Sumario

Java Servlet

El **servlet** es una clase en el lenguaje de programación Java, utilizada para ampliar las capacidades de un servidor. Aunque los servlets pueden responder a cualquier tipo de solicitudes, estos son utilizados comúnmente para extender las aplicaciones alojadas por servidores web, de tal manera que pueden ser vistos como applets de Java que se ejecutan en servidores en vez de navegadores web. Este tipo de servlets son la contraparte Java de otras tecnologías de contenido dinámico Web, como PHP y ASP.NET.

La palabra *servlet* deriva de otra anterior, *applet*, que se refiere a pequeños programas que se ejecutan en el contexto de un navegador web.

El uso más común de los *servlets* es generar páginas web de forma dinámica a partir de los parámetros de la petición que envíe el navegador web.

Historia

La especificación original de Servlets fue creada por Sun Microsystems (la versión 1.0 fue terminada en junio de 1997). Comenzando con la versión 2.3, la especificación de Servlet fue desarrollada siguiendo el Proceso de la Comunidad Java (*Java Community Process*).

Para desarrollar aplicaciones Web a través del lenguaje de programación Java, hacemos el uso de la tecnología Servlet, la cual está diseñada para crear aplicaciones web.

Las aplicaciones web son aplicaciones auxiliares que residen en un servidor web y crean páginas web dinámicas. Una página dinámica puede ser como una página que elige al azar una imagen para mostrar o incluso una página que muestra la hora actual.

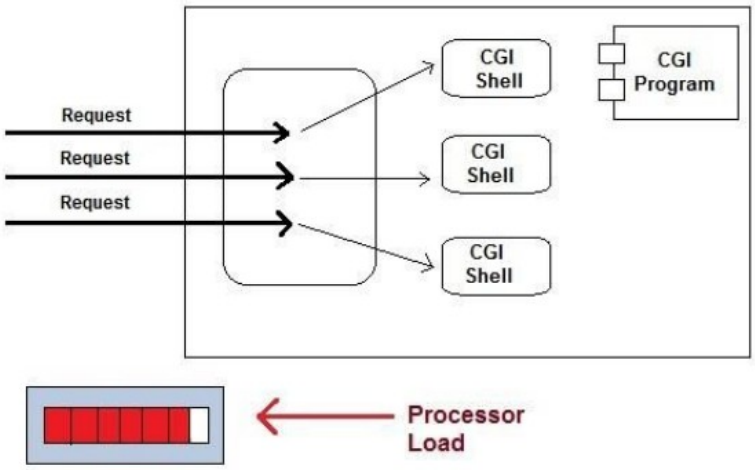
Antes de Java desarrollara la tecnología Servlets, se usaba la programación CGI (Common Gateway Interface) para crear aplicaciones web. Así es como funciona un programa CGI:

● El usuario hace clic en un enlace que tiene URL a una página dinámica en lugar de una página estática.

● La URL decide qué programa CGI ejecutar.

● Los servidores web ejecutan el programa CGI en un shell del sistema operativo separado. El shell incluye el entorno del sistema operativo y el proceso para ejecutar el código del programa CGI.

● La respuesta CGI se envía de vuelta al servidor web, que envuelve la respuesta en una respuesta HTTP y la envía de vuelta al navegador web.



Este tipo de desarrollo presenta las siguientes desventajas:

● Tiempo de alta resolución porque los programas CGI se ejecutan en su propio shell del sistema operativo.

● CGI no es escalable.

● Los programas CGI no siempre son seguros ni están orientados a objetos.

● Es dependiente de la plataforma.

Debido a estas desventajas, los desarrolladores comenzaron a buscar mejores soluciones. Y luego Java desarrolló Servlet como una solución sobre la tecnología CGI tradicional.

Un servlet es una clase de Java en J2EE que cumple con la API de Java Servlet, un estándar para implementar clases que responden a una arquitectura cliente servidor. Los servlets pueden, en principio, comunicarse a través de cualquier protocolo cliente-servidor, pero con mayor frecuencia se utilizan con HTTP. Por lo tanto, "servlet" se usa a menudo como abreviatura de "HTTP servlet". Un desarrollador de software puede usar un servlet para agregar contenido dinámico a un servidor web usando la plataforma Java. El contenido generado suele ser HTML, pero puede tratarse de otros datos, como XML. Los servlets pueden mantener variables de estado en sesión en muchas transacciones de servidor mediante el uso de cookies HTTP o reescritura de URL.

Para implementar y ejecutar un servlet, se debe usar un contenedor web. Un contenedor web (también conocido como contenedor de servlet) es esencialmente el componente de un servidor web que interactúa con los servlets. El contenedor web es responsable de administrar el ciclo de vida de los servlets, asignar una URL a un servlet determinado y garantizar que el solicitante de URL tenga los derechos de acceso correctos.

Ciclo de vida

1. Inicializar el servlet

Cuando un servidor carga un servlet, ejecuta el método `init` del servlet. El proceso de inicialización debe completarse antes de poder manejar peticiones de los clientes, y antes de que el servlet sea destruido.

Aunque muchos servlets se ejecutan en servidores *multi-thread*, los servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método `init` al crear la instancia del servlet, y no lo llamará de nuevo a menos que vuelva a recargar el servlet. El servidor no puede recargar un servlet sin primero haber destruido el servlet llamando al método `destroy`.

2. Interactuar con los clientes

Después de la inicialización, el servlet puede dar servicio a las peticiones de los clientes. Estas peticiones serán atendidas por la misma instancia del servlet, por lo que hay que tener cuidado al acceder a variables compartidas, ya que podrían darse problemas de sincronización entre requerimientos simultáneos.

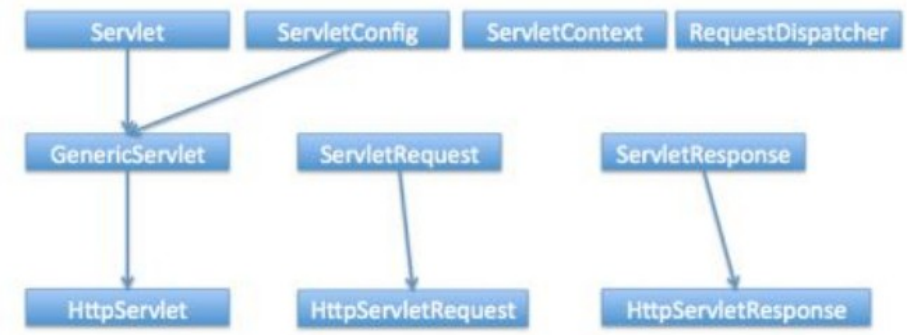
3. Destruir el servlet

Los servlets se ejecutan hasta que el servidor los destruye, por cierre del servidor o bien a petición del administrador del sistema. Cuando un servidor destruye un servlet, ejecuta el método `destroy` del propio servlet. Este método sólo se ejecuta una vez y puede ser llamado cuando aún queden respuestas en proceso, por lo que hay que tener la atención de esperarlas. El servidor no ejecutará de nuevo el servlet hasta haberlo cargado e inicializado de nuevo.

APIs Correspondientes a J2EE

Un Servlet se crea con el paquete `javax.servlet`.

javax.servlet.Servlet Es la interfaz base de Servlet API. Hay otras interfaces y clases que debemos tener en cuenta cuando trabajamos con Servlets. También con las especificaciones de Servlet 3.0, la API de servlet introdujo el uso de anotaciones en lugar de tener toda la configuración de servlet en el descriptor de implementación. El siguiente diagrama muestra la jerarquía de la API de servlet:



javax.ejb

La API Enterprise JavaBeans define un conjunto de APIs que un contenedor de objetos distribuidos soportará para suministrar persistencia, RPCs (usando RMI o RMI-IIOP), control de concurrencia, transacciones y control de acceso para objetos distribuidos.

java.transaction

Definen la Java Transaction API (JTA).

javax.jms

Estos paquetes definen la API JMS.

javax/persistence

Define las clases e interfaces para gestionar la interacción entre los proveedores de persistencia, las clases administradas y los clientes de la Java Persistence API (JPA).

API Servlet

Como se mencionó anteriormente, se necesita usar la API servlet para crear servlets. Hay dos paquetes que debe recordar al usar el API, el paquete javax.servlet que contiene las clases para admitir el servlet genérico (servlet independiente del protocolo) y el paquete javax.servlet.http que contiene clases para admitir el servlet http.

Generic Servlet

Cada servlet debe implementar la interfaz java.servlet.Servlet, puede hacerlo extendiendo una de las siguientes dos clases: javax.servlet.GenericServlet o javax.servlet.http.HttpServlet. El primero es para Servlet independiente del protocolo y el segundo para Servlet http.

Como se mencionó anteriormente, si está creando un Servlet genérico, debe extender la clase javax.servlet.GenericServlet. La clase GenericServlet tiene un método abstracto de servicio. Lo que significa que la subclase de GenericServlet siempre debe implementar el método service().

Firma del método service ():

El método service () acepta dos argumentos objeto ServletRequest y objeto ServletResponse. El objeto de solicitud le dice al servlet acerca de la solicitud realizada por el cliente, mientras que el objeto de respuesta se utiliza para devolver una respuesta al cliente.

HTTP Servlet

Si crea un Http Servlet, debe extender la clase javax.servlet.http.HttpServlet, que es una clase abstracta. A diferencia del servlet genérico, el servlet HTTP no sobre escribe el método service(). En cambio, sobre escribimos uno o más de los siguientes

métodos:

● **doGet:** El método de servicio servlet llama a este método para gestionar la solicitud HTTP GET del cliente. El método Get se usa para obtener información del servidor.

● **doPost:** Se utiliza para publicar información en el servidor.

● **doPut:** Este método es similar al método doPost pero a diferencia del método doPost donde enviamos información al servidor, este método envía archivos al servidor, esto es similar a la operación FTP de cliente a servidor.

● **doDelete:** Permite que un cliente elimine un documento, página web o información del servidor.

● **init y destroy:** Se utiliza para administrar recursos que se mantienen durante la vida del servlet.

● **getServletInfo:** Devuelve información sobre el servlet, como autor, versión y derechos de autor.

En Http Servlet no hay necesidad de reemplazar el método service ya que este método distribuye las solicitudes Http al manejador de método correcto, por ejemplo, si recibe HTTP GET Request envía la solicitud al método doGet.

Interfaces en el paquete javax.servlet

● Servlet

● ServletRequest

● ServletResponse

● ServletConfig

● ServletContext

● SingleThreadModel

● RequestDispatcher

● ServletRequestListener

● ServletRequestAttributeListener

● ServletContextListener

● ServletContextAttributeListener

● Filter

● FilterConfig

● FilterChain

Classes en el paquete javax.servlet

● GenericServlet

● ServletInputStream● ServletOutputStream

● ServletException

● ServletRequestWrapper

● ServletRequestEvent

● ServletResponseWrapper

● ServletContextEvent

- ServletRequestAttributeEvent
- ServletContextAttributeEvent
- UnavailableException

Interfaces en el paquete javax.servlet.http

- HttpSession
- HttpServletRequest
- HttpServletResponse
- HttpSessionAttributeListener
- HttpSessionListener
- HttpSessionBindingListener
- HttpSessionActivationListener
- HttpSessionContext

Classes en el paquete javax.servlet.http

- HttpServlet
- Cookie
- HttpSessionEvent
- HttpSessionBindingEvent
- HttpServletRequestWrapper
- HttpServletResponseWrapper
- HttpUtils

Interface HttpServletRequest

Representa el request del cliente.

Interface HttpServletResponse

Representa el responce del servidor.

Contenedores

El container proporciona un entorno de tiempo de ejecución para aplicaciones servlet. Un contenedor web es una aplicación predefinida proporcionada por un servidor para procesar aplicaciones: Servlet y JSP.

En las aplicaciones java basadas en consola, una clase que contiene el método principal actúa como un contenedor para otras clases, llamado main.

Entre las operaciones que efectúa un contenedor servlet tenemos:

1. Gestión del ciclo de vida
2. Soporte de comunicación
3. Soporte multiproceso
4. Seguridad, etc.

1. Gestión del ciclo de vida: Un Servlet y JSP son recursos dinámicos de aplicaciones web basadas en Java. El Servlet o JSP se ejecutará en un servidor y en el servidor un contenedor se ocupará de la vida y la muerte de un Servlet o JSP. Un contenedor instanciará, Inicializará, Servicio y destruirá un Servlet o JSP. Significa que el ciclo de vida será administrado por un contenedor.

2. Soporte de comunicación: Si el servlet o JSP quiere comunicarse con el servidor que necesita alguna lógica de comunicación como la programación de socket.

El diseño de la lógica de comunicación aumenta la carga de los programadores, pero el contenedor actúa como un mediador entre un servidor y un Servlet o JSP y proporciona comunicación entre ellos.

3. Multithreading: Un contenedor crea un hilo para cada solicitud, mantiene el hilo y finalmente lo destruye cada vez que se termina su trabajo.

4. Seguridad: No se requiere que un programador escriba código de seguridad en un Servlet / JSP. Un contenedor proporcionará automáticamente seguridad para un Servlet / JSP.Para utilizar Servlets y JSPs es necesaria la utilización de un "Contenedor de Servlets".

Existen muchos contenedores de Servlets como Apache Tomcat, Jboss, etc.

Estructura de una aplicación WEB

Para crear una aplicación web, debemos seguir la estructura de directorio estándar proporcionada por Oracle. De acuerdo con la estructura del directorio, una aplicación contiene una carpeta raíz con cualquier nombre y dentro de ella los siguientes recursos:

- En la carpeta raíz se necesita una subcarpeta con un nombre WEB-INF.
- En WEB-INF dos subcarpetas: classes y lib.
  - Todos los archivos jar requeridos, deben de ser colocados dentro de la carpeta lib.
  - Todas las clases requeridas (.class) deben de ser colocadas dentro de la carpeta classes.
  - En la carpeta raíz o en WEB-INF, puede haber cualquier otra carpeta o archivo.
  - Todos los archivos de imagen, html, .js, jsp, etc. se colocan dentro de la carpeta raíz

Web.xml

Es un archivo descriptor de despliegue de una aplicación web, contiene una descripción detallada sobre la aplicación web, como la configuración de Servlet, la gestión de sesiones, los parámetros de inicio, el archivo de bienvenida, etc.

No podemos cambiar el directorio ni el nombre de la extensión de este web.xml porque es un nombre estándar reconocido por el contenedor en tiempo de ejecución.

El archivo web.xml está presente dentro de la carpeta WEB-INF.

WEB-INF

Esta es una carpeta de inicialización de la aplicación web reconocida por el contenedor web en tiempo de ejecución, el nombre de la carpeta debe ser WEB-INF para desplegar la aplicación web con éxito, de lo contrario, el nombre del despliegue no tiene éxito.
En la carpeta lateral de WEB-INF, colocamos el archivo web.xml, la carpeta de clases, la carpeta lib y cualquier carpeta definida por el usuario.

Ejemplo

Código de un Servlet que procesa una petición GET y devuelve una página web HTML sencilla:

```
package org.pruebas;

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

public class HolaSextoInformaticaServlet extends HttpServlet {

    /**
     * Servlet de ejemplo que procesa una petición GET
     * @param request
     * @param response
     * @throws ServletException
     * @throws IOException
     */
    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//
EN\">");
        out.println("<html>");
        out.println("<head><title>Ejemplo HolaSextoInformatica</title></head>");
        out.println("<body>");
        out.println("<h1>¡Hola HolaSextoInformatica!</h1>");
        out.println("</body></html>");
    }
}
```

Servlet GetPost

Vamos a ver un ejemplo, un poco más complicado. Si llamamos un servlet desde un formulario HTML, podremos hacerlo de dos formas: **GET** y **POST**. Con la primera los parámetros del formulario están incluidos la url que se utiliza para invocar el servlet y en el segundo caso los parámetros se almacenan en un buffer especial del servidor.

Para procesar el primer tipo de peticiones (GET) está el método **doGet** mientras que para el segundo tenemos el método **doPost** . La implementación por defecto del método **service** es capaz de determinar el tipo de petición HTTP que en un momento dado recibe el servlet. Una vez identificada llama o al método **doGet** o al **doPost** según el caso. Como, en la mayoría de los casos, seremos nosotros quien programe el formulario que llame al servlet, sabremos que tipo de llamada se hará, por lo que podemos optar por redefinir uno sólo de los métodos. En el caso de que no lo supiéramos se deben implementar los métodos *doGet* y *doPost*.

Un servlet que tiene diferente respuesta en función de la llamada que se le hace es el ejemplo **GetPost.java** cuyo código fuente era el siguiente:

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

public class GetPost extends HttpServlet {

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

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
  {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<body>");
    out.println("<h1>Hola Mundo (llamada GET)</h1>");
    out.println("</body>");
    out.println("</html>");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
  {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<body>");
    out.println("<h1>Hola Mundo (llamada POST)</h1>");
    out.println("</body>");
    out.println("</html>");
  }
}
```

Para llamar al servlet con un tipo de llamada get, podemos usar el ejemplo **get.html**. Que podemos colocar en el directorio **C:\java\tomcat\webapps\tutorial** y cuyo código es:

```
<html>
<body>
   <h1>Método GET</h1>
   <form method="GET" action="/tutorial/servlet/GetPost">
      <input type="submit">
   </form>
</body>
</html>
```

Para llamar al servlet con un tipo de llamada post, podemos usar el ejemplo **post.html**. Que podemos colocar en el directorio **C:\java\tomcat\webapps\tutorial** y cuyo código es:

```
<html>
<body>
   <h1>Método POST</h1>
   <form method="POST" action="/tutorial/servlet/GetPost">
      <input type="submit">
   </form>
</body>
</html>
```

Ciclo de vida de un Servlet

Como hemos visto antes, cuando se crea un servlet, el servidor llama al método **init** y cada vez que un cliente acceda al servlet el servidor llamará al método **service** que se encargará de redirigir la llamada **doGet** o a **doPost**. Esto nos quiere decir que cuando se llama por primera vez a un servlet se ejecutara primero **init** y después **service**, pero ... ¿Y la segunda vez y sucesivas también se llama a **init** o sólo a **service**?.

Normalmente, el servidor crea el servlet (llama, por tanto, al método **init**) y lo mantiene funcionando, si ha pasado un tiempo suficiente (y que puede ir desde unos segundos a nunca) sin que el servlet se llame lo deja de ejecutar. Es decir, un servlet se empieza a ejecutar con la primera llamada y, normalmente, se seguirá ejecutando.

De esta forma, vamos a crear un servlet que cuente el número de visitas que recibe, para ello nos bastará crear una variable contador que inicializaremos en el método **init** y que incrementaremos en **doPost/doGet**. Por lo que, el contador se inicializará cuando se llame por primera vez al servlet y se irá incrementando en llamadas sucesivas. Este ejemplo, lo llamaremos **Contador.java** y su código fuente era el siguiente:

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

public class Contador extends HttpServlet {
  //variable contador
  int contador;

  public void init(ServletConfig conf)
    throws ServletException {
    super.init(conf);

    //inicializamos la variable contador
    contador = 1;
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
```

```
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    int tmp = contador;

    //incrementamos la variable contador
    contador++;

    out.println("<html>");
    out.println("<body>");
    out.println("<h1>Numero de peticiones " + tmp +"</h1>");
    out.println("</body>");
    out.println("</html>");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    doGet(req, res);
  }
}
```

NOTA: Observa que un mismo servlet puede ser llamado por más de un cliente a la vez. En este caso, el servidor crea una hebra del servlet por cada petición y esas dos hebras accederán concurrentemente a los mismos datos (la variable contador). Como la lectura e incremento de contador no es una operación atómica, se podría utilizar la primitiva de sincronización *syncronized* para que se realice de forma atómica:

```
    .....
    PrintWriter out = res.getWriter();
    int tmp;

    synchronized(this) {
      //leemos el contador
      tmp = contador;
      //incrementamos la variable contador
      contador++;
    }

    out.println("<html>");
    .....
```

Parámetros desde un formulario HTML

Normalmente los servlets tendrán parámetros o fuentes de información que le darán su aspecto dinámico. Es decir, para generar una simple página HTML no nos complicamos tanto la vida, se escribe la página HTML y se ha terminado. Las fuentes de información de las que un servlet hace uso, pueden ser varias: el propio servlet, el servidor web, ficheros o bases de datos a los que pueda acceder o parámetros que le pase el cliente. De todas estas fuentes, nos interesan los accesos a bases de datos que veremos más adelante y los pa;rámetros que nos pasa el cliente mediante formularios HTML.

Cuando pasamos parámetros a través de un formulario, en los Servlets a través de la clase *ServletRequest*, disponemos de los siguientes métodos para su tratamiento:

- **String getParameter(String nombre)**: Nos devuelve el valor del parámetro cuyo nombre le indicamos. Si la variable no existiera o no tuviese ningún valor, devolvería *null*
- **Enumerate getParameterNames()**: Nos devuelve una enumeración de los nombres de los parámetros que recibe el servlet.

- **Enumerate getParameterValues(String)**: Nos devuelve los valores que toma un parámetro dado, esto es útil para listas de selección múltiple donde un parámetro tiene más de un valor.

Veamos un ejemplo de un pequeño formulario que tenga distintos tipos de parámetros, se los envíe a nuestro servlet y éste los muestre por pantalla (aún no sabemos guardarlos en la base de datos). El formulario los llamaremos **formulario.html**. Que podemos colocar en el directorio **C:\java\tomcat\webapps\tutorial** y cuyo código es:

```
<html>
<title>Formulario de ejemplo</title>
<body>
   <h1>Formulario</h1>
   <form method="POST" action="/tutorial/servlet/Parametros">
     Nombre: <INPUT TYPE="TEXT" NAME="nombre"><BR>
     Primer Apellido:<INPUT TYPE="TEXT" NAME="apellido1"><BR>
     Segundo Apellido:<INPUT TYPE="TEXT" NAME="apellido2"><BR>
     <hr>
     Correo electronico: <INPUT TYPE="TEXT" NAME="email"><BR>
     Clave: <INPUT TYPE="PASSWORD" NAME="clave"><BR>
     <hr>
     Comentario: <TEXTAREA NAME="comenta" ROWS=3 COLS=40>
                 </TEXTAREA><BR>
     <hr>
     Sexo:<BR>
          <INPUT TYPE="RADIO" NAME="sexo" VALUE="hombre">Hombre<BR>
          <INPUT TYPE="RADIO" NAME="sexo" VALUE="mujer">Mujer<BR>

     Areas de interés:<br>
          <SELECT NAME="intereses" MULTIPLE>
                <OPTION>Informatica</OPTION>
                <OPTION>Derecho</OPTION>
                <OPTION>Matematicas</OPTION>
                <OPTION>Fisica</OPTION>
                <OPTION>Musica</OPTION>
          </SELECT>

      <center><input type="submit" value="Enviar"></center>
   </form>
</body>
</html>
```

Veamos primero un servlet que conociendo de antemano los distintos parámetros que va a recibir los vaya mostrando en una página HTML. El servlet lo llamaremos **Parametros.java** y su código es:

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

public class Parametros extends HttpServlet {

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

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
```

```
    out.println("<html>");
    out.println("<body>");
    out.println("<h1>Parámetros del servlet desde un formulario HTML</h1>");
    out.println("<br> Nombre:"+req.getParameter("nombre") );
    out.println("<br> Primer apellido:"+req.getParameter("apellido1") );
    out.println("<br> Segundo apellido:"+req.getParameter("apellido2") );
    out.println("<br> Correo electrónico:"+req.getParameter("email") );
    out.println("<br> Contraseña:"+req.getParameter("clave") );
    out.println("<br> Comentario:"+req.getParameter("comenta") );
    out.println("<br> Sexo:"+req.getParameter("sexo") );
    out.println("<br> Areas de interés:"+req.getParameter("intereses") );
    out.println("</body>");
    out.println("</html>");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    doGet(req, res);
  }
}
```

Si en la lista de de selección múltiple escogemos más de un valor en la implementación actual sólo mostraría la primera elección, si quisiéramos mostrar todos los valores deberíamos de usar **getParameterValues("intereses")** e ir recorriendo y mostrando cada uno de los valores seleccionados del parámetro *intereses*.

Otra posible implementación del servlet **Parametros.java** sería uno que mostrase los parámetros y sus valores sin tener que conocerlos previamente. El código sería:

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


public class Parametros extends HttpServlet
{
  public void init(ServletConfig conf)
        throws ServletException
  {
        super.init(conf);
  }

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

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

    //Escribimos el principio de la página HTML
    out.println("<html>");
    out.println("<body>");
    out.println("<h1>Parámetros del servlet desde un formulario HTML</h1>");

    //cogemos los nombres de los parametros
    Enumeration paramNames = request.getParameterNames();

    //vamos mostrando los parámetros en unwhile
    while(paramNames.hasMoreElements()) {
      //cogemos el siguiente parámetro
      String paramName = (String)paramNames.nextElement();
```

```
      //Mostramos el nombre del parámetro
      out.print(paramName + " = ");

      //Cogemos los valores del parámetro
      String[] paramValues = request.getParameterValues(paramName);

      //Miramos si tiene más de un valor
      if (paramValues.length == 1) {
        //Si tiene un sólo valor, miramos si está vacío o no
        String paramValue = paramValues[0];
        if (paramValue.length() == 0)
          out.println("<i>Sin valor</i><br>");
        else
          out.println(paramValue + "<br>");
      }
      else {
        //Si tiene más de un sólo valor, los mostramos
        for(int i=0; i<paramValues.length; i++)
          out.println(paramValues[i] + ", ");
        out.println("<br>");
      }
    }//end while

    //Escribimos el final de la página HTML
    out.println("</body>");
    out.println("</html>");
  }

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

Interface Servlet

javax.servlet.Servlet es la interfaz base de Java Servlet API. La interfaz del servlet declara los métodos del ciclo de vida del servlet. Todas las clases de servlets son necesarias para implementar esta interfaz. Los métodos declarados en esta interfaz son:

● public abstract void init (ServletConfig paramServletConfig) throws ServletException: Este es un método muy importante invocado por el contenedor de servlets para inicializar el servlet y los parámetros de ServletConfig. El servlet no está listo para procesar la solicitud del cliente hasta que el método init () haya terminado de ejecutarse. Este método se llama solo una vez en el ciclo de vida del servlet y hace que la clase Servlet sea diferente de los objetos normales de Java. Podemos extender este método en nuestras clases de servlet para inicializar recursos como conexión de DB, conexión de socket, etc.

● public abstract ServletConfig getServletConfig (): Este método devuelve un objeto de configuración del servlet que contiene los parámetros de inicialización y la configuración de inicio para este servlet. Podemos utilizar este método para obtener los parámetros init de las definiciones de servlet en el descriptor de despliegue (web.xml) o mediante la anotación en Servlet 3.

● public void service (ServletRequest req, ServletResponse res) throws ServletException, IOException: Este método es responsable de procesar la solicitud del cliente. Cada vez que el contenedor de servlet recibe una solicitud, crea un nuevo hilo y ejecuta el método service () pasando

la solicitud y la respuesta como argumento. Los servlets generalmente se ejecutan en un entorno de subprocesos múltiples, por lo que es responsabilidad del desarrollador mantener los recursos compartidos seguros para subprocesos mediante la sincronización.

● public abstract String getServletInfo (): Este método devuelve una cadena que contiene información sobre el servlet, como su autor, versión y copyright. La cadena devuelta debe ser texto sin formato y no puede tener marcas.

● public abstract void destroy (): Este método solo se puede llamar una vez en el ciclo de vida del servlet y se usa para cerrar cualquier recurso abierto. Esto es como el método de finalización de una clase Java.

Interface ServletConfig

javax.servlet.ServletConfig se usa para pasar información de configuración a un Servlet. Cada servlet tiene su propio objeto ServletConfig y el contenedor de servlets es responsable de crear instancias de este objeto. Podemos proporcionar los parámetros de servlet init en el archivo web.xml o mediante el uso de la anotación WebInitParam. Podemos usar el método getServletConfig () para obtener el objeto ServletConfig del servlet.
Los métodos importantes de la interfaz ServletConfig son:

● public abstract ServletContext getServletContext (): Este método devuelve el objeto ServletContext para el servlet.

● public abstract Enumeration <String> getInitParameterNames (): Este método devuelve la Enumeración <String> del nombre de los parámetros init definidos para el servlet. Si no hay parámetros de init definidos, este método devuelve la enumeración vacía.

● public abstract String getInitParameter (String paramString): Este método se puede usar para obtener el valor específico del parámetro init por su nombre. Si el parámetro no está presente con el nombre, devuelve nulo.

Interface ServletContext

La interface javax.servlet.ServletContext proporciona acceso a las variables de la aplicación web al servlet. El ServletContext es un objeto único y está disponible para todos los servlets en la aplicación web. Cuando queremos que algunos parámetros init estén disponibles para múltiples o todos los servlets en la aplicación web, podemos usar el objeto ServletContext y definir parámetros en web.xml usando el elemento <context-param>. Podemos obtener el objeto ServletContext a través delmétodo getServletContext () de ServletConfig. Los motores de servlet también pueden proporcionar objetos de contexto que son únicos para un grupo de servlets y que están vinculados a una parte específica del espacio de nombres de la ruta de URL del host.
Algunos de los métodos importantes de ServletContext son:

● public abstract ServletContext getContext (String uripath): Este método devuelve el objeto ServletContext para un uripath determinado o nulo si no está disponible o no está visible para el servlet.

● public abstract URL getResource (String path) MalformedURLException: Este método devuelve el objeto URL que permite el acceso a cualquier recurso de contenido solicitado. Podemos acceder a los elementos ya sea que residan en el sistema de archivos local, un sistema de archivos remoto, una

base de datos o un sitio de red remota sin conocer los detalles específicos de cómo obtener los recursos.

● public abstract InputStream getResourceAsStream (String path): Este método devuelve un flujo de entrada a la ruta de acceso del recurso o nulo si no se encuentra.

● public abstract RequestDispatcher getRequestDispatcher (String urlpath): Este método se utiliza principalmente para obtener una referencia a otro servlet. Después de obtener un RequestDispatcher, el programador de servlets reenvía una solicitud al componente de destino o incluye contenido de él.

● public void log (String msg): Este método se utiliza para escribir una cadena de mensaje dada en el archivo de registro del servlet.

● Object público abstracto getAttribute (String name): Devuelve el atributo de objeto para el nombre de pila. Podemos obtener la enumeración de todos los atributos usando el método público abstracto Enumeration<String> getAttributeNames ().

● public abstract void setAttribute (String paramString, Object paramObject): Este método se usa para establecer el atributo con el alcance de la aplicación. El atributo será accesible para todos los demás servlets que tengan acceso a este ServletContext. Podemos eliminar un atributo usando el método public abstract void removeAttribute (String paramString).

● String getInitParameter (String name): Este método devuelve el valor String para el parámetro init definido con el nombre en web.xml, devuelve null si elnombre del parámetro no existe. Podemos utilizar Enumeration<String> getInitParameterNames () para obtener la enumeración de todos los nombres de los parámetros init.

● boolean setInitParameter (String paramString1, String paramString2): Podemos usar este método para establecer los parámetros init para la aplicación. Idealmente, el nombre de esta interfaz debe ser ApplicationContext porque es para la aplicación y no es específico de ningún servlet. Además, no se confunda con el contexto de servlet pasado en la URL para acceder a la aplicación web.

Interface ServletRequest

La interface ServletRequest se usa para proporcionar información de solicitud del cliente al servlet. El contenedor de servlet crea el objeto ServletRequest a partir de la solicitud del cliente y lo pasa al método servlet service () para su procesamiento.
Algunos de los métodos importantes de la interfaz ServletRequest son:

● Object getAttribute (String name): Este método devuelve el valor del atributo named como Object y null si no está presente. Podemos usar el método getAttributeNames () para obtener la enumeración de los nombres de atributos para la solicitud. Esta interface también proporciona métodos para establecer y eliminar atributos.

● String getParameter (String name): Este método devuelve el parámetro de solicitud como String. Podemos usar el método getParameterNames () para obtener la enumeración de nombres de parámetros para la solicitud.
● String getServerName(): Devuelve el nombre de host del servidor.
● int getServerPort(): Devuelve el número de puerto del servidor en el que está escuchando.
La interface hija de ServletRequest es HttpServletRequest que contiene algunos otros métodos para la gestión de sesión, las cookies y la autorización de solicitud.Interface ServletResponse

El servlet utiliza la interface ServletResponse para enviar una respuesta al cliente. El contenedor de servlet crea el objeto ServletResponse y lo pasa al método servlet service() y luego utiliza el objeto de respuesta para generar la respuesta HTML para el cliente.
Algunos de los métodos importantes en HttpServletResponse son:

1. void addCookie (Cookie Cookie): Se usa para agregar cookies a la respuesta.

2. void addHeader (String name, String value): Se usa para agregar un encabezado de respuesta con el nombre y el valor.

3. String encodeURL (java.lang.String url): Codifica la URL especificada al incluir la ID de la sesión en ella o si la codificación no es necesaria, devuelve la URL sin cambios.

4. String getHeader (String name): Devuelve el valor para el encabezado especificado, o null si este encabezado no se ha establecido.

5. void sendRedirect (String location): Se usa para enviar una respuesta de redireccionamiento temporal al cliente utilizando la URL de ubicación de redireccionamiento especificada.

6. void setStatus (int sc): Utilizado para establecer el código de estado para la respuesta.

Interface de RequestDispatcher

La interface RequestDispatcher se usa para reenviar la solicitud a otro recurso que puede ser HTML, JSP u otro servlet en el mismo contexto. También podemos usar esto para incluir el contenido de otro recurso en la respuesta. Esta interface se utiliza para la comunicación de servlets dentro del mismo contexto.
Hay dos métodos definidos en esta interfaz:

● void forward (solicitud ServletRequest, respuesta ServletResponse):
Reenvía la solicitud de un servlet a otro recurso (servlet, archivo JSP o archivo HTML) en el servidor.

● void include (solicitud ServletRequest, respuesta ServletResponse): Incluye el contenido de un recurso (servlet, página JSP, archivo HTML) en la respuesta.

Podemos obtener RequestDispatcher en un servlet usando el método ServletContext getRequestDispatcher (String path). La ruta debe comenzar con / y se interpreta como relativa a la raíz de contexto actual.

Clase HTTPServlet

HTTPServlet es una clase abstracta que amplía GenericServlet y proporciona una base para crear aplicaciones web basadas en HTTP. Hay métodos definidos para ser reemplazados por subclases para diferentes métodos HTTP.

● **doGet():** Para solicitudes HTTP GET
● **doPost():** Para solicitudes HTTP POST
● **doPut():** Para solicitudes HTTP PUT
● **doDelete():** Para solicitudes HTTP DELETE

Atributos de un servlet

Los atributos del servlet se utilizan para la comunicación entre servlets, podemos establecer, obtener y eliminar atributos en la aplicación web. Hay tres ámbitos para los atributos de servlet:

- Alcance de la solicitud (Request)
- Alcance de la sesión (Session)
- Alcance de la aplicación (Application)

Las interfaces ServletRequest, HttpSession y ServletContext proporcionan métodos para obtener / establecer / eliminar atributos de la solicitud, la sesión y el alcance de la aplicación, respectivamente.
Los atributos del servlet son diferentes de los parámetros init definidos en web.xml para ServletConfig o ServletContext.

Anotaciones en Servlet 3

Antes de Servlet 3, todos los mapas de servlet y sus parámetros init se usaban para definirse en web.xml, esto no era conveniente y más propenso a errores cuando el número de servlets es enorme en una aplicación.
El servlet 3 introdujo el uso de anotaciones java para definir un servlet, filtros y servlets de oyente y parámetros init.
Algunas de las anotaciones importantes de Servlet son:

- WebServlet: Podemos usar esta anotación con clases de servlet para definir los parámetros de init, loadOnStartup value, description y url patterns etc.
DEBE declararse al menos un patrón de URL en el valor o en el atributo urlPattern de la anotación, pero no en ambos. La clase en la que se declara esta anotación DEBE extender HttpServlet.

- WebInitParam: Esta anotación se usa para definir los parámetros init para servlet o filter, contiene nombre, par de valores y también podemos proporcionar una descripción. Esta anotación se puede usar dentro de una anotación WebFilter o WebServlet.

- WebFilter: Esta anotación se utiliza para declarar un filtro de servlet. Esta anotación es procesada por el contenedor durante la implementación, la clase de Filtro en la que se encuentra se creará según la configuración y se aplicará a los patrones de URL, Servlets y DispatcherTypes. La clase anotada DEBE implementar la interfaz javax.servlet.Filter.

- WebListener: La anotación utilizada para declarar un oyente para varios tipos de eventos, en un contexto de aplicación web dado.

Manejo de sesiones

¿Qué es una sesión?
El protocolo HTTP y los servidores web son sin estado, lo que significa que para el servidor web cada solicitud es una nueva solicitud de proceso y no pueden identificar si proviene del cliente que ha estado enviando la solicitud anteriormente.
Pero a veces en las aplicaciones web, debemos saber quién es el cliente y procesar la solicitud en consecuencia. Por ejemplo, una aplicación de carrito de compras debe saber quién está enviando la solicitud para agregar un artículo y en qué carro se debe agregar el artículo o quién envía una solicitud de pago para que pueda cargar el monto al cliente correcto.

La sesión es un estado de conversión entre el cliente y el servidor y puede consistir en múltiples solicitudes y respuestas entre el cliente y el servidor. Como HTTP y Web Server son apátridas, la única forma de mantener una sesión es cuando se pasa información única sobre la sesión (id. De sesión) entre el servidor y el cliente en cada solicitud y respuesta.
Hay varias maneras a través de las cuales podemos proporcionar un identificador único en la solicitud y la respuesta:

● Autenticación de usuario: Esta es la forma más común en la que el usuario puede proporcionar credenciales de autenticación desde la página de inicio de sesión y luego podemos pasar la información de autenticación entre el servidor y el cliente para mantener la sesión. Este no es un método muy efectivo porque no funcionará si el mismo usuario está conectado desde diferentes navegadores.

● Campo oculto de HTML: Podemos crear un campo oculto único en el HTML y cuando el usuario comienza a navegar, podemos establecer su valor exclusivo para el usuario y realizar un seguimiento de la sesión. Este método no se puede usar con enlaces porque necesita enviar el formulario cada vez que se realiza una solicitud de cliente a servidor con el campo oculto. Tampoco es seguro porque podemos obtener el valor de campo oculto de la fuente HTML y usarlo para hackear la sesión.

● Reescritura de URL: Podemos agregar un parámetro de identificador de sesión con cada solicitud y respuesta para realizar un seguimiento de la sesión. Esto es muy tedioso porque necesitamos hacer un seguimiento de este parámetro en cada respuesta y asegurarnos de que no esté en conflicto con otros parámetros.

● Cookies: Las cookies son una pequeña porción de información que el servidor web envía en el encabezado de respuesta y se almacena en las cookies del navegador. Cuando el cliente realiza una solicitud adicional, agrega la cookie al encabezado de solicitud y podemos utilizarla para realizar un seguimiento de la sesión. Podemos mantener una sesión con las cookies, pero si el cliente desactiva las cookies, no funcionará.

● API de administración de sesión: La API de administración de sesión se basa en los métodos anteriores para el seguimiento de la sesión. Algunas de las principales desventajas de todos los métodos anteriores son:

    ○ La mayoría de las veces no queremos solo rastrear la sesión, tenemos que almacenar algunos datos en la sesión que podemos usar en futuras solicitudes. Esto requerirá un gran esfuerzo si tratamos de implementar esto.

    ○ Todos los métodos anteriores no son completos en sí mismos, ninguno de ellos funcionará en un escenario particular. Por lo tanto, necesitamos una solución que pueda utilizar estos métodos de seguimiento de sesión para proporcionar administración de la sesión en todos los casos.

Es por eso que se necesita un API de administración de sesiones y la tecnología de servlets J2EE viene con la API de administración de sesiones que podemos usar.

HttpSession

El API Servlet proporciona administración de sesión a través de la interfaz HttpSession. Podemos obtener la sesión del objeto HttpServletRequest usando los siguientes métodos. HttpSession nos permite establecer objetos como atributos que se pueden recuperar en solicitudes futuras.

● HttpSession getSession (): Este método siempre devuelve un objeto HttpSession. Devuelve el objeto de sesión adjunto con la solicitud, si la solicitud no tiene una sesión adjunta, crea una nueva sesión y la devuelve.

● HttpSession getSession (indicador booleano): Este método devuelve el objeto HttpSession sí request tiene session else devuelve null.

Algunos de los métodos importantes de HttpSession son:

● String getId (): Devuelve una cadena que contiene el identificador único asignado a esta sesión.

● Object getAttribute (String name): Devuelve el objeto enlazado con el nombre especificado en esta sesión, o nulo si ningún objeto está vinculado bajo el nombre. Algunos otros métodos para trabajar con los atributos de sesión son getAttributeNames (), removeAttribute (String name) y setAttribute (String name, Object value).

● long getCreationTime (): Devuelve la hora en que se creó esta sesión, medida en milisegundos desde la medianoche del 1 de enero de 1970 GMT. Podemos obtener el último tiempo de acceso con el método getLastAccessedTime ().

● setMaxInactiveInterval (int interval): Especifica el tiempo, en segundos, entre las solicitudes del cliente antes de que el contenedor de servlets invalide esta sesión. Podemos obtener el valor de tiempo de espera de la sesión del método getMaxInactiveInterval().

● ServletContext getServletContext (): Devuelve el objeto ServletContext para la aplicación.

● boolean isNew (): Devuelve true si el cliente aún no conoce la sesión o si el cliente elige no unirse a la sesión.

● void invalidate (): Invalida esta sesión y luego desvincula los objetos vinculados a ella.

Comprender la cookie JSESSIONID

Cuando usamos el método HttpServletRequest getSession () y crea una nueva solicitud, crea el nuevo objeto HttpSession y también agrega una Cookie al objeto de respuesta con el nombre JSESSIONID y valor como ID de sesión. Esta cookie se usa para identificar el objeto HttpSession en futuras solicitudes del cliente. Si las cookies están desactivadas en el lado del cliente y estamos utilizando la reescritura de URL, este método utiliza el valor jsessionid de la URL de solicitud para encontrar la sesión correspondiente. La cookie JSESSIONID se utiliza para el seguimiento de la sesión, por lo que no deberíamos utilizarla para nuestros fines de aplicación para evitar problemas relacionados con la sesión.

Reescritura de URL

Como se explico, se puede administrar una sesión con HttpSession, pero si desactivas las cookies en el navegador, no funcionará porque el servidor no recibirá la cookie JSESSIONID del cliente. La API Servlet proporciona soporte para la reescritura de URL que podemos usar para gestionar la sesión en este caso. Desde el punto de vista de la codificación, es muy fácil de usar e implica un paso: codificar la URL. Otra cosa buena con Servlet URL Encoding es que se trata de un enfoque alternativo e inicia sólo si las cookies del navegador están desactivadas.

Podemos codificar URL con el método HttpServletResponse encodeURL () y si tenemos que redirigir la solicitud a otro recurso y queremos proporcionar información de la sesión, podemos usar el método encodeRedirectURL().