nuestro servlet.
Si la petición que ha llegado al Servlet era de tipo GET se invocará doGet() y si era tipo POST invocará doPost()

→ *Debes distinguir cuándo se genera una petición GET o POST*

Tanto **doPost()** como **doGet()** reciben 2 parámetros: 1 objeto **HttpServletRequest** y 1 objeto **HttpServletResponse.**
HttpServletRequest y **HttpServletResponse** representan respectivamente una petición y una respuesta del protocolo HTTP.

El objeto **HttpServletRequest** ofrece métodos para obtener información relativa a la petición HTTP realizada, como puede ser información incluida en formularios HTML, cabeceras de petición HTTP, etc

El objeto **HttpServletResponse** nos permite elaborar la respuesta HTTP a enviar al cliente. Esta información incluye cabeceras de respuesta del protocolo HTTP, códigos de estado, y lo más utilizado: nos permite obtener un objeto PrintWriter, donde escribir texto HTML.

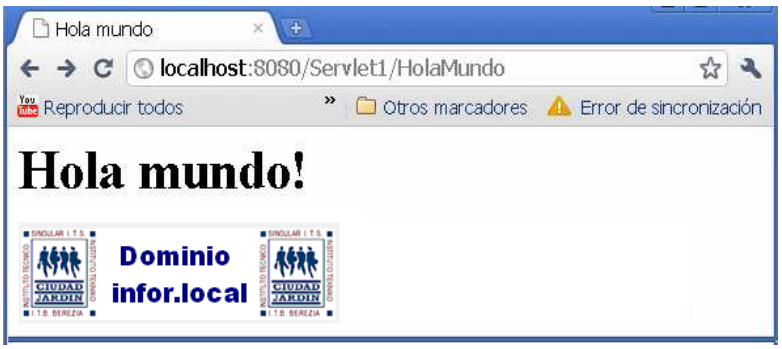
Al reescribir **doGet** y **doPost**, hay que tener en cuenta que lanzan las excepciones ServletException e IOException

Ejemplo 1: Servlet HolaMundoServlet

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HolaMundoServlet extends HttpServlet {

        protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                throws ServletException, IOException
        {
                response.setContentType("text/html;charset=UTF-8");
                PrintWriter out;
                out = response.getWriter();
                try {
                        out.println("<html>");
                        out.println("<head>");
                        out.println("<title>Hola mundo</title>");
                        out.println("</head>");
                        out.println("<body>");
                        out.println("<h1>Hola mundo!</h1>");
                        out.print("<img src='falta.png'/>");
                        out.println("</body>");
                        out.println("</html>");
                }
                finally {
                        out.close();
                }
        }
}
```

Servlet básico que genera una salida html estática.
En este momento, simplemente debes tener una idea intuitiva sobre la estructura de la clase, el método doGet, el significado de los parámetros recibidos, etc

Este ejemplo genera una página web estática que incluye un saludo y una imagen. La imagen debe de estar situada en la raíz de la carpeta web para poder acceder a ella empleando la ruta indicada en el código.
Si desplegamos este Servlet en Tomcat empleando Eclipse, Deberiamos de poder acceder mediante nuestro navegador y nos mostrará esta página:



Observa la URL que se muestra en el navegador.

- La máquina es localhost.
- El protocolo (aunque Chrome lo omite) http.
- El puerto en el cual está el servidor es el 8080
- La aplicación está corriendo dentro del contexto "Servlet1" del servidor de aplicaciones.
- Y el recurso de la aplicación al cual estamos accediendo es HolaMundo.

Registrar Servlets

Debemos identificar nuestros servlets (nombrarlos) e indicar qué peticiones pueden atender. Existen 2 maneras:

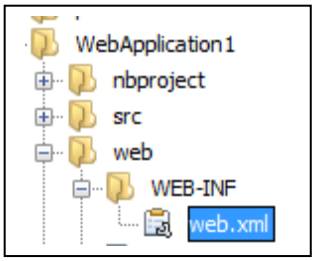
Mediante **anotaciones**: Líneas encabezadas por @ dentro de la propia clase-servlet

En la anotación indicamos un nombre interno para el servlet y los "patrones URL" a los que deseamos que el servlet responda (los patrones pueden usar el carácter comodín *)

Ejemplo:

@WebServlet(name="HolaMundoServlet", urlPatterns={"/HolaMundo/*","*.saludo"})

- Da al servlet el nombre interno HolaMundoServlet (coincide con el nombre de la clase, aunque no tendría por qué)
- El Servlet se registra para responder a URL's como las siguientes:
  http:// máquina:puerto/contexto/HolaMundo/synopsis
  http:// máquina:puerto/contexto/HolaMundo/complete?date=today
  http:// máquina:puerto/contexto/HolaMundo
  http:// máquina:puerto/contexto/Hola.saludo
  http://máquina:puerto/contexto/directorio/directorio2/compl?date=today.saludo
  http:// máquina:puerto/contexto/directorio/fichero.saludo



Configurando el descriptor de despliegue: archivo **web.xml**

El descriptor de despliegue es un fichero de nombre **web.xml** que debe de situarse en la raíz del directorio WEB-INF. Algunos de sus usos son:

- registrar servlets e informar de correspondencias de URLs con servlets
- definir parámetros de inicialización de servlets y aplicaciones web
- configurar la sesión
- configurar la seguridad
- …….

Contenido de web.xml equiparable a la siguiente anotación en el fichero ***HolaMundoServlet.java*** :
@WebServlet(name="HolaMundoServlet", urlPatterns={"/HolaMundo"})

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
        <servlet>
                <servlet-name>HolaMundoServlet</servlet-name>
                <servlet-class>paquete.HolaMundoServlet</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>HolaMundoServlet</servlet-name>
                <url-pattern>/HolaMundo</url-pattern>
        </servlet-mapping>
</web-app>
```

La etiqueta <servlet> registra un Servlet en el contenedor. Incluye las subetiquetas: <servlet-name>, con el nombre interno del Servlet, y <servlet-class>, con la clase a la que pertenece (nombre completo con el paquete).

La etiqueta <servlet-mapping> permite especificar a qué URLs va a responder cada Servlet. Dentro de ella debemos indicar el nombre interno del Servlet y el patrón/patrones de URL al cual queremos que éste responda. Se trata de directorios VIRTUALES sin correspondencia con ninguna ubicación física

El descriptor de despliegue tiene prioridad sobre las anotaciones.

Métodos **doGet** y **doPost**

La solicitud de un recurso web por parte de un cliente al servidor, se materializa en una petición http que puede ser de tipo GET o POST

Éste es el aspecto que tienen este tipo de peticiones a su llegada al servidor:

Petición GET	Petición POST
GET /servlet/MyServlet?nombre=Juan&pais=es HTTP/1.1 Connection: Keep-Alive User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows NT) Host: www.datsi.fi.upm.es Accept: image/gif, image/x-xbitmap, image/jpeg,	POST /servlet/MyServlet HTTP/1.1 User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows NT) Host: www.datsi.fi.upm.es Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */ Content-type: application/x-www-form-urlencoded Content-length: 30 nombre=Juan&pais=es

Una petición GET de un servlet (que es el tipo de petición URL por defecto) se traducirá en una llamada al método **doGet** que habremos codificado en dicho servlet. Asimismo, una petición HTTP POST se traducirá a una llamada al método **doPost** del servlet.

Volviendo al ejemplo de **HolaMundoServlet**, el método doGet tiene 2 parámetros:

doGet (HttpServletRequest request, HttpServletResponse response)

- El primero (request) es un objeto Java que envuelve la petición que ha llegado al servidor. Empleando métodos de este objeto, podremos acceder a información como a las cabeceras de la petición, parámetros de un formulario si es que se ha enviado, etcétera. En nuestro ejemplo con ***HolaMundoServlet***, este objeto no se usa para nada.

- El segundo objeto (response) se utiliza para generar la respuesta.
  Habitualmente, se obtiene de él un flujo de tipo PrintWriter (salida de texto), donde se escribe html.

El funcionamiento sería idéntico si el método fuera doPost

Es habitual implementar sólo un método (o bien doGet o bien doPost) y hacer que el otro lo referencie.

Ejemplo 2

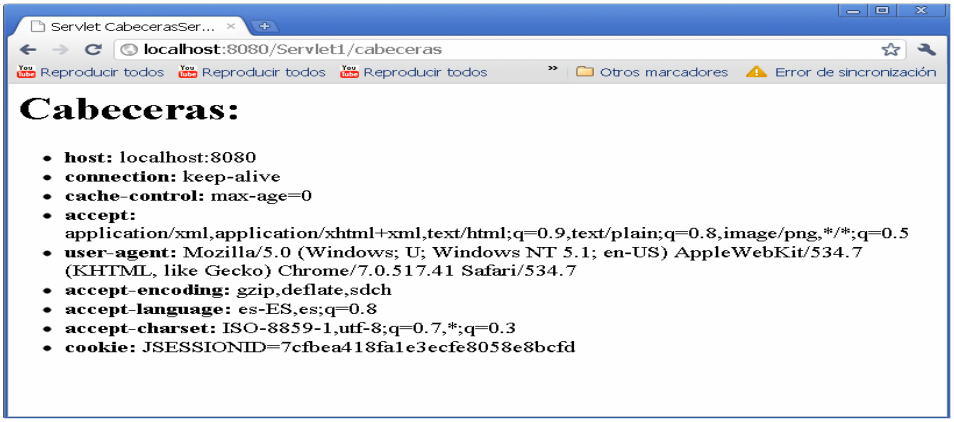
Servlet que genera una respuesta al usuario donde se muestran todas las cabeceras de la petición http recibida y el primero de sus valores (ya que es posible que 1 cabecera tenga múltiples valores).

El patrón de URL asociado con el ejemplo es /cabeceras.
Este Servlet codifica los métodos doGet y doPost de tal modo que ambos hacen lo mismo: invocan al método processRequest pasándole los parámetros con los que han sido invocados.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException
{
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Servlet CabecerasServlet</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("<h1>Cabeceras: </h1>");
                out.println("<ul>");
                Enumeration<String> nombresDeCabeceras =request.getHeaderNames();
                while (nombresDeCabeceras.hasMoreElements()) {
                        String cabecera = nombresDeCabeceras.nextElement();
                        out.println("<li><b>" + cabecera + ": </b>"+ request.getHeader(cabecera) + "</li>");
                }
                out.println("</ul>");
                out.println("</body>");
                out.println("</html>");
        } finally {
                out.close();
        }
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
                processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
                processRequest(request, response);
}

}
```

Salida generada para Google Chrome

En este ejemplo, la salida que genera es dinámica: texto estático combinado con variables Java

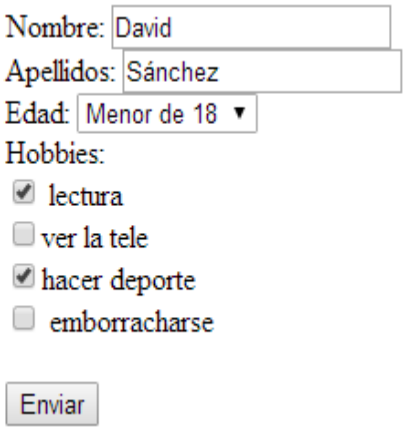


Ejemplo 3

Servlet para procesar formularios

Es habitual usar Servlets para procesar información que el cliente nos manda, normalmente desde formularios.

Supongamos que tenemos el siguiente formulario:




```
<form method="get" action="ServletLeeDatos" >
        Nombre: <input name="nombre"><br>
        Apellidos: <input name="apellidos"><br>
        Edad:
        <select name="edad">
                <option value=”Menor”>Menor de 18</option>
                <option value=”Joven”>De 18 a 30</option>
                <option value=”Adulto”>De 30 a 55</option>
                <option value=”Mayor”>Mayor de 55</option>
        </select>
        <br>
        Hobbies:<br>
        <input name="hobbies" value="lectura" type="checkbox"> lectura</br>
        <input name="hobbies" value="tele" type="checkbox">ver la tele</br>
        <input name="hobbies" value="deporte" type="checkbox">
                                                Hacer deporte</br>
        <input name="hobbies" value="emborracharse" type="checkbox">
                                                emborracharse</br></br>
        <input type=”submit” value=”Enviar”/>
</form>
```

** Atención, un conjunto de checkboxes relacionados, si va a ser procesado por un servlet Java, no necesita que le asignemos un nombre de array (a diferencia de en php). Basta con que el **name** sea el mismo para todos ellos*
(Lo mismo para cualquier entrada multiselect de formulario)

Observa que el formulario va a enviar sus datos a la URL formada al retirar el nombre del formulario de la URL de la página web donde se aloja este, y añadir "/ServletLeeDatos".
Por ejemplo, si el formulario estaba en ***http://localhost:8080/contexto/formulariohobbies.html***, la URL a la cual se enviará la petición es: ***http://localhost:8080/contexto/ServletLeeDatos*** empleando el método GET.
En esa URL debe haber un Servlet preparado para procesar la información. Este formulario envía sus parámetros como parte de la URL, ya que emplea el método GET; obtendríamos la URL:

http://localhost:8080/contexto/ServletLeeDatos?nombre=Abraham&apellidos=Otero+Quintana&edad=De+30+a+55&hobbies=lectura&hobbies=tele&hobbies=emborracharse

Si hubiésemos indicado que el formulario se enviase empleando el método POST sólo veríamos la URL ***http://localhost:8080/contexto/ServletLeeDatos***, y el resto de la información se enviaría en el cuerpo del mensaje.

Cuando la petición llega al servidor, tanto si se envía por GET como por POST, el contenedor de Servlets recoges los parámetros del formulario y los guarda en el objeto HttpServletRequest. Podemos acceder a ellos a través de los métodos **getParameter(String)** o **getParameterValues(String)**. El parámetro de estos métodos es el atributo "name" de cada input de formulario.

Servlet que procesa el formulario anterior:

```
public class ServletLeeDatos extends HttpServlet {

        protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                                throws ServletException, IOException {
                request.setCharacterEncoding("utf-8");
                String nombre = request.getParameter("nombre");
                String apellidos = request.getParameter("apellidos");
                String edad = request.getParameter("edad");
                String[] hobbies = request.getParameterValues("hobbies");

                response.setContentType("text/html;charset=UTF-8");
                PrintWriter out = response.getWriter();
                try {
                        out.println("<html>");
                        out.println("<head>("<title>Servlet que procesa un formulario basico</title></head>");
                        out.println("<body>");
                        out.println("<h1>" + "Hola " + nombre + " " + apellidos+"</h1>");
                        out.println("Eres " + edad + " );
                        if (hobbies!=null){
                                out.println ("Tus hobbies son:");
                                out.println("<ul>");
                                for (String hobby : hobbies) {
                                        out.println("<li>" + hobby + "</li>");
                                }
                                out.println("</ul>");
                        }
                        out.println("Este formulario ha sido invocado con Los siguientes parámetros:<br/>");
                        out.println(request.getQueryString());
                        out.println("</body>");
                        out.println("</html>");
                } finally {
                        out.close();
                }

}
```

Como hasta ahora:
- Los campos de texto se envían siempre
- Los radios, checkboxes, etc no se envían si no se selecciona ninguno. En ese caso, en Java, el parámetro contiene null y tienes que evitar posibles NullPointerException's

En el caso de los hobbies, hemos usado el método ***String [] getParameterValues (String)*** ya que el campo de formulario 'hobbies' es de selección múltiple y puede contener varios valores.
El método **getQueryString** se emplea para mostrar, con propósitos didácticos, los parámetros enviados al servidor. Ejemplo de la salida producida por el Servlet:

Hola David Sánchez

Eres "Menor" y tus hobbies son:

- lectura
- deporte

Este formulario ha sido invocado con Los siguientes parametros:
nombre=David&apellidos=S%C3%A1nchez&edad=%E2%80%9DMenor%E2%80%9D&hobbies=lectura&hobbies=deporte

Llamadas a servlets

Un servlet es un recurso web y puede ser accedido de diversos modos, (que irán apareciendo):

- Llamado desde un formulario
- Usando etiquetas HTML (<a href=referencia_servlet ….>, <img src=referencia_servlet….>, etc
- Redireccionando desde otro servlet
- Escribiendo su URL en un navegador
- …

Ciclo de vida de un servlet: init, service, destroy

Los Servlets son gestionados por el contenedor de servlets. El contenedor decide cuándo se debe crear un Servlet, y cuándo se debe destruir. El programador no instancia ni controla directamente la vida del Servlet.

Un mismo servlet http sirve para atender todas las peticiones web:
Los servlets, una vez iniciados, quedan activos hasta que su gestor los desactiva

Inicialización: método ***init*** *(ServletConfig config).*

Lo ejecuta el contenedor una sóla vez, cuando instancia el Servlet y antes de atender peticiones. Hasta no completar su init, un servlet no responderá a peticiones de clientes.
init permite al servlet realizar labores de inicialización costosas (con BD, conexiones a otros equipos, etc), que haya que ejecutar una sóla vez.
init puede recibir un argumento de tipo *ServletConfig*→ siguiente apartado
El método init ya está implementado en la clase GenericServlet, de modo que si queremos hacer una inicialización específica de nuestro servlet lo podemos redefinir, sin olvidar invocar en primer lugar el init heredado.

```
public class EjemploInit extends HttpServlet {

   private int contadorVisitas;
   private MiObjeto ob;

   public void init(ServletConfig config) throws ServletException {

         //Inicializa el objeto ServletConfig y registra la inicialización
         super.init( config );

         //Inicializa las variables contados y ob
         contadorVisitas=0;
         ob=new MiObjeto();
   }
……
}
```

Si definimos atributos en 1 servlet, éstos tienen ámbito de "aplicación", es decir, se comparten por todas las peticiones/clientes

El contenedor de servlets crea una única instancia del Servlet por cada definición de dicho Servlet que encontremos en el descriptor de despliegue de la aplicación.
(A no ser que el descriptor de despliegue indique varios nombres (<servlet-name>) para un mismo servlet (<servlet-class>))
Esto no quiere decir que vaya a haber un único hilo que ejecute dichas peticiones. Múltiples hilos pueden estar ejecutando el cuerpo del Servlet para responder a peticiones simultáneas de múltiples usuarios.

Peticiones y respuestas: método **service**

Una vez el servlet está en marcha, cada petición por parte del cliente se traduce en una llamada al método ***service****(ServletRequest request, ServletResponse response)* del servlet. El cometido de este método es generar una respuesta por cada petición recibida de un cliente y enviársela.
Podría haber múltiples respuestas procesándose al mismo tiempo, pues los servlets son multithread.

No es aconsejable definir (codificar) el método ***service()*** (No queda más remedio si nuestro servlet hereda de GenericServlet). Lo habitual es heredarlo de HttpServlet.
El método service() de HttpServlet define service( ) llamando a otros métodos (doPost, doGet, ….) que son los que sí tiene que redefinir el programador

service(….) recibe 2 parámetros:

- La petición del cliente: Cabeceras, datos de formulario, datos de sesión, …
- La respuesta del servlet. Requiere llamar al método setContentType para definir el contenido MIME

Finalización ordenada: método **destroy**

Los servlets se ejecutan indefinidamente hasta que el servidor los destruye (por apagado del servidor, por petición del administrador….)
El contenedor de servlets, antes de destruir nuestro Servlet, invocará al método **destroy**(), dándonos la oportunidad de realizar en él limpieza de recursos adquiridos: cerrar ficheros abiertos, conexiones con bases de datos, etc. Nuestros servlets pueden redefinir ***destroy*** con tal propósito.

Configurando servlets. Clase ServletConfig

La clase ServletConfig es empleada por el contenedor, sobre todo, para pasarle parámetros al Servlet durante su inicialización. La cabecera del método init es:

*public void init (**ServletConfig** config) throws ServletException*

¿Cómo podemos especificar los parámetros de inicialización de un Servlet?.

Añadiéndolos a la anotación @WebServlet:

Cada parámetro se indica con una anotación @WebInitParam, que tiene 2 atributos: el nombre (name) del parámetro y su valor (value).

```
@WebServlet(name="ConfigurableServlet",
urlPatterns={"/ConfigurableServlet"}, initParams={@WebInitParam(name="parametro1", value="Valor1"),
                                                  @WebInitParam(name="parametro2", value="Valor2")})
```

Indicándolos en el descriptor de despliegue (web.xml):

```
<servlet>
    <servlet-name>ConfigurableServlet</servlet-name>
    <servlet-class>paquete.ConfigurableServlet</servlet-class>
    <init-param>
            <param-name>parametro1</param-name>
            <param-value>Valor1</param-value>
    </init-param>
    <init-param>
            <param-name>parametro2</param-name>
            <param-value>Valor2</param-value>
    </init-param>
</servlet>
```

Si definimos el mismo parámetro con valores distintos en una anotación y en el descriptor de despliegue, el descriptor de despliegue tiene prioridad

Estos son algunos métodos de ServletConfig:

String getInitParameter(String name)	Devuelve el parámetro de inicialización asociado con el nombre que se le pasa como argumento.
Enumeration<String> getInitParameterNames()	Devuelve los nombres de los parámetros de inicialización. Si no hay ningún parámetro, la enumeración estará vacía.
ServletContext getServletContext()	Devuelve una referencia al contexto del Servlet
String getServletName()	Devuelve el nombre de esta instancia de Servlet.

sendRedirect Vs Forward

Veremos la diferencia entre estas dos formas de redireccionar:

SendRedirect.

Es un método de la respuesta y pide al navegador del cliente que realice una nueva petición (mediante URL) de un recurso. En definitiva, es la máquina del cliente (el navegador) quien realiza una nueva petición del recurso indicado. Esto implica dos aspectos fundamentales:

1. Se pierden los objetos request y response de la petición precedente.
2. Mediante esta operación no podemos tener acceso a los ficheros contenidos en el WEB-CONTENT.

response.sendRedirect ("URL del recurso");

Forward

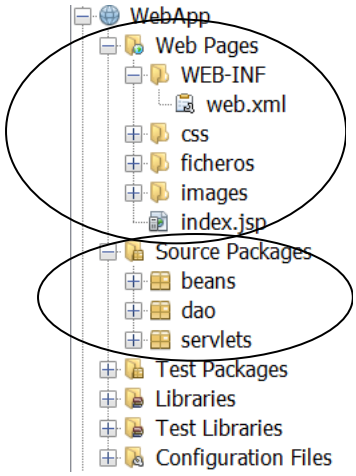
En este caso, es **el servlet quien pasa la peticion** contra un nuevo recurso requerido. Esto implica:

1. **Se pueden conservar los objetos request y response** de la petición (por ejemplo, los parámetros introducidos por el cliente en un formulario en la petición anterior)
2. Mediante esta operación **podemos acceder a los ficheros contenidos en el WEB-CONTENT**

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
                                                        throws ServletException, IOException
{
        request.getRequestDispatcher("OtroServlet").forward(request, response);
 }
```

Se resume en que: *sendRedirect* es un método del objeto response y *forward* es un método del objeto request.

Estructura de una aplicación Java Web



Zona WEB:
páginas hmtl, jsp, recursos estáticos: estilos, imágenes, etc

Clases Java:
servlets, beans, clases dao (de acceso a BBDD)

- En la **zona Web**, está, entre otros el directorio **WEB-INF**, que es privado y contiene todos los recursos que no deben servirse directamente al cliente.
  Aquí está el archivo web.xml donde se establece la configuración de la aplicación web y las librerías. (También almacena las clases y librerías de Java compiladas una vez se despliegue el proyecto)

- La zona "Source Packages", almacena clases Java.
  Señalar que estas clases no pueden acceder a recursos de la zona Web directamente por su ruta.

Para que un servlet acceda a un archivo de la zona web, podemos elaborar su "ruta" desde el contexto, utilizando:

….getServletContext().getRealPath(archivo);

Ejemplo: acceso desde 1 servlet a fichero de texto de la zona web

```
class MiServlet extends HttpServlet(……..,……..){

   void doPost (…..){

     FileReader fw=new FileReader("ficheros/fich.txt");  //MAL
     FileReader fw=new FileReader                        //BIEN
           (this. getServletContext().getRealPath("ficheros/fich.txt"));
   }
```

El método getRealPath() no es seguro ni útil para una aplicación real :

- Expone una ruta absoluta dentro del disco del servidor y
- Puede fallar en algunos entornos de producción: por ejemplo, si el contenedor del servlet no expande el archivo WAR al desplegarlo (como Tomcat). En esos casos, es mejor utilizar:

URL recurso=….getServletContext().getResource("archivo");

O si necesitamos un flujo de entrada a ese archivo:

InputStream contenido=….getServletContext().getResourceAsStream("archivo");